Approche algorithmique pour l'amélioration des performances du système de détection d'intrusions PIGA

Pierre Clairet

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHEMATIQUES, INFORMATIQUE, PHYSIQUE THEORIQUE et INGENIERIE DES SYSTEMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS

THÈSE présentée par :

Pierre CLAIRET

soutenue le : **24 juin 2014**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **INFORMATIQUE**

Approche algorithmique pour l'amélioration des performances du système de détection d'intrusions PIGA

THÈSE dirigée par :

Pascal BERTHOMÉ	Professeur, INSA Centre Val de Loire

RAPPORTEURS :

Dominique BARTH	Professeur, Université de Versailles
Christophe ROSENBERGER	Professeur, ENSICAEN

JURY :

Dominique BARTH	Professeur des Universités, Université Versailles Saint Quentin
Pascal BERTHOMÉ	Professeur des Universités, INSA CVL
Jérémy BRIFFAUT	Maître de conférences, INSA CVL
Fabien DE MONTGOLFIER	Maître de conférences, Université Paris 7
Sébastien LIMET	Professeur des Universités, Université d'Orléans
Christophe ROSENBERGER	Professeur des Universités, ENSICAEN

Remerciements

Je remercie Dominique Barth et Christophe Rosenberger d'avoir accepté d'être mes rapporteurs et ainsi d'évaluer mes travaux de thèse.

Je remercie Jérémy Briffaut, Fabien De Montgolfier et Sébastien Limet d'avoir accepté de faire partie de mon jury de soutenance.

Je tiens également à remercier mon directeur de thèse, Pascal Berthomé, pour son soutien durant ces quatre années de thèse, mais aussi le Laboratoire d'Informatique Fondamentale d'Orléans, notamment l'équipe Sécurité et Distribution des Systèmes, ainsi que l'INSA Centre Val de Loire pour son accueil.

Je remercie Christian Toinard et ma compagne Aurélie Leday pour avoir pris le temps de relire ce manuscrit.

Merci à Jérémy pour la documentation très fournie de PIGA.

Merci à Damien Gros, Maxime Fonda et Jonathan Rouzaud-Cornabas pour l'ambiance dans le bureau des doctorants pendant ma première année de thèse.

Merci à Damien, Maxime, Thibault, Aurélie, André, Christel, Adam, Céline, Momo, Aurélien, Guillaume, Perrine ainsi que la Brasserie BOS pour toutes les soirées et vacances qui ont animé mes quatre années de thèse.

Merci ma famille, à Aurélie et à mon chat de m'avoir soutenu tout au long de ma thèse.

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Introduction

Du début de la démocratisation d'Internet à aujourd'hui, de *Datacrime* à l'affaire *Snowden*, la sécurité des systèmes informatiques a toujours été un sujet d'actualité. De nos jours, avec l'utilisation massive d'Internet et des périphériques de stockage externes, les affaires de piratages ou d'attaques informatiques se multiplient. Elles touchent les entreprises avec des attaques par déni de service (DOS et DDOS) ou des vols de données. Par exemple, Sony Online Entertainment en 2011 ou Orange en 2014 ont été victimes de vols de données, tandis que le site WikiLeaks a été victime d'une attaque de déni de service en 2012.

Mais les attaques informatiques sont maintenant utilisées par les états contre d'autres états. Une des premières attaques contre un état eut lieu en 2007 contre l'Estonie. Elle rendit notamment indisponibles les sites gouvernementaux et les sites des banques. Plus récemment, le virus *Stuxnet* aurait eu pour objectif de mettre une centrale nucléaire iranienne hors service. Que ce soit contre une entreprise ou contre un état, les conséquences économiques des cyberattaques peuvent être très importantes.

Ces cyberattaques profitent souvent de vulnérabilités des logiciels présents sur les systèmes informatiques. Or, les systèmes d'exploitation et leurs applications sont de plus en plus complexes. Cette augmentation de complexité a pour effet de multiplier les moyens de s'introduire dans le système. Ainsi, les comportements intrusifs sont nombreux et peuvent être très différents. Les systèmes de détection d'intrusions ont pour objectif de détecter toute tentative d'intrusion sur un système d'exploitation. Parmi ces systèmes de détection d'intrusions, PIGA a la particularité de générer une signature pour chaque comportement intrusif possible sur un système d'exploitation. Une fois ces signatures générées, il est possible de détecter les tentatives d'intrusion. Cependant, l'augmentation du nombre de comportements intrusifs possibles conduit à générer une base de signatures de plus en plus volumineuse. Celle-ci devant être stockée en mémoire lors de la détection, les performances du système d'exploitation, sur lequel PIGA est utilisé, s'en trouvent amoindries.

L'objectif de cette thèse est de réduire la quantité de mémoire utilisée par PIGA pour la détection d'intrusion. Nous avons réduit ce problème, à une diminution du volume de la base de signatures, son stockage étant la principale source d'utilisation de la mémoire. Pour cela, nous proposons deux méthodes. La première réduit les graphes à partir desquels les signatures sont générées. La seconde réduit directement la base de signatures en supprimant des signatures inutiles suivant le contexte.

Ce manuscrit est composé de deux parties regroupant quatre chapitres. La première partie décrit le fonctionnement des deux outils avec lesquels nous avons travaillé. La seconde partie présente les deux grandes étapes de notre contribution.

Le premier chapitre décrit le fonctionnement de PIGA. Il présente, notamment, les mécanismes de génération de signatures et de détection. Nous y définissons également la problématique principale de cette thèse : la réduction de la base de signatures. Enfin, nous proposons deux solutions différentes à ce problème.

Le deuxième chapitre présente la décomposition modulaire. Cette décomposition est un outil de la théorie des graphes pouvant être utilisé pour réduire un graphe en minimisant les pertes d'informations. Ce chapitre présente, en particulier, les différentes structures liées à la décomposition modulaire, ainsi que le fonctionnement des algorithmes l'appliquant. Nous montrons également l'intérêt de cet outil pour notre problématique.

Le troisième chapitre détaille la première partie de notre contribution : la modification du système de génération de signatures. Nous y présentons nos deux méthodes pour réduire le nombre de signatures : une application de la décomposition modulaire et la suppression par inclusion. Nous montrons que le nouveau système décrit les mêmes comportements, sans perte ou ajout d'informations, dans le cas de système dit « propre ». Dans les autres cas, les comportements ajoutés ne sont pas signifiants pour le système et permettent ainsi d'avoir une bonne réduction de la taille de la base. Ensuite, nous détaillons plusieurs expérimentations afin d'évaluer ces méthodes. Enfin, nous montrons les limites d'utilisation de ces méthodes.

Le dernier chapitre décrit la deuxième partie de notre contribution : la modification du système de détection. Nous y détaillons les nouveaux cas engendrés par la nouvelle forme des signatures, ainsi qu'un algorithme de détection permettant de gérer ces cas. Puis, nous évaluons les performances, en termes de temps d'exécution et de consommation mémoire, de notre nouveau système de détection.

Nous présentons ensuite une conclusion dans laquelle nous proposons différentes perspectives à notre travail.

Première partie

État de l'art

Chapitre 1

PIGA

1.1 Fonctionnement général

Dans cette section, nous verrons tout d'abord comment PIGA se place par rapport aux autres systèmes de détection d'intrusions. Nous présenterons les types de systèmes de détection d'intrusions, les principales méthodes de détection et les critères d'évaluation de l'efficacité de la détection. Nous reviendrons sur certaines notions de sécurité telles que les propriétés de sécurité ou les bases de comportements. Enfin, nous expliquerons le principe de fonctionnement général de PIGA.

1.1.1 PIGA, un système de détection d'intrusions

PIGA (Policy Interaction Graph Analysis) est un système de détection d'intrusions (IDS) qui a été développé par l'équipe Sécurité et Distribution des Systèmes du Laboratoire d'Informatique Fondamentale d'Orléans, en particulier durant la thèse de J. Briffaut.

Les types de systèmes de détection d'intrusions

Le but d'un système de détection d'intrusions est de détecter tout comportement intrusif effectué par un utilisateur ou un service, sur le système surveillé. Un système de détection d'intrusions fonctionne par analyse de données, les données dépendant du type du système de détection d'intrusions. Il existe deux types d'IDS : les systèmes de détection d'intrusions réseau (Network IDS ou NIDS) et les systèmes de détection d'intrusions système (Host IDS ou HIDS).

Les HIDS, tels que *Samhain* [WPRW05] ou *OSSEC* [Cid04], analysent les données fournies par le système. Ces données peuvent être, par exemple, des séquences d'appels système ou des logs, du système d'exploitation ou d'un service en particulier. PIGA est un système de détection d'intrusions système.

Les NIDS, tels que *Snort* [Roe99] ou *Bro* [Pax99], analysent les paquets transitant sur le réseau. Un des avantages des NIDS est qu'il est possible de protéger plusieurs systèmes appartenant au même réseau avec un seul HIDS.

Les méthodes de détection

Il existe deux principales méthodes de détection : la détection par comportement et la détection par signature.

Le principe de la détection par comportement est de détecter toute utilisation *anormale* d'un service ou d'une application [Co80, Den87]. Pour cela, il est d'abord nécessaire de définir le comportement *normal* des services et des applications présents sur le système. Cela passe par une phase d'apprentissage durant laquelle les services et applications sont utilisés normalement. Cette méthode de détection permet théoriquement de détecter toutes les attaques, qu'elles soient connues ou non. Cependant, elle est très dépendante de la qualité et de l'exhaustivité de la phase d'apprentissage. L'IDS *McPAD* [PAF+09] utilise ce type de méthode de détection.

Le but de la détection par signature est de détecter les attaques connues [HLR92]. Pour cela, cette méthode utilise une base de signatures d'attaques. La base permet alors de détecter toutes les attaques dont elle possède la signature. Cependant, cette méthode ne permet de détecter les attaques inconnues et demande une mise à jour fréquente de la base de signature. L'IDS *ASAX* [HLCMM92] utilise ce type de méthode.

PIGA utilise la détection paramétrée par une politique. Cette méthode se rapproche de la détection par comportement car elle va détecter les attaques grâce à leur comportement. Cependant, elle ne nécessite pas d'apprentissage. En effet, un comportement est considéré comme une attaque s'il viole la politique. La principale difficulté dans l'utilisation de cette méthode est la définition de la politique qui demande une grande connaissance du système sur lequel l'IDS est installé. L'IDS décrit dans [KR02] utilise cette méthode.

Critères d'efficacité

L'efficacité d'un système de détection d'intrusions peut être évalué grâce à deux indicateurs : le taux de *faux-positifs* et le taux de *faux-négatifs*. Un *faux-positif* correspond à une alerte levée alors qu'il n'y a pas eu d'intrusion, c'est-à-dire une fausse alerte. Un *faux-négatif* correspond à l'absence d'alerte levée alors qu'il y a eu une intrusion, c'est-à-dire une intrusion non détectée. Pour ces deux indicateurs, plus le taux est bas, plus le système de détection d'intrusions est efficace.

1.1.2 Propriété de sécurité

Nous considérons un système d'informations comme un automate à états finis. Afin de pouvoir détecter une intrusion, il est nécessaire de pouvoir reconnaître les comportements intrusifs. Un comportement est considéré comme intrusif s'il ne respecte pas les conditions établies dans la politique du système de détection d'intrusions. Ces conditions sont appelées propriétés de sécurité. 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 constituent la politique de sécurité du système. L'*intégrité*, la *confidentialité* et la

disponibilité sont les propriétés de sécurité les plus communes [Dep85].

Dans cette section, nous utiliserons les termes généraux *information*, *entité* et *ressource*. De façon générale, une information peut notamment correspondre à une donnée ou au contenu d'un fichier. Une entité peut, par exemple, faire référence à un utilisateur ou un processus et une ressource, à un service, un fichier ou un composant matériel.

Confidentialité

La *confidentialité* est la garantie qu'une entité ne peut pas accéder à une information sans en avoir l'autorisation. La *propriété de confidentialité* peut être définie comme suit :

Définition 1 (Confidentialité [Off91, Dep85, Bis03])
Soit I de l'information et soit X un ensemble d'entités non autorisées à accéder à I. La propriété de confidentialité de X envers I est respectée si aucun membre de X ne peut obtenir l'information I de manière entière ou partielle.

La propriété de confidentialité doit empêcher les entités non autorisées à accéder à l'information confidentielle, tout en permettant aux entités autorisées d'y accéder. On dit qu'une information est confidentielle pour une entité, si celle-ci ne peut pas y accéder.

Intégrité

L'*intégrité* d'une information est la garantie que celle-ci ne peut pas être modifiée ou détruite, volontairement ou non, par une entité non autorisée. L'*intégrité* d'une ressource désigne le fait que celle-ci « fonctionne » correctement. La *propriété d'intégrité* peut être définie comme suit :

Définition 2 (Intégrité [Off91, Dep85, Bis03])
Soit X un ensemble d'entités et soit I de l'information ou une ressource. Alors la propriété d'intégrité de X envers I est respectée si aucun membre de X ne peut modifier I.

La propriété d'intégrité permet d'empêcher toute modification d'une information ou d'une ressource par une entité non autorisée.

Disponibilité

La *disponibilité* est la garantie qu'un ensemble d'entités peut utiliser une information ou une ressource. La *propriété de disponibilité* peut être définie comme suit :

Définition 3 (Disponibilité [Off91, Dep85, Bis03])
Soit X un ensemble d'entités et soit I une ressource. Alors la propriété de disponibilité de X envers I est respectée si tous les membres de X ont accès à I.

Séparation des privilèges

Il existe également un ensemble de propriétés correspondant à des cas particuliers. Cet ensemble correspond aux propriétés d'abus de privilèges. Parmi celles-ci, nous trouvons la *séparation des privilèges*. La séparation des privilèges d'une entité sur un objet empêche l'entité d'exécuter l'objet si elle l'a modifié précédemment. La *propriété de séparation des privilèges* peut être définie comme suit :

Définition 4 (Séparation des privilèges)
Soit X un ensemble d'entités et soit O un objet exécutable. Alors la propriété de séparation des privilèges de X envers O est respectée si aucun membre de X ne peut exécuter O après l'avoir modifié.

Support des propriétés de PIGA

PIGA est capable de supporter les propriétés précédentes, plus d'autres propriétés telles que :

- La propriété d'absence de changement de contexte.
- La propriété d'exécutables de confiance.
- La propriété de respect des règles de contrôle d'accès.

Cependant, le principe de PIGA réside dans son extensibilité puisqu'il peut facilement supporter de nouvelles propriétés exprimables au moyen du langage PIGA-SPL.

1.1.3 Base de comportements malicieux

Afin d'identifier les comportements allant à l'encontre des propriétés de sécurité, il est nécessaire d'avoir un moyen de les représenter. L'objet représentant l'ensemble des comportements malicieux est la base de tous les comportements malicieux possibles.

Définition 5 (Base de comportements malicieux)
Une base de comportements malicieux *est un objet qui identifie l'ensemble des comportements illicites pour un système et une politique de sécurité donnés. Une base de comportements malicieux peut être vue comme une boîte noire qui, pour un comportement donné, indique si le comportement est licite ou non.*

La liste exhaustive de tous les comportements violant la politique de sécurité sur un système est l'exemple le plus simple de base de comportements malicieux. Le système de signatures de PIGA, générées à partir de plusieurs graphes représentant le système ainsi que d'une politique de sécurité, est également un exemple de base de comportements malicieux.

Cette base énumère exhaustivement toutes les activités système qui sont autorisées par la politique MAC et qui violent les propriétés définissant la politique PIGA. L'énumération est possible lorsque l'analyseur PIGA prend, en entrée, une politique MAC restreignant les états possibles du système. Sur cet ensemble restreint d'états, l'analyseur PIGA cherche les activités qui correspondent à la violation d'une propriété.

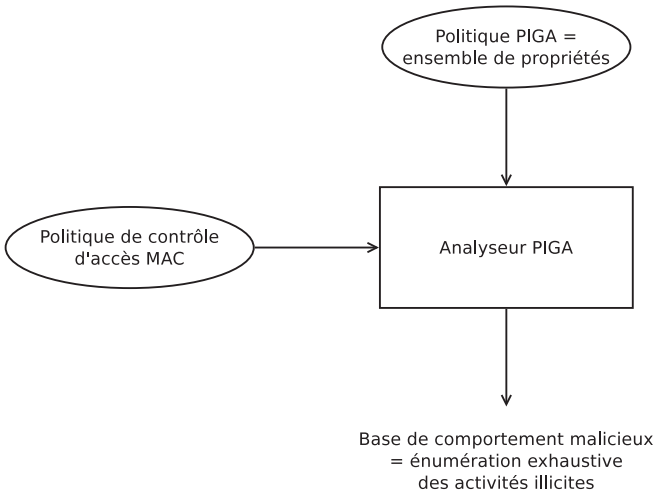


FIGURE 1.1 – Fonctionnement de la génération de la base de comportements malicieux de PIGA

Définition 6 (Équivalence des bases de comportements malicieux)
Deux bases de comportements malicieux sont équivalentes si, pour un même comportement donné, celui-ci est soit illicite pour les deux bases, soit licite.

1.1.4 Utilisation d'un système de contrôle mandataire

PIGA fonctionne comme une surcouche d'un système de contrôle d'accès mandataire (MAC). Un système de contrôle d'accès mandataire permet de définir une politique de d'accès qui ne peut pas être modifiée par les utilisateurs. Les droits définis par la politique sur un objet ne peuvent pas être modifiés par un utilisateur, même si celui-ci est propriétaire de cet objet. Ce système s'oppose au système de contrôle d'accès discrétionnaire (DAC), dans lequel chaque utilisateur définit les droits d'accès des objets qu'il possède. Une politique de système de contrôle d'accès mandataire définit les actions autorisées. Ainsi, tout ce qui n'est pas dans la politique est considéré comme interdit. L'avantage de PIGA est de permettre de discriminer plus précisément, pour l'ensemble des états autorisés par un système MAC, les comportements licites et ceux illicites. Durant sa thèse [Bri07], J. Briffaut a implémenté PIGA sur deux systèmes de contrôle d'accès mandataire : SELinux et GRSECURITY [AEKBT+05, BBLT06, BBC+06b, BBC+06a].

PIGA utilise la politique du système de contrôle d'accès mandataire pour récupérer l'ensemble des actions autorisées par le système. Ces actions sont appelées *interactions* et sont définies grâce à trois éléments : sc_{src} le contexte de sécurité source, sc_{cible} le contexte de sécurité cible et *eo* l'opération élémentaire.

Un contexte de sécurité est une étiquette regroupant des objets du système. Un contexte peut regrouper un ou plusieurs objets du système mais un objet n'appartient qu'à un seul contexte. Il existe deux ensembles disjoints de contexte de sécurité : les contextes *sujets* et les contextes *objets*. L'ensemble des contextes sujets représente les contextes pouvant effectuer des opérations élémentaires. Les contextes ne pou-

vant pas effectuer d'opération élémentaire appartiennent à l'ensemble des contextes objets. Une opération élémentaire peut être soit un appel système, soit une opération spécifique au système de contrôle mandataire. Elle peut être seule (e.g., `{write}`) ou être associée à une classe (e.g., `{file:write}`). Cet exemple signifie que l'autorisation d'écriture ne s'applique qu'aux fichiers.

Ainsi, on peut lire une *interaction* $it = (sc_{src}, sc_{cible}, eo)$ de cette manière : un objet du contexte sc_{src} est autorisé à effectuer l'opération élémentaire eo sur un objet appartenant au contexte sc_{cible}. Par exemple, l'interaction $it =$ (`admin_d`, `shadow_t`, `write`) signifie que les éléments du contexte `admin_t` ont l'autorisation d'écriture sur les éléments du contexte `shadow_t`. Les administrateurs peuvent donc modifier le fichier `shadow`.

À partir des interactions récupérées de la politique du système de contrôle d'accès mandataire, il est possible de générer des vecteurs d'interactions. Un vecteur d'interactions contient toutes les opérations élémentaires autorisées pour un contexte source et un contexte cible donnés.

Le vecteur d'interactions iv du contexte sc_i sur sc_j contient toutes les opérations élémentaires autorisées de sc_i sur sc_j. Ces opérations sont représentées par un ensemble EOS tel que, si $eo \in EOS$, alors l'interaction $it = (sc_i, sc_j, eo)$ est autorisée par la politique du système MAC. Ce vecteur d'interactions est noté $iv = (sc_i, sc_j, EOS)$.

1.1.5 Génération de signatures et détection des comportements illicites

Grâce aux vecteurs d'interactions, PIGA génère un graphe nommé graphe d'interactions qui représente l'ensemble des interactions autorisées. Il est ensuite possible de dériver de ce graphe d'autres graphes, tels que le graphe de flux d'informations ou le graphe de transition. Nous détaillons ces différents graphes ainsi que leur génération dans la section 1.2.

À partir des différents graphes, PIGA génère les signatures correspondant à l'ensemble des comportements permettant de violer les propriétés de sécurité décrites dans sa politique. Suivant le type des propriétés de sécurité, les graphes utilisés dans la génération des signatures correspondantes sont différents. Le mécanisme de génération de signatures de PIGA est détaillé dans la section 1.3

La base de signatures obtenue permet à PIGA de détecter les comportements violant une ou plusieurs propriétés. Le mécanisme de détection de PIGA est détaillé dans la section 1.4

La génération des graphes et le calcul des signatures sont effectués une seule fois en mode hors-ligne avant que le système ne soit utilisé. Le système de détection est en ligne. Durant son utilisation, la base de signatures est chargée en mémoire afin de détecter, en direct, les comportements violant la politique de sécurité.

La figure 1.2 schématise la génération de signatures liées aux propriétés de confidentialité.

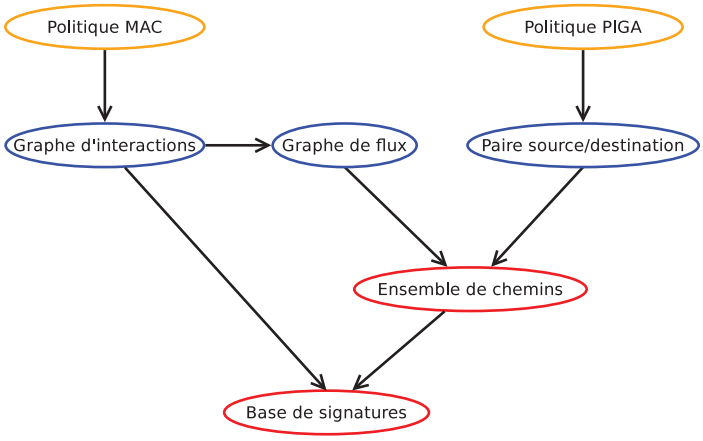


FIGURE 1.2 – Fonctionnement de la génération de signatures

1.2 Graphe d'interactions et graphes dérivés

1.2.1 Graphe d'interactions

PIGA modélise l'ensemble des autorisations, définies par la politique du système de contrôle d'accès mandataire, sous forme d'un graphe nommé graphe d'interactions. Ce graphe orienté est construit à partir de l'ensemble des vecteurs d'interactions issus de la politique du système MAC. On peut donc définir le graphe d'interactions $G = (V, A)$, tel que :

- V représente l'ensemble des contextes de sécurité impliqués dans une autorisation, et
- A l'ensemble des autorisations entre ces contextes.

De plus, chaque arc de ce graphe est étiqueté. L'étiquette de l'arc entre le contexte de sécurité sc_1 et le contexte de sécurité sc_2 représente l'ensemble des opérations élémentaires de sc_1 sur sc_2 autorisées par la politique MAC du système.

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

L'exemple ci-dessus représente les opérations élémentaires possibles du contexte `user_u:user_r:xserver_t` sur `system_u:object_r:mtrr_device_t`. C'est-à-dire que n'importe quel élément, ayant `user_u:user_r:xserver_t` comme contexte de sécurité, peut effectuer des opérations de type lecture ou écriture sur les fichiers du contexte `system_u:object_r:mtrr_device_t`. SELinux fournit un grand nombre d'opérations pouvant être autorisées entre différents contextes. La figure 1.3 présente un graphe d'interactions généré à partir de la table 1.1 présentant un ensemble simplifié de vecteurs d'interactions. Ce graphe représente les différents mécanismes de connexion ainsi que les moyens d'accès possibles pour les utilisateurs aux commandes Linux (contexte `bin`).

L'algorithme 1 est utilisé pour construire le graphe d'interactions à partir des vecteurs d'interactions produits par la politique du système de contrôle d'accès mandataire.

iv_1	(passwd_d,shadow_t,{read})
iv_2	(passwd_d,login_d,{transition,signal})
iv_3	(passwd_d,ssh_d,{transition,signal})
iv_4	(login_d,admin_d,{transition,write})
iv_5	(login_d,user_d,{transition,write})
iv_6	(ssh_d,admin_d,{transition,write})
iv_7	(ssh_d,user_d,{transition,write})
iv_8	(user_d,admin_d,{transition,write})
iv_9	(admin_d,bin_t,{read,write,execute})
iv_{10}	(user_d,bin_t,{read,execute})

TABLE 1.1 – Exemple simplifié d'ensemble de vecteurs d'interactions

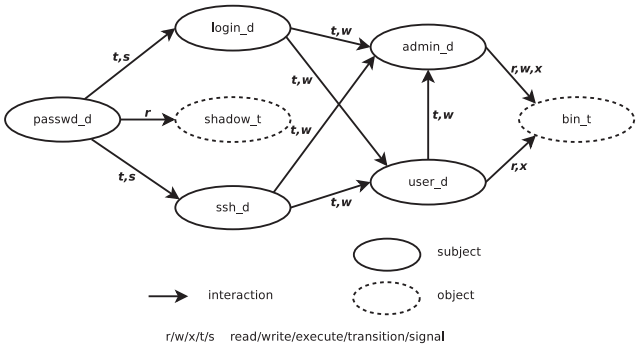


FIGURE 1.3 – Le graphe d'interactions généré à partir de la table 1.1

1.2.2 Graphes dérivés

Table de changement d'état

À partir du graphe d'interactions, nous pouvons générer d'autres graphes utilisés pour la génération de signatures. Ces nouveaux graphes sont obtenus en filtrant le graphe d'interactions sur l'étiquetage. Ces opérations de filtrage peuvent avoir des conséquences différentes sur les arcs : un arc peut être supprimé du graphe si son étiquette, une fois filtrée, est vide ; un arc peut aussi changer d'orientation. Le filtrage est effectué à partir de tables de changement d'état qui diffèrent suivant le graphe que l'on veut obtenir. Ces tables de changement d'états associent à une opération

Algorithme 1 Création du graphe d'interactions

Entrée: IVS, l'ensemble des vecteur d'interactions de la politique MAC
Sortie: Le graphe d'interactions $G = (V, A)$
  pour tout vecteur d'interactions $iv = (sc_{src}, sc_{cible}, EOS) \in IVS$ **faire**
    $V \leftarrow V \cup \{sc_{src}\} \cup \{sc_{cible}\}$
    $e \leftarrow (sc_{src}, sc_{cible})$
    $A \leftarrow A \cup \{e\}$
    ajoutEtiquette(e,EOS)
  retourner $G = (V, A)$

eo_1	$\rightarrowtail$
eo_2	$\rightarrowtail$
eo_3	$\leftarrowtail$
eo_4	$\leftarrowtail$

TABLE 1.2 – Exemple de table de changement d'états

`blk_file` : read	$\leftarrowtail$
`chr_file` : read	$\leftarrowtail$
`dir:read`	$\leftarrowtail$
`fifo_file:read`	$\leftarrowtail$
`file:read`	$\leftarrowtail$
`lnk_file:read`	$\leftarrowtail$
`blk_file:write`	$\rightarrowtail$
`chr_file:write`	$\rightarrowtail$
`dir:write`	$\rightarrowtail$
`fifo_file:write`	$\rightarrowtail$
`file:write`	$\rightarrowtail$
`lnk_file:write`	$\rightarrowtail$

TABLE 1.3 – Table de changement d'états de flux avec un mapping de 35

élémentaire un sens d'arc. Si une opération n'apparaît pas dans la table, alors aucun arc ne la représentera.

Par exemple, on considère un ensemble de 5 opérations élémentaires $\{eo_1, ..., eo_5\}$, et un graphe d'interactions donné. Pour générer un graphe dérivé à partir de la table 1.2 et d'un graphe d'interactions, il faut ajouter un arc $sc_1 \rightarrow sc_2$ au graphe dérivé pour chaque arc $sc_1 \xrightarrow{eo_1} sc_2$ ou $sc_1 \xrightarrow{eo_2} sc_2$ du graphe d'interactions. Les arcs $sc_1 \xrightarrow{eo_5} sc_2$ n'entraînent pas l'ajout d'arc sur le graphe dérivé, car l'opération eo_5 n'est pas présente dans la table.

La table de changement d'états est déterminée en fonction d'un fichier de mapping ainsi que du niveau de sécurité souhaité. Le fichier de mapping associe, à chaque interaction, un type (w, r, b ou n) ainsi qu'un niveau de criticité. Plus ce niveau est élevé, plus l'interaction est critique. Le fichier de mapping est tiré de l'outil d'analyse de politique SELinux *apol* appartenant à la suite d'outils *SETools*. Ce fichier a été modifié par J. Briffaut afin de l'adapter à PIGA. La table 1.3 (resp. 1.4) montre la table de changement d'états correspondant à un niveau de sécurité de 35 (resp. 42).

`dir:read`	$\leftarrowtail$
`file:read`	$\leftarrowtail$
`dir:write`	$\rightarrowtail$
`file:write`	$\rightarrowtail$

TABLE 1.4 – Table de changement d'états de flux avec un mapping de 42

Graphe de flux

Dans le cas de la propriété de confidentialité, la génération de signatures utilise le *graphe de flux* $FG = (V, A)$ où V est l'ensemble des contextes de sécurité et A représente tous les transferts d'informations possibles entre deux contextes de sécurité. Le graphe de flux est généré à partir du graphe d'interactions, en supprimant les étiquettes n'impliquant pas de transfert d'informations. Les arcs restants représentent des interactions de type `read` ou `write`. Si une interaction de type `write` existe de sc_1 sur sc_2, alors il y a un transfert d'informations de sc_1 vers sc_2 et un arc de sc_1 vers sc_2 apparaîtra sur le graphe de flux. Si une interaction de type `read` existe de sc_1 sur sc_2, alors il y a un transfert d'informations de sc_2 vers sc_1 et un arc de sc_2 vers sc_1 apparaîtra sur le graphe de flux. La figure 1.4 représente le graphe de flux généré à partir du graphe d'interactions de la figure 1.3 et de la table de changement d'états de flux 1.5. Lors des expérimentations sur les systèmes réels, nous avons utilisé les tables 1.3 et 1.4.

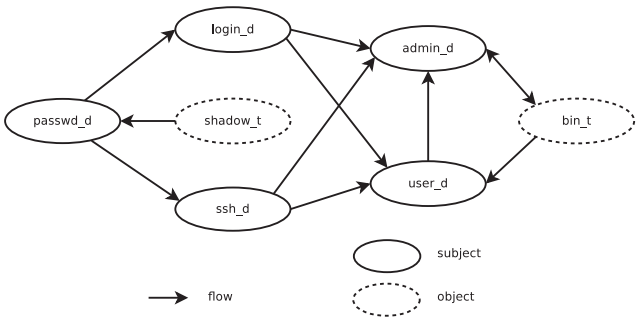


FIGURE 1.4 – Le graphe de flux associé au graphe d'interactions de la figure 1.3

`read`	$\longleftarrow$
`write`	$\longrightarrow$
`signal`	$\longrightarrow$
`transition`	$\longrightarrow$

TABLE 1.5 – Table de changement d'états de flux utilisée pour générer le graphe représenté par la figure 1.4

Graphe de transition

Le *graphe de transition* permet de représenter les transitions de contextes autorisées. Il est composé uniquement des contextes sujets du graphe d'interactions car une transition ne peut se faire qu'entre deux contextes sujets. De plus, ses arcs sont différents de ceux du graphe d'interactions. Pour chaque vecteur d'interactions $iv = (sc_{src}, sc_{cible}, EOS)$, le graphe de transition possède un arc $a = (sc_{src}, sc_{dest})$ étiqueté par les opérations élémentaires de type transition. La table de changement d'état de transition indique donc si une opération élémentaire est de type transition. Si tel est le cas, l'opération élémentaire est associée à $\longleftarrow$. La figure 1.5 représente le graphe de transition correspondant au graphe d'interactions de la figure 1.3.

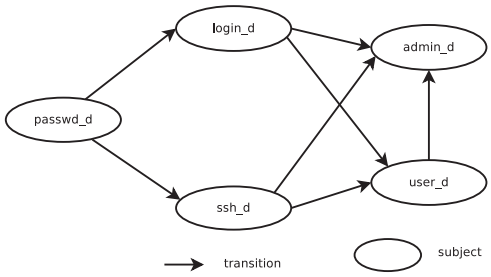


FIGURE 1.5 – Le graphe de transition associé au graphe d'interactions de la figure 1.3

1.3 Mécanisme de génération de signatures

1.3.1 Signatures pour PIGA

En utilisant les différents graphes qui ont été définis dans la section précédente, l'évolution réelle du système peut être vue comme la construction dynamique d'un graphe représentant les interactions déjà effectuées sur le système. Des informations temporelles peuvent y être ajoutées. Naturellement, ce graphe est un graphe partiel du graphe d'interactions défini par la politique MAC. Chaque opération élémentaire que l'on peut effectuer sur un système d'exploitation (commande système) génère des opérations système de plus ou moins bas niveau qui passeront par le filtre la politique MAC.

À l'aide de la modélisation avec le graphe de flux, il est facile de suivre comment une information transite dans le système et, en particulier, si elle atteint une zone interdite si on considère une propriété de confidentialité.

PIGA utilise, ainsi, des signatures afin d'identifier les comportements malicieux. Une signature est un objet permettant de représenter un ou plusieurs comportements allant à l'encontre d'une propriété de sécurité. Ainsi, une signature est associée à une propriété de sécurité, tandis qu'une propriété de sécurité engendre plusieurs signatures. Une signature est composée d'une suite d'actions ordonnée formant un comportement malicieux. Une signature représente un ensemble d'interactions ordonné.

Théorème 1
La base de signatures de PIGA est équivalente à la liste exhaustive des comportements autorisés par le système MAC et violant les propriétés de sécurité de la politique PIGA.

Ce théorème a été prouvé par J. Briffaut dans sa thèse [Bri07].

Les signatures sont calculées à partir des différents graphes générés par PIGA. Suivant les propriétés de sécurité, les graphes utilisés sont différents. Les signatures sont calculées pour chaque propriété de la politique de sécurité et forment ainsi la base de signatures.

Afin de décrire la forme des comportements, nous utiliserons les notations suivantes :

- $sc_1 \xrightarrow{eo} sc_2$: une interaction du contexte sc_1 sur le contexte sc_2 effectuant l'opération élémentaire eo.
- $sc_1 \Rightarrow sc_2$: une séquence d'interactions quelconque du contexte sc_1 vers le contexte sc_2
- $sc_1 \xrightarrow{trans} sc_2$: une transition du contexte sc_1 vers le contexte sc_2
- $sc_1 \Rightarrow_{trans} sc_2$: une séquence de transitions du contexte sc_1 vers le contexte sc_2
- $sc_1 > sc_2$: un transfert d'informations de sc_1 vers sc_2
- $sc_1 \gg sc_2$: un flux d'informations (une suite de transferts causalement dépendants) de sc_1 vers sc_2

De plus, certains types de signatures complexes sont composés de plusieurs comportements illicites ordonnés dans le temps. Pour les représenter, nous utiliserons la composition ($\circ$). Ainsi $sc_3 \xrightarrow{eo_2} sc_4 \circ sc_1 \xrightarrow{eo_1} sc_2$ signifie que la signature est composée de l'interaction $sc_1 \xrightarrow{eo_1} sc_2$ suivie de l'interaction $sc_3 \xrightarrow{eo_2} sc_4$.

1.3.2 Cas particulier : la propriété de confidentialité

Durant cette thèse, nous nous sommes concentrés sur la confidentialité. Les expérimentations de [BRCTZ09] montrent que cette propriété génère le plus grand nombre de signatures. Dans les expérimentations du chapitre 3, nous verrons qu'une seule propriété de confidentialité peut engendrer plus de 85000 signatures.

La propriété de confidentialité d'un contexte sc_1 pour un contexte sc_2 est violée par un comportement entraînant soit un transfert d'informations direct, soit un flux d'informations de sc_1 vers sc_2. Une signature associée à une propriété de confidentialité peut donc avoir deux formes :

- $sc_1 > sc_2$ correspond un transfert d'informations de sc_1 vers sc_2
- $sc_1 \gg sc_2$ correspond un flux d'informations de sc_1 vers sc_2

Une signature est donc une séquence représentant une succession de transferts d'informations possibles. Pour deux contextes de sécurité sc_1 et sc_2, $sign(sc_1, sc_2)$ représente l'ensemble des signatures entre sc_1 et sc_2. $sign(sc_1, sc_2)$ peut être calculé comme l'ensemble des chemins simples entre sc_1 et sc_2 sur le graphe de flux. Cette propriété a été prouvée dans la thèse de J.Briffaut [Bri07].

$$sign(sc_1, sc_2) = \left\{ \begin{array}{l} \text{chemins simples entre } sc_1 \\ \text{et } sc_2 \text{ dans le graphe de flux} \end{array} \right\} \tag{1.1}$$

Par exemple, pour un système correspondant à la figure 1.3, l'administrateur souhaite s'assurer que tous les fichiers appartenant au contexte de sécurité `shadow_t` ne soient pas accessibles aux utilisateurs (appartenant au contexte `user_d`). En considérant le graphe de flux de la figure 1.4, il est possible que de l'information aille de `shadow_t` à `user_d` en seulement 3 étapes sans violer la politique SELinux. L'analyse du graphe de flux révèle 4 chemins simples violant 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

Tous les comportements ci-dessus sont de la forme shadow_t ≫ user_d et se décomposent en plusieurs transferts d'informations. Les comportements correspondants sont donc :

– 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

Une fois les chemins obtenus, il est nécessaire, pour chacun d'eux, de connaître les interactions qui engendrent les transferts d'informations qui les composent. Pour cela, on utilise la table de changement d'état de flux, utilisée précédemment, pour générer le graphe de flux.

Ici, nous considérerons la table 1.5. Un transfert d'informations $sc_1 > sc_2$ peut donc correspondre aux interaction $sc_1 \xrightarrow{write} sc_2$, $sc_1 \xrightarrow{trans} sc_2$ ou $sc_2 \xrightarrow{read} sc_1$. La table 1.6 associe les transferts d'informations présents dans les comportements ci-dessus avec les interactions les engendrant.

shadow_t > passwd_d	passwd_d $\xrightarrow{read}$ shadow_t	
passwd_d > ssh_d	passwd_d $\xrightarrow{trans}$ ssh_d	
passwd_d > login_d	passwd_d $\xrightarrow{trans}$ login_d	
ssh_d > admin_d	ssh_d $\xrightarrow{write}$ admin_d,	ssh_d $\xrightarrow{trans}$ admin_d
login_d > admin_d	login_d $\xrightarrow{write}$ admin_d,	login_d $\xrightarrow{trans}$ admin_d
admin_d > bin_t	admin_d $\xrightarrow{write}$ bin_t	
ssh_d > user_d	ssh_d $\xrightarrow{write}$ user_d,	ssh_d $\xrightarrow{trans}$ user_d
login_d > user_d	login_d $\xrightarrow{write}$ user_d,	login_d $\xrightarrow{trans}$ user_d
bin_t > user_d	user_d $\xrightarrow{read}$ bin_t	

TABLE 1.6 – Table de correspondance des transferts d'informations et des interactions

Les signatures obtenues peuvent être générées en format XML afin d'être lisibles. Voici la signature XML correspondant au comportement :

shadow_t > passwd_d > ssh_d > admin_d > bin_t > user_d.

```
<sequence nom="inf$3" type="conf">
 <vector cpt="0" scsource="passwd_d" sctarget="shadow_t" IV="read;" />
 <vector cpt="1" scsource="passwd_d" sctarget="ssh_d" IV="transition;" />
 <vector cpt="2" scsource="ssh_d" sctarget="admin_d" IV="write;transition;" />
 <vector cpt="3" scsource="admin_d" sctarget="bin_t" IV="write;" />
 <vector cpt="4" scsource="user_d" sctarget="bin_t" IV="read;" />
</sequence>
```

La signature XML possède un nom et se compose de plusieurs lignes. Le nom correspond au type de signature et à son numéro. Ici, la signature est la troisième de type flux d'informations (i.e., associée à une propriété de confidentialité), son nom est donc inf$3. Ensuite, chaque ligne est composée de plusieurs éléments :

- `cpt` : le numéro du vecteur d'interactions dans la signature ;
- `scsource` : le contexte de sécurité source du vecteur d'interactions ;
- `sctarget` : le contexte de sécurité cible du vecteur d'interactions ;
- `IV` : les opérations élémentaires du vecteur d'interactions.

1.3.3 Autres propriétés de sécurité

Intégrité des objets

La propriété d'intégrité des objets d'un contexte sc_1 envers un contexte sc_2 peut être violée de deux façons, soit par une écriture directe de sc_1 sur sc_2, soit par flux partant de sc_1 et se terminant sur sc_2. Par exemple, un accès à un contexte s'effectue grâce à une séquence de transitions. Ainsi, le contexte sc_1 a accès au contexte sc_i s'il existe une séquence de transitions de sc_1 à sc_i. On obtient au moins deux formes possibles pour les signatures associées à une propriété d'intégrité des objets de sc_1 envers sc_2 :

- $sc_1 \xrightarrow{write} sc_2$ correspond à l'écriture directe de sc_1 sur sc_2 ;
- $sc_1 \Rightarrow_{trans} sc_i \xrightarrow{write} sc_2$ correspond à l'accès de sc_1 à sc_i puis à l'écriture directe de sc_i sur sc_2.

Ainsi, le calcul des signatures pour la propriété d'intégrité des objets nécessite le graphe d'interactions et le graphe de transition. Les signatures de la forme $sc_1 \xrightarrow{write} sc_2$ sont générées uniquement à partir du graphe d'interactions. Pour obtenir ces signatures, il suffit, pour chaque couple (sc_1,sc_2), d'analyser l'arc de sc_1 vers sc_2 puis de récupérer toutes les opérations élémentaires de cet arc impliquant une écriture. Ainsi, toutes ces signatures en format XML sont de la forme :

```
<sequence nom="xxx" type="int">
 <vector cpt="0" scsource="sc_1" sctarget="sc_2" IV="write_like_eo;" />
</sequence>
```

Les signatures de la forme $sc_1 \Rightarrow_{trans} sc_i \xrightarrow{write} sc_2$ sont générées à partir du graphe de transition, pour la partie $sc_1 \Rightarrow_{trans} sc_i$ et à partir du graphe d'interactions, pour la partie $sc_i \xrightarrow{write} sc_2$. La construction se fait donc en deux étapes.

Premièrement, le graphe d'interactions est utilisé afin d'obtenir une liste de tous les voisins du contexte cible sc_2 qui peuvent le modifier. Pour cela, pour chaque voisins sc_v de sc_2, on ajoute ce voisin à la liste si une des opérations élémentaires étiquetant l'arc (sc_v,sc_2) implique une écriture.

Deuxièmement, le graphe de transition est utilisé pour obtenir toutes les séquences de transitions allant du contexte sc_1 aux contextes de la liste générée précédemment. Ces séquences de transitions sont des chemins dans le graphe de transition.

À partir de ces séquences de transitions, les signatures sont construites en ajoutant, à la fin de chacune de ces séquences, l'interaction de type écriture du dernier contexte de cette séquence sur le contexte destination. Ainsi, pour chaque contexte intermédiaire sc_i, on obtiendra autant de signatures qu'il existe de séquences de transitions entre sc_1 et sc_i. La table 1.7 montre la construction des signatures à partir des séquences de transitions.

Séquences de transitions	Interaction de type écriture	Signatures obtenues
$sc_1 \Rightarrow_{trans1} sc_i$	$sc_i \xrightarrow{write} sc_2$	$sc_1 \Rightarrow_{trans1} sc_i \xrightarrow{write} sc_2$
$sc_1 \Rightarrow_{trans2} sc_i$		$sc_1 \Rightarrow_{trans2} sc_i \xrightarrow{write} sc_2$
$sc_1 \Rightarrow_{trans3} sc_i$		$sc_1 \Rightarrow_{trans3} sc_i \xrightarrow{write} sc_2$

TABLE 1.7 – Construction des signatures à partir des séquences de transitions

Les signatures de la forme $sc_1 \Rightarrow_{trans} sc_i \xrightarrow{write} sc_2$ en format XML seront de la forme :

```
<sequence nom="xxx" type="int">
 <vector cpt="0" scsource="sc_1" sctarget="sc_j1" IV="transition;" />
 <vector cpt="1" scsource="sc_j1" sctarget="sc_j2" IV="transition;" />
 ...
 <vector cpt="x" scsource="sc_jn" sctarget="sc_i" IV="transition;" />
 <vector cpt="x+1" scsource="sc_i" sctarget="sc_2" IV="write_like_eo;" />
</sequence>
```

Séparation des privilèges

La propriété de séparation des privilèges d'un contexte sc_1 envers un contexte sc_2 est violée par deux comportements ordonnés dans le temps. Dans un premier temps, une modification du contexte sc_2 par le contexte sc_1 que ce soit directement ou par accès à un contexte intermédiaire sc_i. Dans un second temps, une exécution du contexte sc_2 par le contexte sc_1 que ce soit directement ou par accès à un contexte intermédiaire sc_j. On obtient notamment quatre formes possibles pour les signatures associées à une propriété de séparation des privilèges de sc_1 envers sc_2. Ces formes sont des compositions de la modification du contexte sc_2 par sc_1 et de son exécution :

- $sc_1 \xrightarrow{execute} sc_2$ ∘ $sc_1 \xrightarrow{write} sc_2$
- $sc_1 \xrightarrow{execute} sc_2$ ∘ $sc_1 \Rightarrow_{trans} sc_i \xrightarrow{write} sc_2$
- $sc_1 \Rightarrow_{trans} sc_j \xrightarrow{execute} sc_2$ ∘ $sc_1 \xrightarrow{write} sc_2$
- $sc_1 \Rightarrow_{trans} sc_j \xrightarrow{execute} sc_2$ ∘ $sc_1 \Rightarrow_{trans} sc_i \xrightarrow{write} sc_2$

Le calcul des signatures pour les propriétés de séparation des privilèges nécessite donc les graphes de transition et d'interactions. De la même manière que pour les propriétés d'intégrité des objets, les exécutions directes ($sc_1 \xrightarrow{execute} sc_2$) et les écritures directes ($sc_1 \xrightarrow{write} sc_2$) utilisent uniquement le graphe d'interactions.

Les exécutions et écritures par accès à un contexte intermédiaire ($sc_1 \Rightarrow_{trans} sc_j \xrightarrow{execute} sc_2$ et $sc_1 \Rightarrow_{trans} sc_i \xrightarrow{write} sc_2$) utilisent le graphe de transition pour la partie $sc_1 \Rightarrow_{trans} sc_j$ et le graphe d'interactions pour les parties $sc_1 \xrightarrow{execute} sc_2$ et $sc_1 \xrightarrow{write} sc_2$.

Les deux composantes des signatures associées à une propriété de séparation des privilèges peuvent donc être composées d'une à deux parties suivant que l'écriture et l'exécution s'effectuent de manière directe ou par accès à un contexte intermédiaire.

La composante liée à la modification du contexte cible se construit de la même manière que les signatures associées à une propriété d'intégrité des objets. Cette composante a également la même forme que les signatures correspondant à une propriété d'intégrité des objets :

- Si la composante est de la forme $sc_1 \xrightarrow{write} sc_2$, elle sera, sous format XML, de la forme :

```
<sequence nom="xxx" type="duties_sep_w">
 <vector cpt="0" scsource="sc_1" sctarget="sc_2" IV="write_like_eo;" />
</sequence>
```

- Si la composante est de la forme $sc_1 \Rightarrow_{trans} sc_i \xrightarrow{write} sc_2$, elle sera, sous format XML, de la forme :

```
<sequence nom="xxx" type="duties_sep_w">
 <vector cpt="0" scsource="sc_1" sctarget="sc_j1" IV="transition;" />
 ...
 <vector cpt="x" scsource="sc_jn" sctarget="sc_i" IV="transition;" />
 <vector cpt="x+1" scsource="sc_i" sctarget="sc_2" IV="write_like_eo;" />
</sequence>
```

La composante liée à l'exécution du contexte cible se construit de manière similaire à la composante précédente. En effet, si la composante est de la forme $sc_1 \xrightarrow{execute} sc_2$, elle est générée à partir du graphe d'interactions. Pour obtenir ces composantes, il suffit, pour chaque couple (sc_1, sc_2), d'analyser l'arc de sc_1 vers sc_2 et de ne conserver que ceux impliquant une exécution. Ces composantes sont sous la forme suivante en format XML :

```
<sequence nom="xxx" type="duties_sep_x">
 <vector cpt="0" scsource="sc_1" sctarget="sc_2" IV="execute;" />
</sequence>
```

Si la composante est de la forme $sc_1 \Rightarrow_{trans} sc_i \xrightarrow{execute} sc_2$, elle est générée à partir du graphe de transition pour la partie $sc_1 \Rightarrow_{trans} sc_i$ et à partir du graphe d'interactions pour la partie $sc_i \xrightarrow{execute} sc_2$. La construction se fait donc en deux étapes de la même manière que la composante liée à la modification. Ces composantes sont donc sous la forme suivante, en format XML :

```
<sequence nom="xxx" type="duties_sep_x">
 <vector cpt="0" scsource="sc_1" sctarget="sc_j1" IV="transition;" />
 ...
 <vector cpt="x" scsource="sc_jn" sctarget="sc_i" IV="transition;" />
 <vector cpt="x+1" scsource="sc_i" sctarget="sc_2" IV="execute;" />
</sequence>
```

Ainsi, pour chaque propriété de séparation des privilèges, on obtient un ensemble de signatures pour chacune des deux composantes de la propriété.

1.4 Mécanisme de détection

1.4.1 Fonctionnement général

Afin de détecter la complétion des signatures, le mécanisme de détection utilise des traces d'interactions. Ces traces sont générées à partir de l'audit du noyau. Chaque fois qu'une interaction (un appel système) est réalisée au niveau du noyau, SELinux génère une trace contenant plusieurs informations et notamment :

- la description de l'interaction, c'est-à-dire le contexte source et destination ainsi que l'opération élémentaire effectuée ;
- la date à laquelle l'interaction a été effectuée.

Le but du mécanisme de détection est de lancer des alertes chaque fois qu'une signature est reconnue, c'est-à-dire lorsque toutes les interactions qui la composent ont été effectuées dans l'ordre défini dans la signature. Pour chaque signature, on définit une interaction *pendante* qui correspond à la prochaine interaction qui fera progresser la signature. Dès que l'interaction pendante est effectuée, l'interaction suivante devient l'interaction pendante, la signature progresse. Lorsque la dernière interaction d'une signature est effectuée alors que celle-ci est pendante, une alerte est levée. En effet, aucune alerte n'est levée dans le cas où la dernière interaction d'une signature est effectué alors qu'elle n'est pas pendante, car cela signifie que toutes les interactions précédentes n'ont pas été effectuées ou qu'elles n'ont pas été effectuées dans l'ordre défini par la signature.

Le mécanisme de détection de PIGA peut être utilisé dans deux modes différents. Le premier mode est le live qui permet de détecter en direct si une propriété de sécurité est violée par un utilisateur. Si PIGA est configuré en tant qu'IDS, une alerte sera alors levée. Si PIGA est configuré en tant qu'IPS, l'action violant la propriété sera bloquée. Le second est le rejeu des traces. Ces traces représentent généralement la suite d'actions effectuées lors d'une session d'un utilisateur. Durant cette thèse, nous avons utilisé uniquement le mode rejeu de trace pour tester les performances du nouveau système de détection. Cela permet, en particulier, de comparer l'efficacité de deux systèmes sur exactement les mêmes actions.

Voici les principaux éléments composant une ligne d'un fichier de trace :

(1)	`Mar 25 14:58:15`
(2)	`read`
(3)	`pid=2323 ppid=2322`
(4)	`dpid=2442 dppid=2434`
(5)	`scontext=passwd_d`
(6)	`tcontext=shadow_t`
(7)	`tclass=file`

Les 7 éléments correspondent à :

- (1) La date et l'heure à laquelle l'interaction a été effectuée.
- (2) L'opération élémentaire effectuée lors de l'interaction.
- (3) Le numéro du processus (PID) ainsi que celui du processus parent (PPID) auquel appartient le contexte source.

– (4) Le numéro du processus (PID) ainsi que celui du processus parent (PPID) auquel appartient le contexte destination.
– (5) Le contexte source de l'interaction.
– (6) Le contexte cible de l'interaction.
– (7) La classe de l'opération élémentaire effectuée lors de l'interaction.

Ainsi, cette ligne de trace correspond à l'interaction `passwd_d` $\xrightarrow{file:read}$ `shadow_t` effectuée le 25 mars à 14h58 et 15 secondes par le processus n° 2323 qui a pour processus parent le processus n° 2322.

1.4.2 Monitoring et progression des signatures

Avant de pouvoir analyser les traces afin de faire progresser les signatures, PIGA crée une table de correspondance associant, à chaque couple source/cible possible, les signatures comportant une interaction associée à ce couple. Le processus de détection est différent suivant que les signatures sont simples ou sont composées. Les signatures liées à une propriété de confidentialité ou d'intégrité sont simples, tandis que celles liées à une propriété de séparation des privilèges sont composées.

Dans cette section, nous illustrons, au travers d'explications simples, les méthodes générales décrites dans la thèse de J. Briffaut [Bri07].

Cas des signatures simples

Tout au long de cette section, nous utilisons les signatures, ainsi que la trace suivante pour illustrer le fonctionnement du mécanisme de détection pour les signatures simples. Afin de simplifier la lecture de la trace, nous gardons uniquement l'opération élémentaire, ainsi que les contextes source et destination.

Signatures :

```
<sequence nom="inf$1" type="conf">
 <vector cpt="0" scsource="passwd_d" sctarget="shadow_t" IV="read;"/>
 <vector cpt="1" scsource="passwd_d" sctarget="ssh_d" IV="write;transition; "/>
 <vector cpt="2" scsource="ssh_d" sctarget="user_d" IV="write;transition; "/>
</sequence>
<sequence nom="inf$2" type="conf">
 <sequence cpt="0" scsource="passwd_d" sctarget="shadow_t" IV="read;"/>
 <sequence cpt="1" scsource="passwd_d" sctarget="login_d" IV="transition;"/>
 <sequence cpt="2" scsource="login_d" sctarget="admin_d" IV="write;transition;"/>
 <sequence cpt="3" scsource="admin_d" sctarget="bin_t" IV="write;"/>
 <sequence cpt="4" scsource="user_d" sctarget="bin_t" IV="read;"/>
</sequence>
```

Trace :

```
1: Jan 13 10:12:06 read scontext=passwd_d tcontext=shadow_t
2: Jan 13 10:12:52 read scontext=user_d tcontext=bin_t
3: Jan 13 10:13:03 signal scontext=passwd_d tcontext=ssh_d
4: Jan 13 10:13:10 transition scontext=passwd_d tcontext=ssh_d
5: Jan 13 10:13:21 transition scontext=passwd_d tcontext=login_d
```

(passwd_d, shadow_t)	inf$1,inf$2
(passwd_d, ssh_d)	inf$1
(ssh_d, user_d)	inf$1
(passwd_d, login_d)	inf$2
(login_d, admin_d)	inf$2
(admin_d, bin_t)	inf$2
(user_d, bin_t)	inf$2

TABLE 1.8 – Table de correspondance des signatures simples

```
6: Jan 13 10:13:28 read scontext=ssh_d tcontext=user_d
7: Jan 13 10:13:40 write scontext=ssh_d tcontext=admin_d
8: Jan 13 10:14:00 read scontext=ssh_d tcontext=user_d
```

À partir des signatures ci-dessus, on obtient la table de correspondance 1.8.

Avant de débuter l'analyse des traces, la première interaction de chaque signature simple est définie comme *pendante*. Les signatures sont dans l'état suivant (les interactions encadrées sont pendantes) :

shadow_t → passwd_d → ssh_d → user_d

shadow_t → passwd_d → login_d → admin_d → bin_t → user_d

Pour chaque ligne de trace, on récupère la paire de contextes source/destination impliqués. On utilise la table de correspondance pour obtenir les signatures qui contiennent une interaction impliquant cette paire. Ensuite, pour chaque signature retenue, on vérifie si l'opération élémentaire de la ligne de trace en cours correspond à une des opérations possibles de l'interaction contenue dans la signature. S'il n'y a pas correspondance, la signature ne progresse pas. Sinon, si l'interaction pendante n'est pas la dernière de la signature, celle-ci progresse. Dans le cas contraire, une alerte est levée et l'interaction pendante le reste. L'alerte contient le numéro de la signature complétée, la signature dans son intégralité, l'interaction ayant complété la signature, ainsi que la date et l'heure où celle-ci a été effectuée.

Pour notre exemple, la première ligne correspond à une interaction du contexte `passwd_d` sur le contexte `shadow_t`. D'après la table de correspondance 1.8, les signatures `inf$1` et `inf$2` comportent une interaction impliquant cette paire de contextes. Cette interaction est pendante sur ces signatures et possède l'opération `read` dans les opérations élémentaires possibles. Les deux signatures progressent, l'interaction `passwd_d` → `ssh_d` devient l'interaction pendante pour `inf$1` et `passwd_d` → `login_d` devient l'interaction pendante pour `inf$2`.

La deuxième ligne correspond à une interaction de `user_d` sur `bin_t`. Seule la signature `inf$2` comporte cette interaction mais celle-ci n'est pas pendante sur le signature. Cette dernière ne progresse donc pas.

La troisième ligne correspond à une interaction de `passwd_d` sur `ssh_d`. Seule la signature `inf$1` comporte cette interaction et celle-ci pendante. Cependant, elle ne possède pas l'opération `signal` comme opération élémentaire possible. La signature ne progresse donc pas.

La quatrième ligne fait progresser la signature `inf$1`. À la fin de l'analyse de cette ligne, les signatures sont dans l'état suivant :

`shadow_t` → `passwd_d` → `ssh_d` → `user_d`

`shadow_t` → `passwd_d` → `login_d` → `admin_d` → `bin_t` → `user_d`

La cinquième ligne fait progresser la signature `inf$2`. La sixième ligne devrait faire progresser la signature `inf$1`. Cependant, l'interaction pendante de cette signature est la dernière interaction. Une alerte est donc levée et l'interaction reste pendante :

```
13 Mar 10:13:28 shadow_t-->passwd_d-->ssh_d-->user_d ssh_d->user_d:write;
```

La septième ligne fait progresser la signature `inf$2`. La huitième lève une seconde alerte pour la signature `inf$1` :

```
13 Mar 10:14:00 shadow_t-->passwd_d-->ssh_d-->user_d ssh_d->user_d:write;
```

À la fin de la lecture de la trace, les signatures sont dans l'état suivant :

`shadow_t` → `passwd_d` → `ssh_d` → `user_d`

`shadow_t` → `passwd_d` → `login_d` → `admin_d` → `bin_t` → `user_d`

L'algorithme 2 correspond au mécanisme de détection des signatures simples. Il prend en paramètres une trace, c'est-à-dire un ensemble ordonné d'interactions, et une base de signatures. La sortie de cet algorithme est l'ensemble des alertes levées par la complétion de signatures lors de l'analyse de la trace. On considère que $I \in S$ est vrai si l'interaction I est une étape de la signature S. La fonction $Pendante(S)$ renvoie l'interaction pendante de la signature S. La fonction $Suivant(S)$ renvoie l'interaction suivant l'interaction pendante de S. Enfin, la fonction $Alerte(S, I)$ crée une alerte de complétion de la signature S par l'interaction I.

Algorithme 2 Algorithme de détection des signatures simples

Entrée: $BASE$ un ensemble de signatures,
$TRACE$ un ensemble ordonné d'interactions
Sortie: $ALERTES$ l'ensemble des alertes levées par la complétion des signatures

1: $ALERTES \leftarrow \emptyset$
2: **pour tout** interaction $I(x \rightarrow y) \in TRACE$ **faire**
3: **pour tout** signature $S \in BASE$ telle que $I \in S$ **faire**
4: **si** $Pendante(S) = I$ **alors**
5: **si** $Suivant(S) = \emptyset$ **alors**
6: $ALERTES \leftarrow ALERTES \cup Alerte(S, I)$
7: **sinon**
8: $Pendante(S) \leftarrow Suivant(S)$

Cet algorithme parcourt toutes les signatures pour chaque ligne de la trace. Puis, à chaque étape, une signature peut soit avancer soit ne rien faire. L'action d'avancer une signature s'effectue en temps constant. La complexité en temps de l'algorithme 2

est donc en $O(s.t)$ où s représente le nombre de signatures de la base et t le nombre de lignes de la trace.

Lors de son déroulement, cet algorithme conserve uniquement en mémoire toutes les signatures ainsi que leur avancement. L'avancement de chaque signature prenant une place fixe en mémoire, la complexité en espace est donc en $O(\mathcal{S})$ où $\mathcal{S}$ représente la base de signatures.

Cas des signatures composées

Dans cette section, nous traiterons deux propriétés (p et q) de séparation des privilèges de la forme $C_2^p \circ C_1^p$ et $C_2^q \circ C_1^q$ où C_1^p (resp. C_1^q) correspond à la composante de modification de la propriété p (resp. q) et C_2^p (resp. C_2^q) correspond à la composante d'exécution de la propriété p (resp. q).

Avant de débuter l'analyse des traces, la première interaction de chaque composante de chaque propriété est définie comme *active* et les composantes initiales de chaque propriété sont définies comme *complétables*. Dans le cas de la séparation des privilèges, il s'agit des composantes de modification (C_1) du contexte cible. La progression de ces composantes s'effectue de la même manière que celle des signatures simples. Quand une des composante est complétée, si elle est complétable, les autres composantes complétables appartenant à la même propriété sont désactivées et les composantes suivantes sont définies comme complétables. Sinon la composante complétée reste dans le même état. Pour la séparation des privilège, une fois une composante de modification (C_1) complétée, les composantes d'exécution (C_2) de la même propriété sont activées. Lorsqu'une des composantes finales est complétée alors qu'elle est complétable, une alerte est levée et toutes les composantes appartenant à la même propriété restent dans l'état courant.

Pour illustrer le mécanisme de détection des signatures composées, nous utiliserons les propriétés de séparation des privilèges de `ssh_d` envers `bin_t` et de `user_d` envers `bin_t`. Voici leur composition :

Composantes de modification de `bin_t` par `ssh_d` (C_1^p) :

- $\texttt{ssh\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t}$ (1)
- $\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t}$ (2)

Composantes d'exécution de `bin_t` par `ssh_d` (C_2^p) :

- $\texttt{ssh\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{execute} \texttt{bin\_t}$ (3)
- $\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d} \xrightarrow{execute} \texttt{bin\_t}$ (4)
- $\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{execute} \texttt{bin\_t}$ (5)

Composante de modification de `bin_t` par `user_d` (C_1^q) :

- $\texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t}$ (6)

Composantes d'exécution de `bin_t` par `user_d` (C_2^q) :

- $\texttt{user\_d} \xrightarrow{execute} \texttt{bin\_t}$ (7)

(ssh_d, user_d)	2,4,5
(ssh_d, admin_d)	1,3
(admin_d, bin_t)	1,2,3,5,6,8
(user_d, admin_d)	2,5,6,8
(user_d, bin_t)	4,7

TABLE 1.9 – Table de correspondance des signatures composées

– $\texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{execute} \texttt{bin\_t}$ (8)

Dans le cas de signatures composées, la table de correspondance associe les paires source/cible aux composantes les utilisant. La table 1.9 représente la table de correspondance générée à partir des composantes précédentes.

Durant cette exemple nous analyserons la trace suivante :

```
1: Jan 13 10:12:06 transition scontext=ssh_d tcontext=user_d
2: Jan 13 10:12:52 transition scontext=ssh_d tcontext=admin_d
3: Jan 13 10:13:03 write scontext=admin_d tcontext=bin_t
4: Jan 13 10:13:10 transition scontext=user_d tcontext=admin_d
5: Jan 13 10:13:21 execution scontext=user_d tcontext=bin_t
6: Jan 13 10:13:28 write scontext=admin_d tcontext=bin_t
```

Avant d'analyser la trace, la première interaction de chaque composante est activée et les composantes 1, 2 et 6 sont définies comme complétables.

$\boxed{\texttt{ssh\_d} \xrightarrow{trans} \texttt{admin\_d}} \xrightarrow{write} \texttt{bin\_t}$ (1)

$\boxed{\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d}} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t}$ (2)

$\boxed{\texttt{ssh\_d} \xrightarrow{trans} \texttt{admin\_d}} \xrightarrow{execute} \texttt{bin\_t}$ (3)

$\boxed{\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d}} \xrightarrow{execute} \texttt{bin\_t}$ (4)

$\boxed{\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d}} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{execute} \texttt{bin\_t}$ (5)

$\boxed{\texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d}} \xrightarrow{write} \texttt{bin\_t}$ (6)

$\boxed{\texttt{user\_d} \xrightarrow{execute} \texttt{bin\_t}}$ (7)

$\boxed{\texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d}} \xrightarrow{execute} \texttt{bin\_t}$ (8)

Après analyse de la ligne 3, la composante 1 est complétée. Les autres composantes complétables de la propriété de séparation de privilèges de `ssh_d` envers `bin_t`, c'est-à-dire la composante 2, sont désactivés. De plus, ces composantes suivantes, c'est-à-dire les composantes 3, 4 et 5, sont définies comme complétables.

$\texttt{ssh\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t}$ (1)

$\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t}$ (2)

$\texttt{ssh\_d} \xrightarrow{trans} \boxed{\texttt{admin\_d} \xrightarrow{execute} \texttt{bin\_t}}$ (3)

$\texttt{ssh\_d} \xrightarrow{trans} \boxed{\texttt{user\_d} \xrightarrow{execute} \texttt{bin\_t}}$ (4)

$$\texttt{ssh\_d} \xrightarrow{trans} \boxed{\texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d}} \xrightarrow{execute} \texttt{bin\_t} \quad (5)$$

$$\boxed{\texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d}} \xrightarrow{write} \texttt{bin\_t} \quad (6)$$

$$\boxed{\texttt{user\_d} \xrightarrow{execute} \texttt{bin\_t}} \quad (7)$$

$$\boxed{\texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d}} \xrightarrow{execute} \texttt{bin\_t} \quad (8)$$

Après analyse de la ligne 5, la composante 5 est complétée. Celle-ci faisant partie des composantes finales, une alerte est levée pour la propriété correspondante et toutes les composantes complétables de cette propriété restent dans l'état précédant l'analyse de cette ligne. La composante complétée est en attente d'une autre complétion de la signature. La composante 8 n'est pas complétée car elle n'est pas définie comme complétable.

$$\texttt{ssh\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t} \quad (1)$$

$$\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d} \xrightarrow{trans} \texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t} \quad (2)$$

$$\texttt{ssh\_d} \xrightarrow{trans} \boxed{\texttt{admin\_d} \xrightarrow{execute} \texttt{bin\_t}} \quad (3)$$

$$\texttt{ssh\_d} \xrightarrow{trans} \boxed{\texttt{user\_d} \xrightarrow{execute} \texttt{bin\_t}} \quad (4)$$

$$\texttt{ssh\_d} \xrightarrow{trans} \texttt{user\_d} \xrightarrow{trans} \boxed{\texttt{admin\_d} \xrightarrow{execute} \texttt{bin\_t}} \quad (5)$$

$$\texttt{user\_d} \xrightarrow{trans} \boxed{\texttt{admin\_d} \xrightarrow{write} \texttt{bin\_t}} \quad (6)$$

$$\boxed{\texttt{user\_d} \xrightarrow{execute} \texttt{bin\_t}} \quad (7)$$

$$\texttt{user\_d} \xrightarrow{trans} \boxed{\texttt{admin\_d} \xrightarrow{execute} \texttt{bin\_t}} \quad (8)$$

À la fin de l'analyse de la trace, la composante 6 est complétée. Les composantes 7 et 8 sont donc définies comme complétables.

1.5 Application

Afin de tester les performances de PIGA en conditions réelles, l'équipe SDS a mis en place un pot de miel sur lequel PIGA IDS a été installé. De plus, le LIFO a participé au défi sécurité Sec&Si organisé par l'ANR. Le but du défi était de proposer un système d'exploitation sécurisé basé sur Linux.

Enfin, PIGA a été adapté afin de l'utiliser pour sécuriser d'autres systèmes que Linux, notamment Windows et des architectures en nuage.

1.5.1 Honeypot

Durant deux ans, PIGA a été utilisé sur un pot de miel à haute interaction afin d'en tester l'efficacité [BRCTZ09]. Un pot de miel est un système rendu volontairement accessible et vulnérable aux attaques. Il peut être notamment utilisé pour obtenir des données sur les attaques mises en place par les potentiels attaquants.

Le pot de miel était composé d'une passerelle, d'un serveur et d'un poste de travail. Plus de 1 300 000 signatures ont été calculées à partir de la politique de

sécurité mise en place.

En deux ans, le pot de miel a subi plus de deux millions d'attaques mais n'a jamais été corrompu.

1.5.2 Défi sécurité

De 2008 à 2010, le LIFO a participé, avec deux autres équipes, au défi Sec&Si organisé par l'ANR [BPT+11]. Le but de ce défi était de déployer un système d'exploitation sécurisé basé sur Linux. Ce système d'exploitation devait comporter un navigateur web, un lecteur de pdf, un lecteur de mails ainsi qu'une suite bureautique. De plus, il devait permettre d'accéder de manière sécurisée à différents services tels que le paiement en ligne, la télédéclaration d'impôts ou les services banquiers en ligne.

Le défi se décomposait en trois phases identiques successives, chacune composée de deux étapes. La première étape était la conception, le développement ainsi que l'amélioration du système d'exploitation. La seconde était la phase d'attaque, chaque équipe tentant de trouver des vulnérabilités dans le système d'exploitation des autres équipes.

Chaque équipe était évaluée en fonction du nombre de vulnérabilités de leur système d'exploitation et de la gravité des attaques réussies (attaque théoriques ou réelles). Le nombre de vulnérabilité que chaque équipe trouvait sur les autres systèmes était également pris en compte. L'équipe du LIFO a remporté le défi en proposant PIGA-OS et a remporté les trois phases du défi.

1.5.3 Adaptation de PIGA

PIGA a été initialement développé pour Linux. Plusieurs projets sont en cours afin d'adapter PIGA à d'autres systèmes.

PIGA-Windows

En partenariat avec le CEA-DAM, PIGA-Windows offre un système MAC pour Windows similaire à SELinux et qui permet d'utiliser PIGA sur Windows 7 [BGBT12]. PIGA-Windows possède ainsi deux niveaux de contrôle d'accès mandataire. Le premier est un driver noyau inspiré par le système MAC SELinux gérant les propriétés de sécurité simples. Le second est un processus, nommé PIGA-Monitor, utilisant le driver noyau et gérant les propriétés de sécurité définissant la politique de PIGA.

PIGA-Cloud

En partenariat avec Alcatel-Lucent, le projet Seed4C [1] vise à mettre en place un moyen de sécuriser les infrastructures en nuage [ABB+12]. PIGA-Cloud est composé de trois mécanismes de protection : le contrôle du système d'exploitation, le contrôle des machines virtuelles et le contrôle de la machine virtuelle Java (JVM). Chaque

1. Seed4C est un projet européen impliquant de nombreux partenaires : `http://www.celticplus-seed4c.org/`

machine virtuelle possède une instance de SELinux en plus de celle du système d'exploitation. Un système MAC développé pour la JVM, SEJava, offre également une protection similaire à SELinux et empêche de profiter des vulnérabilités de la JVM. PIGA-Shared offre des propriétés de PIGA pour ces différents types de systèmes MAC.

1.6 Conclusion

Les applications autour du pot de miel et du défi sécurité ont permis de montrer que PIGA est une solution viable pour une utilisation réelle. De plus, les travaux récents sur le Cloud et Java montrent l'extensibilité de l'approche.

1.6.1 Efficacité de PIGA

La première place de PIGA au défi sécurité est liée à la facilité d'expression des propriétés et à l'énumération complète de vulnérabilités, qui permet de bloquer des comportements complexes impliquant, notamment, des transitions ou des flux d'informations indirects.

Toutefois, cette exhaustivité est théorique. En effet, si les différents graphes représentant le système sont trop grands, il est possible que les ressources du système les calculant soient insuffisantes. Les signatures sont alors limitées en taille afin de permettre aux calculs de s'effectuer. Cette limitation n'a cependant qu'un impact léger sur la sécurité du système car, plus une signature est grande, plus l'attaque qu'elle représente est complexe et difficile à mettre en place. Les signatures non générées représentent donc les comportements illicites les plus difficiles à réaliser. De plus, la génération de signatures s'effectuant hors-ligne, elle peut être externalisée, afin d'effectuer les calculs de signatures sur un système possédant plus de ressources.

L'exhaustivité de PIGA conduit à devoir gérer une base de signatures très volumineuse. La taille de cette base est directement liée à la taille du système et de la politique MAC. En effet, plus la politique MAC contient de règles et plus les contextes représentant le système sont nombreux, plus les signatures seront nombreuses et de taille importante. Or, celles-ci doivent être chargées en mémoire lors de la détection. La taille de la base de signatures peut donc nuire aux performances des systèmes protégés par PIGA.

L'approche PIGA a deux avantages. Premièrement, des nouveaux types de propriétés peuvent facilement être ajoutés à la politique à condition que ceux-ci soient modélisables à partir des différents graphes. Deuxièmement, la forme des signatures permet au mécanisme de détection d'indiquer rapidement si une action est autorisée ou non. Cependant, si ce temps de réponse est faible, celui-ci augmente avec le nombre et la taille des signatures.

L'une des difficultés majeures de l'utilisation de PIGA est sa mise en place. En effet, son installation est assez intrusive pour le système. Il est notamment nécessaire de patcher le noyau de Linux afin d'avoir accès aux appels système et de pouvoir les bloquer. De plus, la définition de la politique de sécurité demande une bonne connaissance des contextes de sécurité et de ce qu'ils représentent. En effet, une

propriété de sécurité trop générale engendrerait le blocage de comportement non-malicieux.

1.6.2 Problématique

Définition du problème

Tout au long de l'utilisation d'un système d'exploitation, le nombre d'applications installées sur ce système augmente de manière naturelle. Ces applications ne peuvent pas s'intégrer à un ou plusieurs contextes existants. Par conséquent, l'installation de nouvelles applications implique la création de nouveaux contextes, ainsi que l'ajout ne nouvelles propriétés de sécurité à la politique. Le but de cette étude est d'améliorer l'extensibilité de PIGA et ainsi, permettre un meilleur passage à l'échelle. En effet, un des problèmes impliqués par l'ajout d'applications est l'augmentation du nombre de signatures générées par PIGA, ainsi que l'augmentation de leur taille.

Durant cette thèse, nous nous intéressons donc au problème de la taille de la base de signatures. Cette base peut atteindre plusieurs centaines de Mo, notamment si le système à sécuriser possède une interface graphique. En effet, la présence d'une telle interface augmente de manière importante le nombre de contextes de sécurité et donc la taille du graphe d'interactions. Les signatures étant des chemins dans les différents graphes dérivés du graphe d'interactions, leur nombre peut augmenter considérablement.

Afin de gagner l'espace mémoire, on peut soit envisager une compression classique de la base de signatures initiale par de algorithmes de compression de données, soit utiliser des méthodes de factorisation. Dans les deux cas, le mécanisme de détection doit conserver sa capacité à gérer l'avancement des signatures et à lever des alertes quand c'est nécessaire. Dans le cadre d'une compression classique, il est possible de décompresser uniquement les signatures qui sont "actives" et d'utiliser la méthode initiale de détection déjà intégrée à PIGA. Le problème d'une telle approche est que, au cours du temps, toutes les signatures seront activées et devront être intégrées dans la partie détection. Cette approche ne tient donc pas sur l'échelle du temps. Dans le cas d'une factorisation des signatures, le mécanisme de détection doit être repensé pour prendre en compte les modifications induites par cette factorisation. Ces modifications peuvent notamment concerner la forme des signatures. L'approche utilisée dans cette thèse est de rechercher les signatures ayant des similarités et les factoriser dans un nombre réduit de signatures.

Nous avons alors considéré deux types d'approches travaillant sur des structures de données différentes. Premièrement, nous pouvons travailler sur le graphe d'interactions. En effet, dans la mesure où les signatures sont des chemins dans des graphes dérivés de celui-ci, si nous réduisons la taille de ce graphe, alors nous diminuerons le nombre de signatures générées. Deuxièmement, nous pouvons travailler directement sur la base de signatures. Dans ce cas, nous nous posons deux questions :

- Est-il possible de changer la structure de la base de manière à en diminuer la taille ?
- Est-il possible de supprimer des éléments de la base ?

Quel que soit le type d'approche, le traitement de notre problème s'effectue sous plusieurs contraintes.

Tout d'abord, il est nécessaire que la base de signatures générée par notre solution soit équivalente à la base originale. Cela ce traduit par deux contraintes. Premièrement, tout comportement représenté par une signature dans la base de signatures original doit être également représenté dans la base générée par notre solution. Deuxièmement, il ne doit pas exister de comportement représenté dans la base générée par notre solution qui ne soit pas représenté dans la base originale.

De plus, il est nécessaire de conserver la génération à partir d'un graphe représentant le système, afin de respecter le fonctionnement de PIGA.

Le calcul de signatures s'effectuant hors-ligne, le temps pris par celui-ci n'est pas réellement restrictif.

Nous nous sommes ici intéressés aux signatures générées par les propriétés de confidentialité, car ce sont les propriétés générant le plus de signatures. De plus, ces signatures sont simples à calculer, car elles sont uniquement composées d'un chemin dans le graphe de flux.

Approches

Durant cette thèse, nous avons étudié deux approches différentes.

La première approche apporte une modification des graphes utilisés lors de la génération de signatures. Dans le cas des propriétés de confidentialité, les graphes impliqués sont les graphes d'interactions et de flux. Afin de réduire la taille de ces graphes, nous regroupons les sommets ayant le même comportement sous forme d'un *méta*-sommet (ou *module*). On considère que deux sommets ont le même comportement s'ils possèdent le même voisinage. Dans un graphe orienté, deux sommets ont le même comportement s'ils ont les mêmes successeurs et les mêmes prédécesseurs. Ainsi, sur le graphe de flux de la figure 1.4, les sommets `login_d` et `ssh_d` ont le même comportement. En effet, ils ont tous les deux `passwd_d` comme prédécesseur et, `admin_d` et `user_d` comme successeurs. Nous pouvons donc regrouper ces sommets dans un seul *méta*-sommet. Dans la section 1.3, nous avons calculé quatre signatures de `shadow_t` vers `user_d`, après regroupement des deux sommets, on obtient deux signatures :

- `shadow_t` > `passwd_d` > *méta*-sommet > `user_d`
- `shadow_t` > `passwd_d` > *méta*-sommet > `admin_d` > `bin_t` > `user_d`

Cette approche consiste à travailler sur le graphe, il est donc difficile de savoir si le temps de calcul de la base de signature va être augmenté ou diminué. En effet, il est nécessaire de calculer le nouveau graphe ce qui implique une augmentation du temps nécessaire. Cependant, le graphe obtenu étant plus petit, la recherche de signature peut être plus rapide.

La seconde approche cherche à supprimer les signatures qui ne sont pas complétables à cause de la présence d'autres signatures dans la base. Durant notre étude, nous nous intéressons aux signatures représentant un comportement incluant un autre comportement interdit. Par exemple, considérons les comportements suivants comme interdits :

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

La deuxième signature n'est pas complétable car il inclut la première qui est interdite. Nous pouvons donc supprimer la deuxième signature. Cette approche consiste à travailler directement sur la base de signatures, après que le calcul des signatures soit effectué. Le temps total nécessaire pour obtenir la base réduite est donc plus long que le temps du calcul de la base originale.

Il est intéressant de constater que ces deux approches sont combinables. Cependant, nous verrons dans la section 3.5 que chacune de ces approches possède des limites d'utilisation.

Chapitre 2

Décomposition modulaire

Au chapitre précédent, nous avons vu l'intérêt de grouper les sommets (i.e. les contextes de sécurité) ayant globalement le même comportement vis-à-vis du reste du système. Dans ce chapitre, nous présenterons l'outil théorique qui permet d'effectuer cette opération sur des graphes quelconques : la décomposition modulaire. Ce chapitre se base sur les travaux de C. Paul [Pau06, HP10], F. De Montgolfier [dM03, MdM05] et M. Rao [Rao06].

Dans la section 2.1, nous verrons les fondements théoriques de cette décomposition. Puis, dans la section 2.2, nous présenterons la notion de module d'un graphe, ainsi que les structures qu'il est possible de générer à partir de cette notion. Ensuite, dans la section 2.3, nous étudierons les algorithmes de décomposition modulaire pour les graphes orientés et non-orientés ainsi que leur complexité. Enfin, nous verrons, dans la section 2.4, quelques applications de la décomposition modulaire à des problèmes de la théorie des graphes ainsi qu'à des problèmes pratiques.

2.1 Fondements théoriques

Pour ce chapitre, nous supposons que le lecteur est familier avec les notions principales de la théorie des graphes. On pourra se référer aux ouvrages de référence [Ber85, BM08, Epp99]. Pour plus de clarté, nous utiliserons les notations suivantes. Pour tout graphe $G = (V, E)$:

- $A_G(n)$ représente l'ensemble des sommets adjacents au sommet n dans le graphe G ;
- $\overline{A}_G(n)$ représente l'ensemble des sommets de G qui n'appartiennent pas à $A_G(n)$;
- $\overline{G}$ représente le graphe complémentaire de G ;
- G_N représente le sous-graphe de G induit par $N \subseteq V$, c'est-à-dire $G_N = (N, E \cap N \times N)$;
- on notera xy une arête de G si G est non-orienté ;
- on notera (x, y) un arc de G si G est orienté.

2.1.1 Partition et graphe quotient

On rappelle qu'une partition $\mathcal{P} = \{P_1, ..., P_k\}$ d'un ensemble S est un sous-ensemble de l'ensemble des parties de S, $\mathcal{P}(S)$, tel que :

- $\forall i\ P_i \subset S$
- $\forall i\ P_i \neq \emptyset$
- $\forall i, j\ P_i \cap P_j = \emptyset$ pour $i \neq j$
- $\bigcup_{i=1}^{k} P_i = S$

À partir d'une partition $\mathcal{P}$ de l'ensemble des sommets d'un graphe G, on peut générer un graphe $G_\mathcal{P}$ nommé graphe *quotient* de G suivant $\mathcal{P}$.

Définition 7 (Graphe quotient)
Soit $G = (V, E)$ un graphe et $\mathcal{P}$ une partition de V. Le ***graphe quotient*** *$G_\mathcal{P}$ est défini comme suit :*

- *$\mathcal{P}$ est l'ensemble des sommets ;*
- *$E_\mathcal{P}$ est l'ensemble des arêtes tel que :*

$$XY \in E_\mathcal{P} \text{ si } X \in \mathcal{P}, Y \in \mathcal{P} \text{ et } \exists\ x \in X, y \in Y \mid xy \in E$$

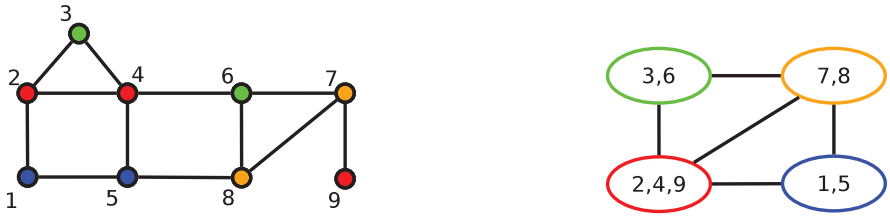


FIGURE 2.1 – Graphe G et le graphe quotient $G_\mathcal{P}$

La figure 2.1 représente le graphe G sur lequel est visible la partition $\mathcal{P} = \{\{1, 5\}\{2, 4, 9\}\{3, 6\}\{7, 8\}\}$. Tous les sommets d'une même couleur appartiennent au même élément de la partition. Cette figure montre également le graphe quotient $G_\mathcal{P}$. On peut voir que l'élément $\{2, 4, 9\}$ est adjacent à l'élément $\{7, 8\}$ car $7 \in A_G(9)$.

Le passage au graphe quotient fait perdre deux types d'informations :

- il est difficile de savoir les interactions réelles entre deux éléments de la partition. Par exemple, pour $\{2, 4, 9\}$ adjacent $\{7, 8\}$, on peut avoir au maximum 6 arêtes sur le graphe d'origine, et au minimum une seule arête.
- toutes les arêtes à l'intérieur de chaque élément de la partition sont, de fait, ignorées.

Comme nous le verrons plus loin, la décomposition modulaire permet de résoudre le premier problème et de diminuer les pertes d'informations sur le second. En effet, le principe de cette décomposition étant de grouper les sommets ayant le même comportement vis-à-vis du reste du graphe, si deux éléments de la partition sont adjacents sur le graphe quotient, alors tous les sommets du premier élément sont adjacents à ceux du deuxième et inversement. De plus, chaque sous-ensemble est re-partitionné de manière récursive. Suivant le type des éléments de chaque partition, il

est possible de connaître les arêtes entre deux éléments d'une même partition. Dans le cas de graphe non-orienté, seul un type ne le permet pas.

L'ensemble des partitions et sous-partitions décrivent une famille partitive.

2.1.2 Famille partitive

Afin de définir ce qu'est une *famille partitive*, nous utilisons les notions de différence symétrique, ainsi que de chevauchement de deux ensembles. La différence symétrique de deux ensembles A et B est notée $A\Delta B = (A \cup B) \setminus (A \cap B)$. Deux ensembles A et B se chevauchent si $A \cap B \neq \emptyset$, $A \setminus B \neq \emptyset$ et $B \setminus A \neq \emptyset$. Nous utilisons la relation $A \perp B$ pour signifier que A et B se chevauchent.

Définition 8 (Famille partitive)
*Soit $\mathcal{F} \subseteq 2^S$ une famille de parties de S. $\mathcal{F}$ est **partitive** si :*

1. *$S \in \mathcal{F}$, $\emptyset \notin \mathcal{F}$ et pour tout $x \in S$, $\{x\} \in \mathcal{F}$*
2. *Pour toute paire de parties $A, B \in \mathcal{F}$ telles que $A \perp B$:*
   - *(a) $A \cap B \in \mathcal{F}$;*
   - *(b) $A \cup B \in \mathcal{F}$;*
   - *(c) $A \setminus B \in \mathcal{F}$ et $B \setminus A \in \mathcal{F}$;*
   - *(d) $A\Delta B \in \mathcal{F}$;*

Définition 9 (Partie forte)
*Une partie $P \in \mathcal{F}$ est dite **forte** si elle ne chevauche aucune autre partie de $\mathcal{F}$. L'ensemble des éléments forts de $\mathcal{F}$ est noté $\mathcal{F}_S$.*

Ainsi, pour deux éléments A et $B \in \mathcal{F}_S$, on a soit $A \subset B$, soit $B \subset A$, soit A et B disjoints. Les ensembles S et $\{x\}$, pour tout $x \in S$, sont des éléments forts triviaux. Nous pouvons donc organiser $\mathcal{F}_S$ en un arbre $T_\mathcal{F}$ dont les sommets sont les éléments de $\mathcal{F}_S$. $T_\mathcal{F}$ est l'arbre d'inclusion des parties fortes de $\mathcal{F}$. Cet arbre est nommé *arbre partitif*.

Il est possible d'étiqueter les sommets internes de l'arbre partitif. Ces sommets peuvent être soit *complets*, soit *premiers*.

Définition 10 (Sommet complet [CHM81])
*Soit N un sommet à k fils de l'arbre partitif $T_\mathcal{F}$. Ses fils sont $F_1^N, ..., F_k^N$. Le sommet N est **complet** si, pour tout $I \subset \{1, ..., k\}$ non vide, on a :*

$$\bigcup_{i \in I} F_i^N \in \mathcal{F}$$

La figure 2.2 représente la famille partitive $\mathcal{F} = \{\{1,2,3,4,5,6,7,8,9\}, \{1,2,3,4\}, \{7,8,9\}, \{7,8\}, \{7,9\}, \{8,9\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}\}$. Les ensembles dessinés en rouge représentent les éléments de $\mathcal{F}$ qui ne sont pas forts.

Les fils d'un sommet de l'arbre d'inclusion n'étant pas ordonnés, on peut dessiner cet arbre de différentes manières, tout en conservant la même hiérarchie. L'ordre dans lequel apparaissent les feuilles, c'est-à-dire les éléments de l'ensemble initial, est une permutation factorisante. Pour une même famille partitive, il existe donc plusieurs permutations factorisantes.

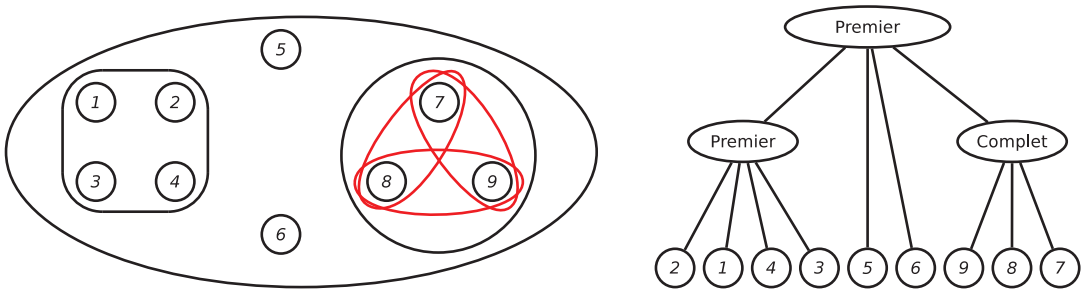


FIGURE 2.2 – Famille partitive $\mathcal{F}$ et son arbre d'inclusion des éléments forts

2.1.3 Permutations factorisantes

Définition 11 (Permutation)
*Une **permutation** σ d'un ensemble S est un ordre total sur S.*

L'ordre des éléments de $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ donné par l'arbre de la figure 2.2 est $(2, 1, 4, 3, 5, 6, 9, 8, 7)$. Cet ordre est une permutation de S.

Définition 12 (Facteur)
*Soient S, un ensemble σ une permutation de S. L'ensemble F est un **facteur** de σ si $F \subset S$ et tous les éléments de F sont consécutifs dans σ.*

Par exemple, les sous-ensembles de S, $\{3, 4, 5\}$ et $\{5, 6, 7\}$, sont des facteurs de la permutation $(1, 2, 3, 4, 5, 6, 7, 8, 9)$. On peut noter que les sommets internes premiers et complets de l'arbre de la figure 2.2 sont des facteurs la permutation donnée par l'ordre des feuilles. Ces facteurs sont $\{1, 2, 3, 4\}$ et $\{7, 8, 9\}$.

Définition 13 (Permutation factorisante [Cap96])
*Soit $\mathcal{F}_S$ l'ensemble des éléments forts de la famille partitive $\mathcal{F}$. Une permutation σ est une **factorisante** pour $\mathcal{F}$, si pour tout ensemble $P \in \mathcal{F}_S$, P est un facteur de σ.*

Ainsi, pour l'ensemble S et les facteurs $\{1, 2, 3, 4\}$ et $\{7, 8, 9\}$, les permutations $(2, 1, 4, 3, 5, 6, 9, 8, 7)$, $(1, 2, 4, 3, 6, 5, 7, 9, 8)$ et $(4, 3, 2, 1, 5, 6, 7, 8, 9)$ sont factorisantes. En effet, dans chacune de ces permutations, les éléments 1, 2, 3 et 4 sont consécutifs, de même pour les éléments 7, 8 et 9.

Dans la suite de chapitre, on appellera permutation factorisante du graphe $G = (V, E)$, une permutation factorisante de V pour la famille des modules de G.

2.2 Module d'un graphe

L'idée du module d'un graphe est de regrouper les sommets qui ont un comportement identique vis-à-vis du reste du graphe, sous forme d'un *méta*-sommet. L'utilisation de modules permet de diminuer la taille du graphe en limitant les pertes d'informations.

Dans cette section, nous verrons la définition d'un module d'un graphe. Puis nous présenterons l'arbre de décomposition modulaire qui ordonne les modules forts d'un

graphe. Ensuite, nous définirons le graphe quotient modulaire à partir de l'arbre de décomposition modulaire. Enfin, nous étudierons les différents types de modules d'un graphe orienté et non-orienté.

2.2.1 Définition d'un module

Pour définir la notion de module, nous pouvons utiliser la notion d'*uniformité* pour un graphe non-orienté. On dit qu'un ensemble de sommets S est uniforme par rapport à un sommet x, si x est voisin de tous les sommets de S ou qu'il est voisin d'aucun sommet de S.

Définition 14 (Uniformité)
*Soient M un ensemble de sommets d'un graphe $G = (V, E)$ et x un sommet de $V \setminus M$. M est **uniforme** par rapport à x si, pour tout $y \in M$, $xy \in E$ ou $xy \notin E$.*

Si on reprend l'idée du module d'un graphe, un module M est donc un sous-ensemble de V *uniforme* par rapport à tout sommet $x \in V \setminus M$.

Définition 15 (Module)
*Soit $G = (V, E)$ un graphe non-orienté. Un **module** M de G est un sous-ensemble de V tel que :*

$$\forall x \in V \setminus M \begin{cases} M \subseteq A_G(x) \text{ ou} \\ M \subseteq \overline{A}_G(x) \end{cases}$$

2.2.2 Arbre de décomposition modulaire

Lemme 1 ([CHM81]) *La famille $\mathcal{M}$ des modules d'un graphe est une famille partitive.*

Les ensembles V et $\{x\}$, pour $x \in V$, sont des modules triviaux du graphe $G = (V, E)$. Un module M est **fort** si c'est un élément fort de la famille des modules $\mathcal{M}$. Ainsi, un module de $\mathcal{M}$ est fort s'il ne chevauche aucun autre module de $\mathcal{M}$. L'ensemble des modules forts $\mathcal{M}_S$ peut donc être organisé sous forme d'un arbre $T_{\mathcal{M}}$. La racine est le module V et les feuilles sont des modules de la forme $\{x\}$, où $x \in V$. Cet arbre est nommé **arbre de décomposition modulaire**. L'arbre de décomposition modulaire de G est noté MT_G. Cet arbre représente l'ordre d'inclusion des modules forts de G.

Définition 16 (Module maximal)
*Un module $M \in \mathcal{M}$ du graphe $G = (V, E)$ est **maximal** si $M \subset V$ et il n'existe pas de module non-trivial $M' \in \mathcal{M}$, tel que $M' \subset V$ et $M \subset M'$.*

L'ensemble des modules forts maximaux $\mathcal{M}$ est une partition de V. Cet ensemble est composé par les sommets de profondeur 1 de l'arbre de décomposition modulaire MT_G. La figure 2.3 représente un graphe sur lequel apparaissent les modules forts. Cette figure comporte également l'arbre de décomposition modulaire sur lequel la partition modulaire maximale est mise en évidence.

Définition 17 (Partition modulaire maximale)
*Soit un graphe $G = (V, E)$, la **partition modulaire maximale** $\mathcal{M}_G$ du graphe G est l'unique partition de V composée des modules forts maximaux de G.*

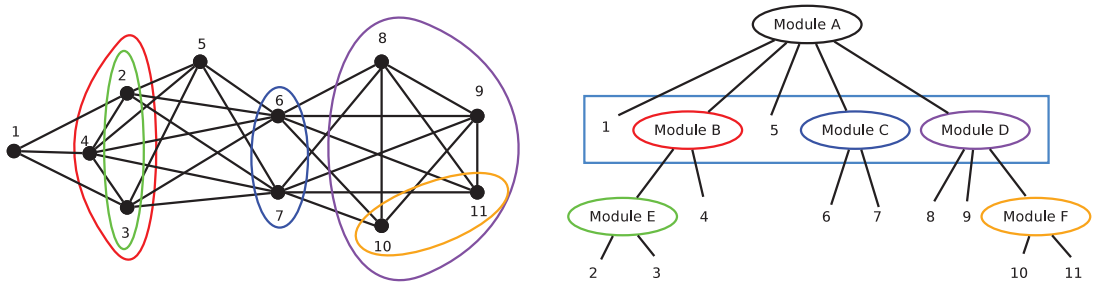


FIGURE 2.3 – Un graphe avec les modules forts et son arbre de décomposition modulaire

2.2.3 Graphe quotient modulaire

À partir de la partition modulaire maximale de G, il est possible de générer le graphe quotient modulaire de G, noté $Quot_G$, en regroupant les sommets d'un même module maximal sous un seul *méta*-sommet. Ce *méta*-sommet est le sommet *représentatif* du module sur le graphe quotient. L'opération consistant à regrouper un module sous son sommet représentatif s'appelle la *contraction*. L'opération inverse est la *substitution* d'un sommet représentatif par le graphe induit du module qu'il représente.

Définition 18 (Graphe quotient modulaire)
Soient un graphe $G = (V, E)$ et $\mathcal{M}_G$ sa partition modulaire maximale, le graphe quotient $G_{\mathcal{M}_G}$ est le ***graphe quotient modulaire*** *de G noté $Quot_G$.*

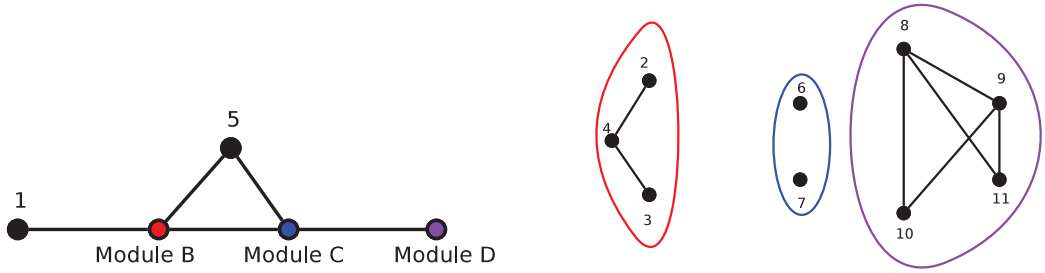


FIGURE 2.4 – Le graphe quotient modulaire du graphe de la figure 2.3 et le graphe induit de chacun de ses modules

L'utilisation de la décomposition modulaire permet de diminuer la perte d'informations lors du passage au graphe quotient. En effet, la définition même d'un module fait que si deux modules M_1 et M_2 sont adjacents sur le graphe quotient modulaire, alors, pour tout $x \in M_1$ et $y \in M_2$, on a $y \in A_G(x)$ et $x \in A_G(y)$. Ainsi, seules les arêtes présentes entre deux éléments d'un même module sont inconnues. Il est donc nécessaire de conserver les graphes induits de chacun des modules forts maximaux afin de ne pas perdre d'informations. La figure 2.4 présente le graphe quotient du graphe de la figure 2.3 ainsi que les graphes induits de chacun de ses modules forts maximaux.

2.2.4 Types de modules

Il est possible de ne pas avoir à disposer du graphe induit de chaque module, pour pouvoir reconstituer le graphe d'origine à partir du graphe quotient. Pour cela, nous avons besoin d'informations supplémentaires sur ce module. Cette information est le type du module, celui-ci correspondant à certaines propriétés que le graphe, induit par ce module, possède.

Module d'un graphe non-orienté

Un module M du graphe non-orienté G peut être de différents types, suivant la forme du graphe quotient du graphe induit G_M. Si ce graphe est une clique, alors M est *série*. Si c'est un stable, alors M est *parallèle*, sinon M est *premier*.

Définition 19 (Module série)
*Soit M un module de G, M est un **module série** si $Quot_{G_M}$ est une clique.*

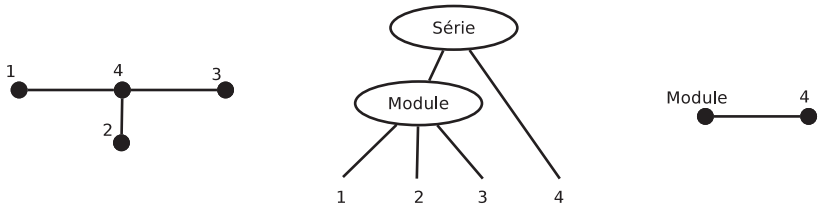


FIGURE 2.5 – Graphe induit d'un module série, son arbre de décomposition modulaire et le graphe quotient associé

La figure 2.5 présente le graphe induit d'un module série composé de 4 sommets. Son graphe quotient modulaire est bien une clique.

Définition 20 (Module parallèle)
*Soit M un module de G, M est un **module parallèle** si $Quot_{G_M}$ est un stable.*

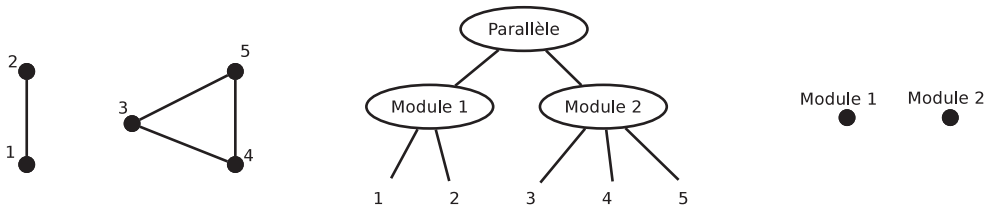


FIGURE 2.6 – Graphe induit d'un module parallèle, son arbre de décomposition modulaire et le graphe quotient associé

La figure 2.6 présente le graphe induit d'un module parallèle composé de 5 sommets. Son graphe quotient modulaire est bien un stable.

Définition 21 (Module premier)
*Soit M un module de G, M est un **module premier** si M n'est ni série ni parallèle.*

La figure 2.7 présente le graphe induit d'un module premier composé de 5 sommets. Son graphe quotient modulaire n'est ni une clique ni un stable.

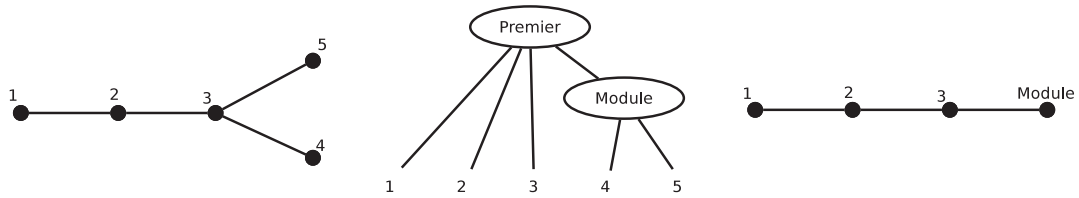


FIGURE 2.7 – Graphe induit d'un module premier, son arbre de décomposition modulaire et le graphe quotient associé

Module d'un graphe orienté

Les modules de type *série* et *parallèle* sont également définis pour les graphes orientés. Le type ordre est défini uniquement pour les graphes orientés. Un module de type *ordre* ou *linéaire* est un module dont le graphe quotient est un ordre total.

Définition 22 (Module ordre)
Soit M un module de G, M est un ***module ordre*** *si $Quot_{G_M}$ représente un ordre total.*

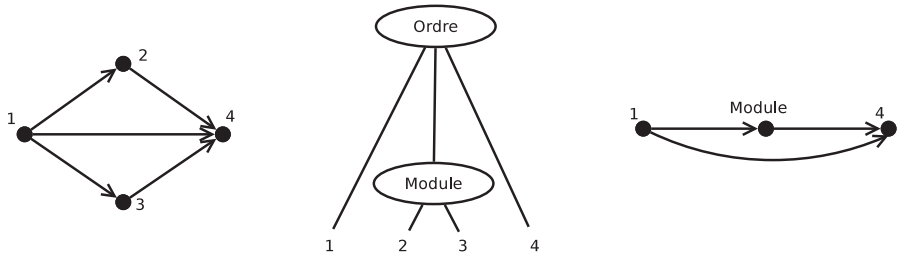


FIGURE 2.8 – Graphe induit d'un module ordre, son arbre de décomposition modulaire et son graphe quotient

La figure 2.8 représente le graphe induit d'un module de type ordre composé de 4 sommets. Son graphe quotient modulaire représente bien un ordre total.

Il est alors nécessaire de redéfinir le type premier. Un module est premier s'il n'est d'aucun autre type.

Grâce à ces étiquettes sur les modules, il est désormais nécessaire de stocker uniquement les graphes quotients du graphe induit de chaque module premier et l'arbre de décomposition modulaire dans le cas de graphes non-orientés. Pour les graphes orientés, il est également nécessaire de stocker, pour chaque module de type ordre, l'ordre des sommets de son graphe quotient.

2.3 Algorithme de décomposition modulaire

L'objectif de l'algorithme de décomposition modulaire est de calculer l'arbre de décomposition modulaire du graphe en entrée. Il se déroule en deux étapes. Tout d'abord, il calcule une permutation factorisante du graphe. Puis, à partir cette permutation factorisante, il calcule l'arbre de décomposition modulaire. La différence

entre les décompositions modulaires orientée et non-orientée se fait sur le calcul de la permutation factorisante.

Le calcul de la permutation factorisante s'effectue en $O(n+m)$, que le graphe soit orienté ou non. Le calcul de l'arbre de décomposition modulaire, à partir d'une permutation factorisante, s'effectue également en $O(n+m)$. L'algorithme de décomposition modulaire d'un graphe s'effectue donc en $O(n+m)$, à la fois pour les graphes orientés et non-orientés.

Cette section décrit le fonctionnement des algorithmes de décomposition modulaire orientée et non-orientée. Il n'est pas nécessaire de la lire pour comprendre la suite du manuscrit.

2.3.1 Calcul d'une permutation factorisante d'un graphe non-orienté

En 1999, M. Habib, C. Paul et L. Viennot [HPV99] décrivent un algorithme calculant une permutation factorisante d'un graphe non-orienté. Cet algorithme utilise les partitions ordonnées, qu'il affine afin d'obtenir la permutation factorisante. Pour affiner une partition ordonnée, on utilise deux règles : la règle du *centre* et la règle du *pivot*.

Partition ordonnée

Une partition ordonnée de l'ensemble de sommets V est un ordre partiel, où les parties sont nommées *classes*. Ces classes sont des sous-ensembles de V numérotés. Ainsi, deux sommets appartenant à deux classes différentes sont ordonnés suivant leur numéro de classe.

Soit une partition ordonnée $\mathcal{O} = C_1 \oplus C_2 \oplus ... \oplus C_k$, $C_i \oplus C_j$ signifie que la classe C_j est consécutive à la classe C_i.

Une partition ordonnée $\mathcal{O} = C_1 \oplus ... \oplus C_k$ de V est *factorisante* pour $G = (V, E)$ si, pour tout C_i, il existe une permutation P_i, telle que $(P_1, P_2, ..., P_k)$ est une permutation factorisante de G.

Règle du centre

L'algorithme commence en appliquant la règle du centre sur un sommet choisi. Ce sommet est appelé *centre*.

Lemme 2 ([HPV99])
Étant donné c un sommet de G, la partition ordonnée $\overline{A}_G(c) \oplus \{c\} \oplus A_G(c)$ est factorisante pour G.

Règle du pivot

La partition ordonnée initiale est donc $\overline{A}_G(c) \oplus \{c\} \oplus A_G(c)$, où c est le *centre*. Ensuite, l'affinage va se faire en utilisant la règle du pivot. La relation $C_i \prec_{\mathcal{O}} C_j$ signifie que la classe C_i précède la classe C_j dans la partition ordonnée $\mathcal{O}$.

Lemme 3 ([HPV99])
Soit une partition ordonnée $\mathcal{O} = C_1 \oplus ... \oplus C_k$ de centre c, et soit $p \in C_i$ un sommet tel que la classe C_j avec $i \neq j$ et $C_j \perp A_G(p)$. Si $\mathcal{O}$ est factorisant alors :

- *si $C_i \prec_{\mathcal{O}} \{c\} \prec_{\mathcal{O}} C_j$ ou si $C_j \prec_{\mathcal{O}} \{c\} \prec_{\mathcal{O}} C_i$, alors la partition où C_j est remplacée par $(\overline{A}_G(p) \cap C_j) \oplus (A_G(p) \cap C_j)$ est factorisante ;*
- *sinon, la partition où C_j est remplacée par $(A_G(p) \cap C_j) \oplus (\overline{A}_G(p) \cap C_j)$ est factorisante.*

À partir du lemme 3, on affine la partition ordonnée en conservant le fait qu'elle soit factorisante. Pour cela, on choisit un pivot p, puis on coupe en deux toutes les classes qui chevauchent le voisinage de p, exceptée la classe de p. On effectue cette opération jusqu'à obtenir un point fixe quel que soit le pivot choisi. Ensuite, on applique de nouveau l'algorithme au graphe induit de chaque classe contenant plus d'un sommet.

Exemple d'utilisation de l'algorithme

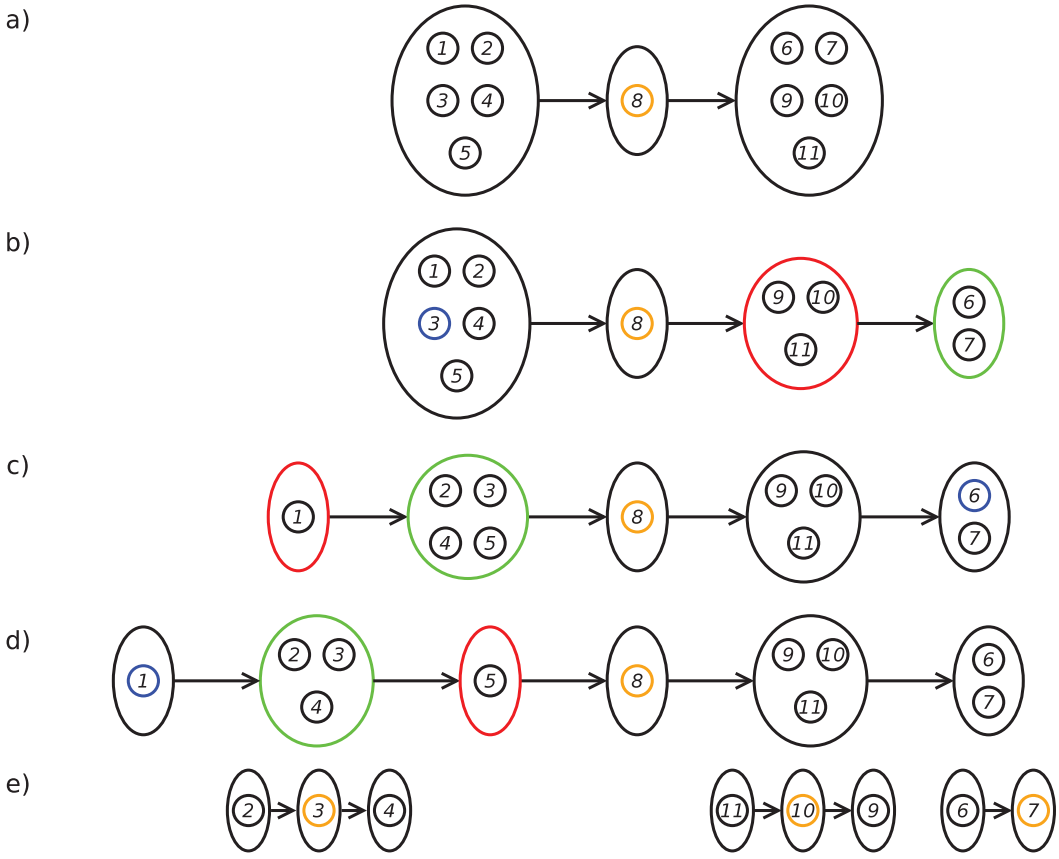


FIGURE 2.9 – Applications de l'algorithme sur le graphe de la figure 2.3

La figure 2.9 montre l'application de l'algorithme calculant une permutation factorisante.

a). Tout d'abord, on applique la règle du centre avec le sommet 8. Ce centre est représenté en orange sur la figure. Ensuite, on applique la règle du pivot sur

les sommets 3, 6 et 1. Ces pivots sont représentés en bleu sur la figure. Dans chaque cas, une seule classe est coupée en deux. Pour chaque classe C coupée en deux par le pivot p, la classe $(\overline{A}_G(p) \cap C)$ est représentée en rouge sur la figure et la classe $(A_G(p) \cap C)$ en vert.

b). Le premier pivot choisi est le sommet 3 de la classe C_p. La classe $C = \{6, 7, 9, 10, 11\}$ chevauche le voisinage du sommet 3. Les sommets 6 et 7 sont voisins de 3, tandis que les sommets sont 9, 10 et 11 ne le sont pas. Nous sommes dans le cas $C_p \prec_{\mathcal{O}} \{8\} \prec_{\mathcal{O}} C$. Donc la classe C est remplacée par $\{9, 10, 11\} \oplus \{6, 7\}$.

c). Ensuite, on applique la règle du pivot sur le sommet 6, la seule classe coupée en deux est la classe $C = \{1, 2, 3, 4, 5\}$. Parmi ces sommets, seul 1 n'est pas voisin de 6. Ici, on est dans le cas $C \prec_{\mathcal{O}} \{8\} \prec_{\mathcal{O}} C_p$. La classe C est donc remplacée par $\{1\} \oplus \{2, 3, 4, 5\}$.

d). Enfin, on applique la règle du pivot sur le sommet 1. Seule la classe $C = \{2, 3, 4, 5\}$ est coupée en deux. Parmi ces sommets, les voisins de 1 sont 2, 3 et 4. Ici, on est dans le cas $C_p \prec_{\mathcal{O}} C \prec_{\mathcal{O}} \{8\}$. La classe C est donc remplacée par $\{2, 3, 4\} \oplus \{5\}$.

e). Après cette application de la règle du pivot, on arrive à un point fixe. On applique donc l'algorithme aux classes contenant plus d'un sommet, c'est-à-dire les classes $\{2, 3, 4\}$, $\{9, 10, 11\}$ et $\{6, 7\}$.

Une fois que toutes les classes de la partition ordonnée contiennent un seul sommet, l'algorithme se termine. On obtient alors la permutation factorisante de l'ensemble des sommets du graphe suivante : (1,2,3,4,5,8,11,10,9,6,7)

Cet algorithme s'effectue en $O(m \log n)$. Pour atteindre une complexité linéaire en $O(n + m)$, M. Habib, F. de Montgolfier et C. Paul [HdMP04] améliore cet algorithme en utilisant les permutations ordonnées de chaînes et une nouvelle règle d'affinage.

2.3.2 Calcul d'une permutation factorisante d'un graphe orienté

Dans le cas des graphes orientés, nous décrirons ici les grandes lignes de l'algorithme sans entrer dans les détails. Il consiste à calculer les arbres de décomposition modulaire de deux graphes non-orientés calculés à partir du graphe d'origine. Ensuite, il utilise les propriétés d'intersections sur les familles partitives pour trouver la permutation factorisante.

Intersection de familles partitives

Soient deux familles partitives $\mathcal{F}_1$ et $\mathcal{F}_2$ sur le même ensemble S. L'*intersection* $\mathcal{F}$ de $\mathcal{F}_1$ et $\mathcal{F}_2$ est la famille des parties de S appartenant à la fois à $\mathcal{F}_1$ et $\mathcal{F}_2$. Elle est notée $\mathcal{F} = \mathcal{F}_1 \cap \mathcal{F}_2$.

Lemme 4 ([dM03])
L'intersection de deux familles partitives est une famille partitive.

À partir de cela, il est défini l'opération $\wedge$ sur les arbres partitifs. Soient deux familles partitives $\mathcal{F}_1$ et $\mathcal{F}_2$ sur le même ensemble S et leurs arbres partitifs respectifs $T_{\mathcal{F}_1}$ et $T_{\mathcal{F}_2}$. $T_{\mathcal{F}_1} \wedge T_{\mathcal{F}_2}$ est l'arbre partitif de la famille partitive $\mathcal{F}_1 \cap \mathcal{F}_2$.

Fonctionnement de l'algorithme

L'algorithme de R. McConnell et F. de Montgolfier [MdM05], permettant de calculer une permutation factorisante de l'ensemble des sommets d'un graphe orienté, se déroule en trois étapes principales.

Tout d'abord, l'algorithme calcule l'arbre de décomposition modulaire de deux graphes non-orientés G_s et G_d dérivés du graphe orienté $G = (V, E)$ en entrée :
- $G_s = (V, E_s)$ tel que $xy \in E_s$ ssi $(x, y) \in E \vee (y, x) \in E$;
- $G_d = (V, E_d)$ tel que $xy \in E_d$ ssi $(x, y) \in E \wedge (y, x) \in E$.

Les arbres de décomposition modulaire obtenus sont MT_{G_s} et MT_{G_d}.

Ensuite, il calcule l'arbre $T_H = MT_{G_s} \wedge MT_{G_d}$. Les sommets de l'arbre T_H sont alors des modules de G_s et de G_d.

Enfin, il ordonne les fils des sommets complets de l'arbre T_H, suivant les propriétés de leur graphe induit dans G. On obtient ainsi une permutation factorisante de l'ensemble des sommets de G.

2.3.3 Calcul de l'arbre de décomposition à partir d'une permutation factorisante

L'algorithme de C. Capelle, M. Habib et F. de Montgolfier [CHdM02], permettant de calculer l'arbre de décomposition modulaire à partir d'une permutation factorisante, s'effectue en $O(n + m)$, dans le cas de graphes orientés et non-orientés.

Cet algorithme se base sur les notions de *séparateurs* et de *fractures*. Un sommet x est un séparateur de $S \subset V$ pour le graphe $G = (V, E)$, si $x \notin S$ et si S n'est pas une module de $G_{S \cup \{x\}}$.

À partir de cette notion de séparateur, on peut définir la fracture d'une paire de sommets consécutifs dans la permutation factorisante. La fracture d'une paire est le plus petit facteur de la permutation contenant cette paire, ainsi que tous les séparateurs de celle-ci. Cette fracture peut être coupée en deux : la fracture gauche et la fracture droite.

En parenthésant la permutation factorisante avec les fractures gauche et droite de chaque paire possible, on obtient une *permutation factorisante parenthésée*.

À partir de cette permutation, on obtient un *arbre des fractures* qui est une approximation de l'arbre de décomposition modulaire. Il suffit de quelques manipulations complémentaires de l'arbre des fractures, pour obtenir l'arbre de décomposition modulaire.

2.4 Applications

2.4.1 Applications sur la théorie des graphes

La plupart des applications de la décomposition modulaire sur d'autres problèmes de la théorie des graphes induisent une réduction du problème aux graphes premiers. En effet, dans beaucoup de problèmes, la résolution est simple dans le cas de stable ou de clique. Or, les graphes quotients des modules parallèles (resp. séries) sont des stables (resp. cliques). Dans le cas de graphe non-orientés, les seuls modules restants sont les modules premiers dont les graphes quotients sont premiers.

Dans cette section, nous ne verrons que deux exemples. Dans sa thèse, M. Rao propose un état de l'art des applications de la décomposition modulaire sur des problèmes de la théorie des graphes [Rao06].

Ensemble stable pondéré

Le problème d'*ensemble stable pondéré* consiste, pour un graphe $G = (V, E)$ et une fonction de pondération $w : V \to \mathbb{N}$, à définir un stable $S \subseteq V$ de G tel que le poids de $S : \sum_{v \in S} w(v)$ soit maximum. On note $\alpha_w(G)$ le poids maximum d'un ensemble stable de G.

À partir de l'arbre de décomposition modulaire, on réduit le problème d'ensemble stable pondéré d'un graphe non-orienté à celui d'ensemble stable pondéré de graphes premiers.

Le principe de la réduction vient du fait que, dans le cas d'un stable ou d'une clique, calculer l'ensemble stable pondéré de poids maximum est trivial. Dans le cas d'un stable, le poids maximum d'un ensemble stable est la somme des poids des sommets du graphe. Dans le cas d'une clique, le poids maximum d'un ensemble stable est le poids maximum d'un sommet. Il est possible d'étendre cela aux modules de type parallèle et série [MR84].

En effet, le graphe quotient d'un module parallèle est un stable. Ainsi, le poids maximum d'un ensemble stable d'un module parallèle est la somme des poids des sommets du graphe quotient de ce module. Le graphe quotient d'un module série est une clique. Ainsi, le poids maximum d'un ensemble stable d'un module parallèle est le poids maximum d'un sommet du graphe quotient de ce module. Dans les deux cas, si un sommet du graphe quotient est le sommet représentatif d'un module, alors son poids est celui de l'ensemble stable pondéré de poids maximum du graphe induit du module qu'il représente.

En appliquant ce principe récursivement sur l'arbre de décomposition modulaire, seul le cas des modules premiers est non-trivial. Ainsi, si on sait résoudre le problème d'ensemble stable pondéré sur les graphes premiers, on peut le résoudre sur n'importe quel graphe non-orienté.

Coloration

La coloration d'un graphe consiste à colorier chacun des sommets du graphe d'une couleur de sorte que deux sommets adjacents soient d'une couleur différente.

Soit un graphe $G = (V, E)$, une *coloration* de G est une partition $\mathcal{P}$ de V telle que toutes les parties de $\mathcal{P}$ sont des ensembles stables de G. Le nombre chromatique $\chi(G)$ d'un graphe correspond au nombre minimum de parties d'une coloration de G, c'est-à-dire le nombre minimum de couleurs nécessaires à sa coloration. Une coloration minimum de G est donc une coloration avec un nombre minimum de couleurs.

À partir de l'arbre de décomposition modulaire, on réduit le problème de coloration d'un graphe non-orienté à celui de coloration de graphes premiers.

De nouveau, le principe de la réduction vient du fait que, dans le cas d'un stable ou d'une clique, calculer la coloration minimum est trivial. Dans le cas d'un stable, tous les sommets sont de la même couleur. Dans le cas d'une clique, chaque sommet a une couleur différente de tous les autres. Il faut donc autant de couleurs qu'il y a de sommets. Il est possible d'étendre cela aux modules de type parallèle et série [MR84].

En effet, le graphe quotient d'un module parallèle étant un stable, le nombre chromatique d'un module parallèle est le nombre chromatique maximum de ses fils dans l'arbre de décomposition modulaire. Le graphe quotient d'un module série étant une clique, le nombre chromatique d'un module série est la somme des nombres chromatiques de ses fils dans l'arbre de décomposition modulaire.

On peut appliquer ce principe récursivement sur l'arbre de décomposition modulaire. Seul le cas des modules premier est non-trivial. Donc, si on sait calculer le nombre chromatique d'un graphe premier, alors on peut le calculer pour n'importe quel graphe non-orienté.

2.4.2 Applications sur des problèmes pratiques

Dessin de graphes

Un des problèmes de la représentation graphique des graphes est d'obtenir un résultat lisible qui respecte certaines règles "esthétiques". Trouver une représentation lisible est particulièrement difficile si le graphe possède beaucoup de sommets et que sa densité est importante.

C. Papadopoulos et C. Voglis [PV05] utilisent la décomposition modulaire afin de résoudre ce problème sur les graphes non-orientés. Pour cela, on applique un algorithme de placement des sommets dépendant du type du module auquel ils appartiennent. Si un module est de type parallèle, alors ses fils sont dessinés sous forme d'une grille. Si un module est de type série, alors ils sont dessinés sous forme d'un cercle. Si un module est de type premier, alors ses fils sont dessinés suivant l'approche d'intégration des forces. Les arêtes ont un pouvoir attractif sur leurs extrémités, tandis que les sommets ont un pouvoir répulsif entre eux. Grâce à une limite de déplacement des sommets qui décroit à chaque itération, on atteint un point fixe.

Si un des fils est un module, alors le graphe induit de ce module est dessiné à son emplacement. Ainsi, on applique cette méthode à tous les modules en commençant par ceux de hauteur la plus élevée jusqu'à la racine de l'arbre de décomposition modulaire.

Réseaux d'interactions protéine-protéine

En biochimie, une interaction protéine-protéine est une interaction physique entre deux protéines. Un complexe protéique est un ensemble de protéines liées par des interactions physiques. Un réseau d'interactions protéine-protéine est un graphe, tel que les sommets sont les protéines et qu'il existe une arête entre deux protéines, s'il existe un complexe protéique auquel ces protéines appartiennent. Ainsi, les complexes protéiques apparaissent comme des cliques, dans le réseau d'interactions protéine-protéine.

J. Gagneur, R. Krause, T. Bouwmeester et G.Casari [GKBC04] utilisent la décomposition modulaire sur ces réseaux afin de trouver les complexes les plus grands possibles. Ce problème consiste à trouver les cliques maximales sur le réseau d'interactions protéine-protéine. Pour cela, on parcourt l'arbre de décomposition modulaire du réseau. L'ensemble des cliques maximales d'un module série est l'ensemble des combinaisons constituées d'une clique maximale de chacun de ses fils dans l'arbre. Quant à l'ensemble des cliques maximales d'un module parallèle, il est l'union des cliques maximales de ses fils.

De plus, la décomposition modulaire offre une représentation compréhensible des règles logiques de coopération des protéines. Des protéines ou des modules fils d'un module parallèle dans l'arbre ne peuvent interagir mais peuvent effectuer des fonctions biologiques proches. Un module série représente, quant à lui, des protéines ou des modules qui fonctionnent ensemble.

2.5 Conclusion

La décomposition modulaire est un outil de la théorie des graphes pour lequel on trouve peu d'applications sur des problèmes pratiques dans la littérature. Pourtant la décomposition modulaire offre de nombreux avantages. Tout d'abord, son exécution s'effectue en temps linéaire. De plus, son application permet, dans certains cas, de réduire un problème sur les graphes non-orientés aux graphes premiers.

Dans cette thèse, nous présentons une application de la décomposition modulaire à la sécurité système. Plus précisément, nous utilisons la décomposition modulaire sur les graphes utilisés par PIGA. Cela permet de réduire la taille les différents graphes utilisés par PIGA. Les signatures étant des chemins dans les différents graphes utilisés par PIGA, la structure des modules fait que les signatures générées à partir d'un graphe quotient compressent de manière efficace les signatures originales, en limitant la sur-approximation.

Dans la partie suivante, nous présenterons la manière dont nous avons intégré la décomposition modulaire au mécanisme de génération de signatures de PIGA. Puis, nous monterons les modifications apportées au mécanisme de détection, afin de gérer la nouvelle forme des signatures. Les limites d'utilisation de la décomposition modulaire sur PIGA seront également présentées.

Deuxième partie

Application de la décomposition modulaire à PIGA-IDS

Chapitre 3

Modification du système de génération de signatures

3.1 Introduction

Le calcul de signatures actuel rencontre deux principaux problèmes. Le premier problème concerne le nombre important de signatures qui sont générées, lorsque la politique MAC est de grande taille, et que la politique PIGA contient un grand nombre de propriétés. C'est notamment le cas lorsque celles-ci traitent des flux indirects, car ces flux peuvent correspondre à beaucoup de chemins dans les graphes associés. Lorsque la détection est activée, les performances du système sont diminuées. Le second problème est que, suivant les caractéristiques de la machine qui calcule les signatures, celles-ci seront plus ou moins limitées en longueur. Ainsi, les comportements malicieux composés de nombreuses interactions ne seront pas représentés dans la base de signatures. Ils ne pourront donc pas être détectés.

Durant cette thèse, nous nous sommes concentrés sur le premier problème. Cependant, une des méthodes développées permet également d'augmenter la longueur maximum des signatures générées. Dans ce chapitre, nous présenterons, dans la section 3.2, la première des deux méthodes développées afin de réduire le nombre de signatures générées. Cette méthode est une application de la décomposition modulaire. Nous verrons le principe général de la méthode, puis deux algorithmes permettant l'utilisation de la décomposition modulaire pour la génération de signatures, ainsi que leurs preuves. Enfin, nous verrons la gestion d'un cas particulier : les boucles sur les modules.

Dans la section 3.3, nous décrirons les expérimentations mises en place pour évaluer les performances de l'application de la décomposition modulaire. Nous l'étudierons d'abord sur un graphe exemple, puis sur des graphes réels de PIGA.

Dans la section 3.4, nous décrirons la seconde méthode développée : le suppression par inclusion. Nous présenterons d'abord le principe de cette méthode, puis nous évaluerons son efficacité.

Enfin, dans la section 3.5, nous conclurons ce chapitre par une discussion sur les limites des méthodes développées.

Les résultats de cette section sont partiellement présentés dans [BBC12].

3.2 Application de la décomposition modulaire

Dans cette partie, nous généralisons l'approche intuitive présentée dans la section 1.6 où la notion de module apparaît naturellement.

3.2.1 Utilisation de la décomposition modulaire non-orientée

Nous avons vu dans la section 1.2, que les graphes utilisés par PIGA pour le calcul des signatures à partir de propriétés de confidentialité sont des graphes orientés. Néanmoins, nous utilisons la décomposition modulaire non-orientée sur ces graphes pour réduire le nombre de signatures et cela pour plusieurs raisons.

Premièrement, les graphes d'interactions et de flux sont tous deux utilisés pour générer les signatures. Or, s'ils possèdent les mêmes sommets, des arcs peuvent être différents. L'application de la décomposition modulaire orientée pourrait générer deux décompositions différentes. Les graphes quotients 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 meilleure compression. Enfin, nous pouvons ré-orienter le graphe quotient. Ceci peut entraîner la génération de signatures incorrectes mais permet une plus grande compression. Malgré tout, les signatures surnuméraires ne sont pas gênantes, dans le contexte qui nous concerne, car elles ne peuvent pas être générées lors de la détection. En effet, tous les arcs produits par la ré-orientation représentent des actions non permises par le système MAC sur lequel se base PIGA. Les graphes ainsi obtenus reflètent une surestimation des attaques.

Afin de mesurer les différences entre le graphe quotient obtenu par notre méthode et celui obtenu par application de la décomposition modulaire orienté, nous définissons la notion d'arc *propre*.

Définition 23 (Arc propre)
*Soit $G = (V, A)$ un graphe orienté et $Q = (M, A')$ un graphe quotient de G. Un arc $(m_1, m_2) \in A'$ est **propre** ssi*

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

À partir de cette définition, on définit également la notion de graphe propre et de signature propre.

Définition 24 (Graphe propre)
*Un graphe quotient est **propre** ssi tous ses arcs sont **propres**.*

Définition 25 (Signature propre)
*Une signature est **propre** ssi elle contient uniquement des arcs **propres**.*

Sur les figures 3.1 et 3.2, on peut voir qu'en appliquant la décomposition modulaire orientée sur le graphe d'interactions de la figure 1.3 et sur le graphe de flux de la figure 1.4, nous obtenons deux graphes quotients n'ayant pas la même structure. En effet, sur le graphe quotient d'interactions, un module regroupe `admin_t` et `user_d`, or ce module n'est pas présent sur le graphe quotient de flux.

FIGURE 3.1 – Graphe quotient d'interaction obtenu par application de la décomposition modulaire non-orientée

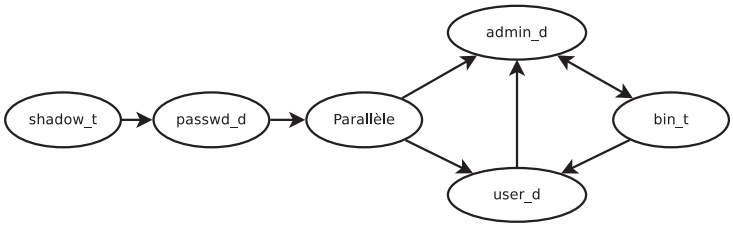


FIGURE 3.2 – Graphe quotient de flux obtenu par application de la décomposition modulaire orientée

3.2.2 Principe général

Dans cette section, nous présentons la méthodologie utilisée pour générer les signatures compressées. Chaque étape du processus sera accompagnée d'un paragraphe illustrant la méthode sur un exemple. Afin de générer des signatures compressées, il faut obtenir les graphes quotients orientés des graphes d'interactions et de flux. L'entrée de notre algorithme est le graphe de flux $Gf = (V, Af)$ et le graphe d'interactions $Gi = (V, Ai)$.

Pour pouvoir utiliser la décomposition modulaire, nous considérons le graphe non-orienté $Gf' = (V, Ef)$ obtenu en symétrisant Gf. Ce graphe est appelé graphe de flux symétrisé. Ensuite, nous calculons la décomposition modulaire sur Gf' pour obtenir l'arbre de décomposition modulaire $MT_{Gf'}$. À partir de cet arbre, nous obtenons $\mathcal{P}$ la partition modulaire maximal de Gf'. Enfin nous calculons, grâce à la partition, $Gi_{\mathcal{P}}$, le graphe quotient d'interaction, et $Gf_{\mathcal{P}}$, le graphe quotient de flux, à partir desquels nous calculons les signatures.

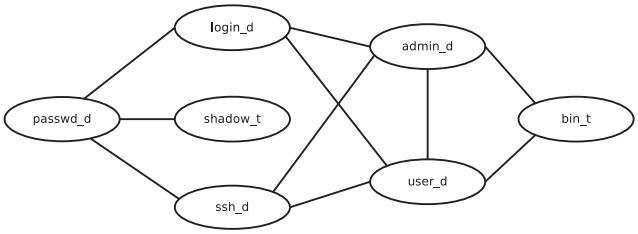


FIGURE 3.3 – Le graphe de flux symétrisé

Pour illustrer le processus, nous allons utiliser le graphe d'interactions simplifié présenté dans la section 1.2 sur la figure 1.3, ainsi que le graphe de flux dérivé de celui-ci (figure 1.4). En symétrisant ce graphe de flux, nous obtenons le graphe de la figure 3.3.

Une fois le graphe symétrisé $Gf' = (V, Ef)$ obtenu, nous pouvons lui appliquer la décomposition modulaire non-orientée afin de récupérer l'arbre de décomposition modulaire $MT_{Gf'}$, puis la partition $\mathcal{P}$.

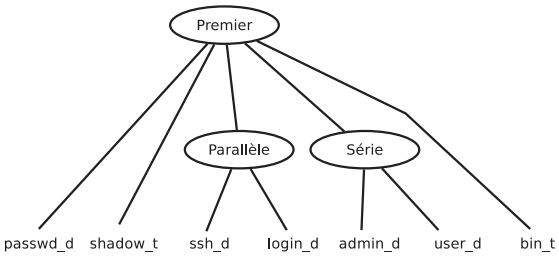


FIGURE 3.4 – L'arbre de décomposition modulaire généré à partir du graphe symétrisé

L'arbre de décomposition modulaire obtenu à partir du graphe 3.3 est représenté par la figure 3.4. La partition $\mathcal{P}$ obtenue est donc : $\mathcal{P} = \{\{\texttt{passwd\_d}\}, \{\texttt{shadow\_t}\}, \{\texttt{ssh\_d}, \texttt{login\_d}\}, \{\texttt{admin\_d}, \texttt{user\_d}\}, \{\texttt{bin\_t}\}\}$.

A partir de cette partition, nous générons les graphes quotients orientés $Gi_{\mathcal{P}}$ et $Gf_{\mathcal{P}}$. Pour cela, nous utilisons l'algorithme 3 qui prend en entrée le graphe d'origine (de flux ou d'interactions) et la partition $\mathcal{P}$.

Algorithme 3 Génération du graphe quotient orienté

Entrée: $G = (V, A)$ le graphe d'origine et $\mathcal{P}$ une partition de V
Sortie: $G' = (\mathcal{P}, A')$ le graphe quotient orienté $G_{\mathcal{P}}$

```
A' ← ∅
pour tout arc (x, y) ∈ A faire
    Soit S1 ∈ P tel que x ∈ S1
    Soit S2 ∈ P tel que y ∈ S2
    si S1 ≠ S2 alors
        si (S1, S2) ∉ A' alors
            A' ← A' ∪ {(S1, S2)}
            label(S1,S2) ← label(x,y)
        sinon
            label(S1,S2) ← label(S1,S2) ∪ label(x,y)
```

L'algorithme 3 fonctionne de la manière suivante pour générer les graphes quotients. Tout d'abord, l'ensemble des sommets du graphe quotient correspond à la partition passée en paramètre. Ensuite, pour chaque arc (x, y) du graphe d'origine, nous créons un arc entre l'ensemble de $\mathcal{P}$ contenant x et celui contenant y, si ceux-ci sont différents et s'il n'existe pas déjà d'arc entre ces deux ensembles. Pour le graphe d'interactions, il faut également gérer les labels présents sur les arcs. Pour cela, lors de la création de l'arc entre le module contenant x et celui contenant y, nous conservons le label de l'arc (x, y) du graphe d'interactions d'origine. Si un arc existe déjà, nous ajoutons le label de (x, y) si celui-ci n'est pas déjà présent.

En appliquant cet algorithme avec le graphe d'interactions Gi de la figure 1.3 et le graphe de flux Gf de la figure 1.4, nous obtenons les graphes quotients $Gi_{\mathcal{P}}$ (figure 3.5) et $Gf_{\mathcal{P}}$ (figure 3.6). La partition $\mathcal{P}$ étant de cardinalité 5, les graphes quotients générés ont 5 sommets. Il y a deux différences entre les deux graphes quotients. Premièrement, l'arc (`passwd_d`,`shadow_t`) du graphe quotient d'interactions

est inversé sur le graphe quotient de flux. Ceci s'explique par le fait que cet arc correspond à une lecture. Le flux d'informations part donc de `shadow_t` vers `passwd_d`. Deuxièmement, l'arc (`Series`,`bin_t`) du graphe quotient d'interactions devient un arc à double sens sur le graphe quotient de flux. L'arc correspondant à une lecture, une écriture ou une exécution, il peut y avoir transfert d'informations dans les deux sens. En effet, le flux d'informations part de `Series` vers `bin_t` dans le cas d'une écriture. Dans le cas d'une exécution ou d'une lecture, le flux d'informations part de `bin_t` vers `Series`.

FIGURE 3.5 – Graphe quotient d'interactions

FIGURE 3.6 – Graphe quotient de flux

3.2.3 Extraction d'une paire source/destination

Dans la section 1.3, 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 et la destination peuvent être contenues dans un module. Pour éviter cela, les extrémités des chemins doivent donc être considérées comme des singletons de la partition utilisée pour générer le graphe quotient. Cela se traduit par le fait que les extrémités doivent être des fils directs de la racine sur l'arbre de décomposition modulaire.

Pour cela, nous pourrions supprimer le module entier, et considérer tous les sommets qu'il contient comme des fils directs de la racine. Cependant, nous voulons conserver un maximum de modules afin que la réduction du graphe reste la meilleure possible. Dans ce but, nous proposons deux algorithmes permettant de conserver un maximum de modules non-triviaux. Le premier permet d'obtenir des modules de plus grande taille, mais ne peut pas être utilisé sur tous les graphes. Le second possède une meilleure complexité en temps que le premier, et peut être utilisé sur n'importe quel graphe. Cependant les modules obtenus peuvent être de plus petite taille. Lors des expérimentations présentées dans la section 3.3, nous avons utilisé le second algorithme.

Extraction des sommets par modification du graphe

L'algorithme 4 modifie le graphe afin que les sommets à « extraire » de l'arbre de décomposition modulaire ne puissent pas appartenir à un module non-trivial, tandis que l'algorithme 5 modifie directement l'arbre de décomposition modulaire en extrayant les sommets des modules auxquels ils appartiennent.

L'algorithme 4 gère le problème de manière globale en recalculant entièrement une décomposition modulaire sur le graphe modifié suivant. Nous ajoutons deux

sommets au graphe ($marqueur1$ et $marqueur2$), puis nous ajoutons un arc reliant $marqueur1$ au premier sommet à extraire, un second arc reliant $marqueur2$ au second sommet à extraire et un troisième arc reliant $marqueur1$ à $marqueur2$ (figure. 3.7). Pour pouvoir utiliser cet algorithme, nous devons être sûrs que les marqueurs ne seront pas dans un module. Le troisième arc ajouté permet de réduire à deux, le nombre de cas où les marqueurs appartiennent à un module. Le premier cas apparaît lorsque un des sommets à extraire est uniquement voisin avec le second sommet à extraire sur le graphe d'origine. Dans ce cas, le premier sommet formera un module avec le marqueur lié au second. Le second apparaît lorsque le graphe n'est pas connexe, les marqueurs apparaissent alors dans un module parallèle comprenant toute la composante connexe.

Algorithme 4 Extraction des sommets par modification du graphe

Entrée: $G = (V, E)$ un graphe non orienté, n et m deux sommets de G
Sortie: T' Un arbre de décomposition de G tel que n et m sont des module triviaux sous la racine

1: Soit $G' = (V', A')$ tel que
2: $V' \leftarrow V \cup \{marqueur1, marqueur2\}$
3: $E' \leftarrow E \cup \{(marqueur1, n), (marqueur2, m), (marqueur1, marqueur2)\}$
4: $T' \leftarrow DecompositionModulaire(G')$
5: $Supprimer_{T'}(marqueur1, marqueur2)$

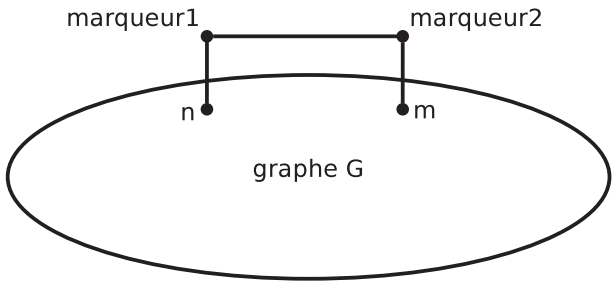


FIGURE 3.7 – Ajout des marqueurs au graphe

La proposition 1 montre le cadre de l'utilisation de cet algorithme et, le théorème 3, sa complexité. Le théorème 2 prouve l'efficacité de l'algorithme.

Proposition 1 *Soit $G = (V, E)$ un graphe connexe et, n et m deux sommets de G tels que n (resp. m) a au moins un voisin différent de m (resp. n). À la fin de l'étape 4 de l'algorithme 3 l'arbre T' est tel que :*

1. *$marqueur1$ et $marqueur2$ sont des modules triviaux situés sous la racine ;*
2. *n et m sont des modules triviaux situés sous la racine ;*
3. *tous les modules non triviaux définis dans T' sont des modules de G.*

Preuve : Cette preuve se décompose en plusieurs parties correspondant aux différents points de la proposition.

1. Supposons que $marqueur1$ et $marqueur2$ appartiennent à un module non trivial. Deux cas peuvent alors se produire :
   - Si $marqueur1$ appartient à un module M :
     - Soit $marqueur2$ appartient à M, alors n et m appartiennent également à M, car $marqueur1$ et $marqueur2$ n'ont pas le même comportement vis à vis de n et de m. Tous les sommets appartenant à M ont le même comportement vis-à-vis du reste du graphe. Or, comme $marqueur1$ et $marqueur2$ n'ont pas de voisin autre qu'eux-mêmes, n ou m, alors le voisinage de M est vide. Le graphe a donc au moins deux composantes connexes : M et le reste du graphe. Ceci est impossible car le graphe est connexe.
     - Soit $marqueur2$ n'appartient pas à M, alors tous les sommets appartenant à M sont voisins de $marqueur2$. Ainsi, seuls $marqueur1$ et m peuvent appartenir à M. Comme M est un module non trivial, il possède au moins deux sommets. Donc m appartient à M. Par conséquent, $marqueur1$ et m ont le même comportement vis-à-vis du reste du graphe, donc ils ont uniquement n et $marqueur2$ comme voisins. Ceci est impossible car m possède au moins un voisin différent de n.
   - Le même raisonnement peut s'appliquer à $marqueur2$. Dans ce cas, on considère m au lieu de n.
2. Supposons que les sommets n et m appartiennent à un module non trivial.
   On considère ici le cas de n, celui de m est symétrique.
   Soit n appartient à un module M. Comme $marqueur1$ ne peut pas appartenir à M et que n est voisin de $marqueur1$, tous les sommets de M sont voisins de $marqueur1$. Seuls n et $marqueur2$ sont voisins de $marqueur1$. Or, $marqueur2$ n'appartient pas à un module non trivial, donc il ne peut pas appartenir à M. Comme un module non trivial possède au moins deux sommets, M ne peut pas être un module non trivial.
3. Montrons maintenant que les modules définis par T' sont des modules du graphe initial G. Un sous-ensemble M de V est module de G si :

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

   Soit M' un module non trivial de G'
   - Par définition M' est un sous-ensemble de V' et nous avons prouvé que $marqueur1$ et $marqueur2$ n'appartiennent pas à M' car il est non trivial. V' étant égal à $V \cup \{marqueur1, marqueur2\}$, M' est un sous-ensemble de V.
   - Par définition :

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

     or V est un sous-ensemble de V' donc :

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

or $E' = E \cup \{(marqueur1, n), (marqueur2, m), (marqueur1, marqueur2)\}$, et $marqueur1$ et $marqueur2$ n'appartiennent ni à V ni à M' donc :

$$\forall x \in V \setminus M', \forall y \in M', (x, y) \in E' \Rightarrow (x, y) \in E$$
$$\forall x \in V \setminus M', \forall y \in M', (x, y) \notin E' \Rightarrow (x, y) \notin E$$

donc :

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

M' est un module de G. Donc tous les modules non triviaux définis dans T' sont des modules de G

■

Conséquence 1 *Les deux sommets marqués sont directement rattachés à la racine de l'arbre de décomposition. L'étape 5 ne modifie pas la structure globale de T' ; elle élimine simplement deux feuilles sous la racine.*

Théorème 2 *Soit $G = (V, E)$ un graphe connexe et, n et m deux sommets de G tels que n (resp. m) a au moins un voisin différent de m (resp. n).*

L'algorithme 4 retourne T' un arbre de décomposition de G dans lequel :

1. *n et m sont des modules triviaux situés sous la racine ;*
2. *tous les modules définis dans T' sont des modules de G.*

Preuve : Le théorème 2 est une conséquence directe de la proposition 1. ■

Pour analyser la complexité en temps de cet algorithme, nous utiliserons les notations suivantes :

– n représente le nombre de sommets du graphe G.
– m représente le nombre d'arêtes du graphe G.

L'ajout de deux sommets et de trois arêtes au graphe G s'effectue en temps constant. Dans la section 2.3, nous avons vu que l'algorithme de décomposition modulaire non-orienté s'effectue en $O(n+m)$. La complexité en temps de cet algorithme est donc en $O(n + m)$.

Théorème 3 *L'algorithme 4 se termine en $O(n + m)$*

Extraction d'un sommet par modification de l'arbre de décomposition modulaire

L'algorithme 4 recalcule une décomposition modulaire à partir du graphe initial. Il possède quelques restrictions qui ne sont pas contraignantes dans les cas concrets. Dans l'algorithme 5, nous cherchons une solution qui mette à profit l'arbre de décomposition modulaire déjà obtenu. Il peut être qualifié d'algorithme local.

Afin d'éviter la gestion de cas particuliers, nous avons utilisé l'algorithme 5 pour extraire les extrémités d'une signature de l'arbre de décomposition modulaire. Cet

Algorithme 5 Extraction d'un sommet par modification de l'arbre de décomposition modulaire

Entrée: n le sommet à extraire, $T = (V, A)$ l'arbre de décomposition modulaire
Sortie: $T' = (V', A')$ l'arbre avec le sommet extrait

1: $V' \leftarrow V$
2: $A' \leftarrow A$
3: **pour tout** $m \in chemin_T(n)$ **faire**
4: $\quad A' \leftarrow A' \setminus \{(Parent_T(m), m)\}$
5: $\quad$ **si** $m \neq Racine_{T'}$ **et** $m \neq n$ **et** $nombreFils_T(m) <= 2$ **ou** m *est Premier* **alors**
6: $\quad\quad$ **pour tout** $o \in enfants_T(m)$ **faire**
7: $\quad\quad\quad A' \leftarrow A' \setminus \{(m, o)\}$
8: $\quad\quad\quad A' \leftarrow A' \cup \{(Racine_{T'}, o)\}$
9: $\quad\quad V' \leftarrow V' \setminus \{m\}$
10: $\quad$ **sinon**
11: $\quad\quad A' \leftarrow A' \cup \{(Racine_{T'}, m)\}$

algorithme extrait un seul sommet. Afin d'extraire les deux extrémités, il faut donc l'utiliser deux fois consécutivement . Il fonctionne de la manière suivante. Tout d'abord, nous récupérons le chemin entre la racine de l'arbre et le sommet à extraire, tous les sommets intermédiaires sont des modules. L'algorithme parcourt alors ce chemin en partant de la racine et, pour chaque module, il supprime l'arête reliant le module courant à son parent, puis, suivant les caractéristiques du module, celui-ci sera supprimé ou uniquement modifié. Si le module possède exactement deux fils ou s'il est premier, alors le module est supprimé et on ajoute des arêtes entre la racine et tous ses fils. Sinon, on ajoute une arête entre la racine et le module.

Théorème 4 *Soit $G = (V, E)$ un graphe connexe.*
L'algorithme 5 retourne T' un arbre de décomposition de G tel que :

1. *n est un module trivial situé sous la racine ;*
2. *tous les modules définis dans T' sont des modules de G.*

Preuve :

1. L'étape 11 de l'algorithme relie directement le sommet à extraire à la racine. Donc n est un module trivial situé sous la racine.
2. Un module dans T' est obtenu à partir des modules de T. Soit M' un module non trivial dans T', il est défini à partir du module M de T, soit comme une copie de M, soit comme un sous-ensemble de M
   - Supposons que M n'est pas sur le chemin entre la racine et n dans T. Le module n'est pas modifié par l'algorithme. M' est donc un module de G
   - Supposons que M est sur le chemin entre la racine et n dans T.
     - M' ne peut pas être premier car l'algorithme éclate tous les modules premiers.
     - Si M' est un module non premier, alors M est un module dans T, donc :

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

Or M' est un sous-ensemble de M, donc :

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

– Si M' est un parallèle, alors M est un parallèle, donc : $\forall x \in M, \forall y \in M, (x,y) \notin E$
Par conséquent :

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

M' est donc un module de G.
– Si M' est un série, alors M est un série, donc : $\forall x \in M, \forall y \in M, (x,y) \in E$
Par conséquent :

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

M' est donc un module de G.

■

Pour analyser la complexité en temps de cet algorithme, nous utiliserons les notations suivantes :
– n représente le nombre de sommets du graphe G.
– m représente le nombre d'arêtes du graphe G.

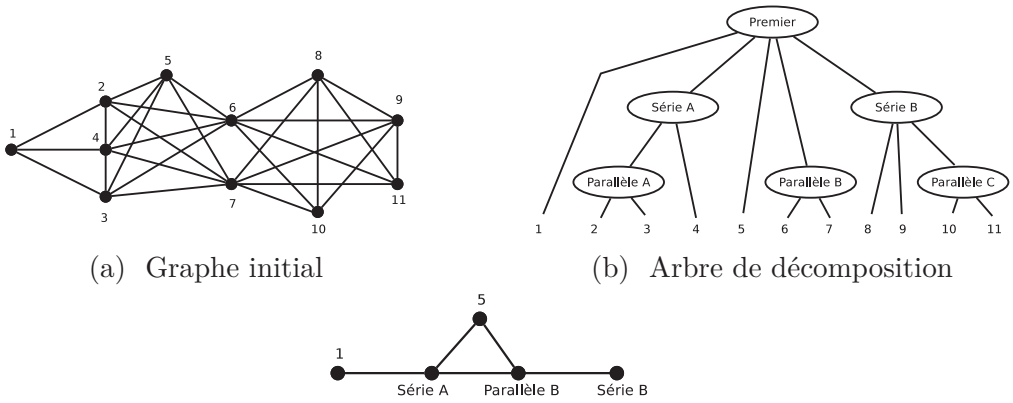
Cet algorithme parcourt le chemin entre la racine de l'arbre et le sommet à extraire. Pour chacun des sommets de ce chemin, il parcourt l'ensemble de ses enfants. Dans le pire des cas, chaque sommet de l'arbre est parcouru deux fois. Le nombre de sommets de l'arbre est au maximum $2n - 1$ car il possède n feuilles et tous les sommets intérieurs sont de degré minimum 2. La complexité en temps de cette algorithme est donc, au pire des cas, en $O(n)$.

Théorème 5 *L'algorithme 5 termine en $O(n)$.*

3.2.4 Applications des algorithmes

Pour illustrer les deux méthodes, nous les appliquons au graphe de la figure 3.8a. Ce graphe non-orienté contient 11 sommets et 29 arêtes. Son arbre de décomposition modulaire est représenté par la figure 3.8b. À partir de cet arbre, nous générons le graphe quotient (figure 3.8c) qui contient les sommets du premier niveau de l'arbre. Sur ce graphe, il est impossible de calculer les chemins entre les sommets 2 et 8.

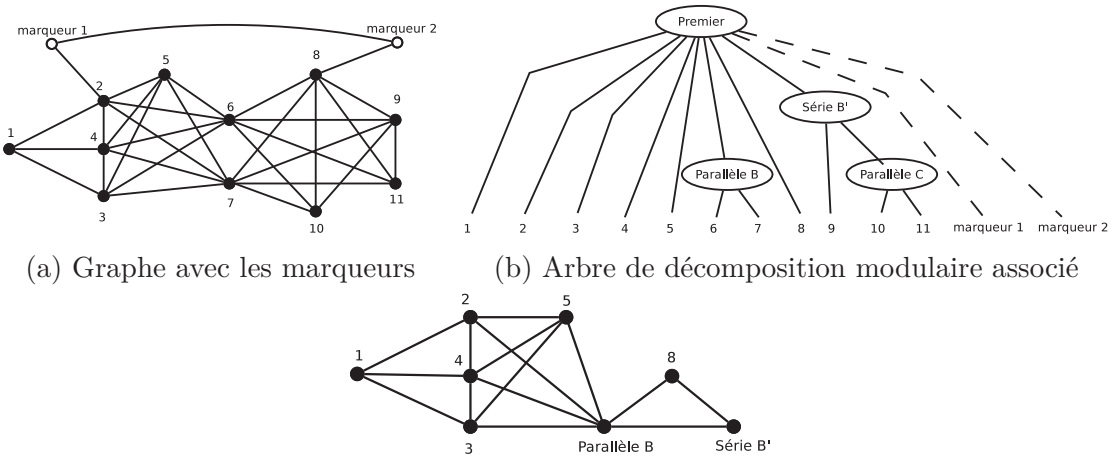
En appliquant l'algorithme 4 à ce graphe afin de faire apparaître les sommets 2 et 8 sur le graphe quotient, nous obtenons le graphe modifié représenté par la



(a) Graphe initial (b) Arbre de décomposition

(c) Graphe quotient

FIGURE 3.8 – Graphe exemple



(a) Graphe avec les marqueurs (b) Arbre de décomposition modulaire associé

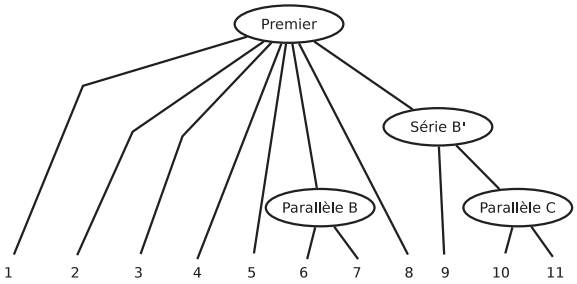
(c) Graphe quotient associé

FIGURE 3.9 – Extraction des sommets 2 et 8 en utilisant l'algorithme 4

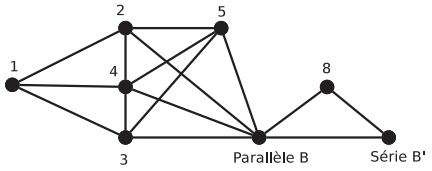
figure 3.9a. Par application de la décomposition modulaire, nous obtenons l'arbre de la figure 3.9b, puis le graphe quotient de la figure 3.9c.

En appliquant l'algorithme 5 successivement aux sommets 2 et 8, nous calculons l'arbre de la figure 3.10a et en déduisons le graphe quotient de la figure 3.10b. Pour cet exemple, on peut voir que l'application de l'un ou l'autre des algorithmes donne le même résultat. Cependant, on peut avoir des résultats différents. Cela est mis en évidence lors de l'étude du graphe de processus de la section 3.3. En effet, en calculant l'arbre de composition de ce graphe avec les contextes `system_u:system_r:sshd_t` et `user_u:user_r:user_t` extraits, on obtient deux arbres différents suivant l'algorithme utilisé. Nous présenterons les différences dans la section 3.3.

La figure 3.11 schématise le génération de signatures utilisant la décomposition modulaire.



(a) Arbre de décomposition modulaire modifié



(b) Graphe quotient associé

FIGURE 3.10 – Extraction des sommets 2 et 8 en utilisant l'algorithme 5

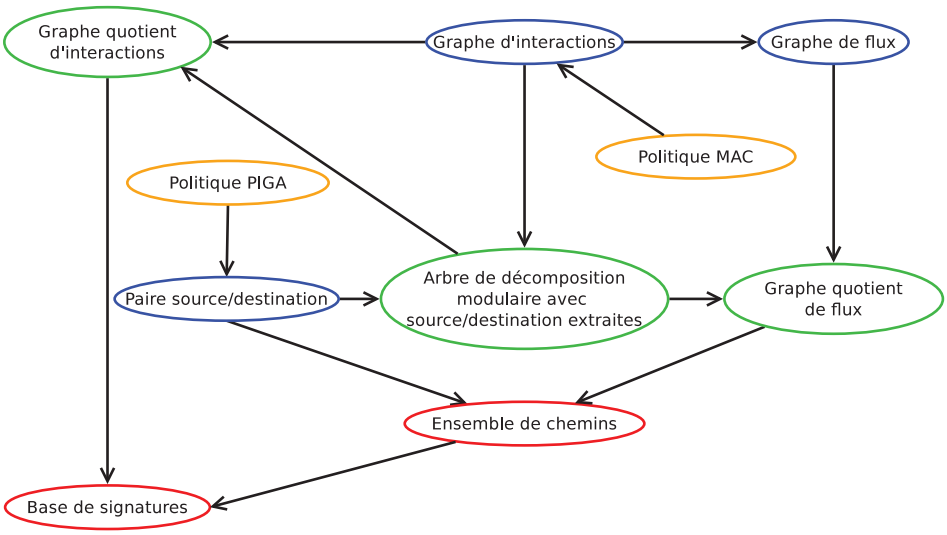


FIGURE 3.11 – Fonctionnement de la génération de signatures utilisant la décomposition modulaire

3.2.5 Gestion des boucles sur module

Dans la section 1.3, nous avons vu que le mécanisme original de génération de signatures n'autorise pas les boucles. Avec l'utilisation de la décomposition modulaire se pose la question de l'autorisation des boucles sur modules. En effet, sur cet exemple, on peut voir qu'une signature originale sans boucle peut se traduire par une signature modulaire avec une boucle.

$$1 \rightarrow 2 \rightarrow 6 \rightarrow 3 \rightarrow 5$$

$$1 \rightarrow \textit{Série } A \rightarrow \textit{Parallèle } B \rightarrow \textit{Série } A \rightarrow 5$$

Nous ne pouvons pas autoriser les boucles sur module sans restriction, car cela engendrerait un nombre infini de signatures. Nous avons donc envisagé deux solutions. Le première consiste à ne pas autoriser les boucles et à gérer les cas comme celui de l'exemple lors de la détection. La deuxième solution est d'autoriser k passages par un module de taille k. Au-delà de k passages, on est obligé de passer deux

fois par le même sommet ce qui correspond à une signature avec boucle dans le système original.

Cette deuxième méthode nécessite une gestion fine et complexe lors de la détection, car il faut se souvenir des différents éléments déjà visités, et enlève un peu d'intérêt à la compression. Elle ne sera évaluée que par rapport au taux de compression.

3.2.6 Compression théorique

Dans cette section, nous évaluons le gain théorique de notre approche. Dans un premier temps, nous donnons le principe général du calcul dans le cas d'une signature *sig* avec un seul module M.

$$s \Rightarrow a \rightarrow M \rightarrow b \Rightarrow d$$

Une signature compressée propre *sig*, contenant un seul module M, représente toutes les signatures non compressées qui sont construites de la manière suivante :

$$s \Rightarrow a \rightarrow x \Rightarrow y \rightarrow b \Rightarrow d$$

tel que :
- $x, y \in M$;
- $s \Rightarrow a$ est un chemin simple reliant s à a ne traversant pas M ;
- $b \Rightarrow d$ est un chemin simple reliant b à d ne traversant pas M ;
- $x \Rightarrow y$ est un chemin simple reliant x à y dans G potentiellement de longueur nulle. Par ailleurs, le chemin $x \Rightarrow y$ n'intersecte pas les chemins $s \Rightarrow a$ et $b \Rightarrow d$.

Étant donnée une signature compressée *sig*, la seule partie variable est donc la partie "interne" au module. Une signature modulaire représente autant de signatures non-modulaires qu'il existe de chemins simples entre deux éléments du module n'intersectant pas le reste de la signature.

D'un point de vue pratique, ceci est difficile à mettre en œuvre. Nous listons, pour la suite, les cas où une borne inférieure peut être obtenue en appliquant le principe précédent, mais en contraignant les chemins entre deux sommets d'un module à rester dans le module. Nous appelons *pur* un module dont tous les fils dans l'arbre sont des feuilles.

Lemme 5 *Une signature compressée propre s contenant un module M de taille k représente au moins $f(k)$ signatures non modulaires, telle que $f(k)$ est la somme des deux quantités décrites ci-dessous :*
- *k, la taille du module ;*
- *le nombre de chemins entre deux éléments distincts appartenant à M, en ne considérant que les chemin à l'intérieur du module M.*

Preuve : Soit s une signature compressée contenant un module, c'est-à-dire, $s = c_1, ..., c_i, M, c_{i+1}, ...c_l$, où M est un module de G tel que $M = \{c'_1, ...c'_k\}$. Tous les contextes c_i et c'_i sont distincts, sinon il y aurait une boucle dans la signature. À

partir de cette signature, nous pouvons dériver des signatures du graphe original de deux manières. Premièrement, nous pouvons remplacer M par un des éléments qui le compose. Les signatures ainsi générées sont de la même longueur que s. Deuxièmement, nous pouvons remplacer M par tout chemin simple entre deux éléments distincts de M. Les signatures générées de cette manière sont plus longues que s. ■

Conséquence 2 *Une signature* propre s *contenant un module parallèle* pur M *de taille* k *représente au moins* k *signatures sur le graphe original.*

Conséquence 3 *Une signature* propre s *contenant un module série* pur M *de taille* k *représente au moins* N *signatures sur le graphe original.*

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

Preuve : Dans ce cas les, sommets appartenant à M forment une clique. Ainsi, le nombre de chemins simples entre deux éléments de M est d'au moins le nombre de chemins à l'intérieur de la clique, chemins de longueur 0 inclus. Définir un chemin simple de longueur i consiste à choisir le premier élément parmi k, puis le deuxième parmi $(k-1)$ et ainsi de suite. Donc le nombre de chemins simples de longueur i est $\prod_{j=0}^{i}(k-j)$. En additionnant tous ces termes, nous obtenons le résultat voulu. ■

Nous nous sommes penchés sur le cas où la signature ne contient qu'un seul module. Dans le cas où il y a plusieurs modules dans la signature, le calcul général devient plus complexe. En effet, si on reprend la logique précédente, on obtient pour une signature contenant n modules :

$$sig = s \Rightarrow i \rightarrow M_1 \rightarrow j_1 \Rightarrow k_1 \ldots j_{n-1} \Rightarrow k_{n-1} \rightarrow M_n \rightarrow l \Rightarrow d$$

Les signatures non-modulaires représentées par sig sont de la forme :

$$s \Rightarrow i \rightarrow x_1 \Rightarrow y_1 \rightarrow j_1 \Rightarrow k_1 \ldots j_{n-1} \Rightarrow k_{n-1} \rightarrow x_n \Rightarrow y_n \rightarrow l \Rightarrow d$$

telles que :
- $\forall i \in [1, n] : x_i, y_i \in M_i$;
- $s \Rightarrow i$ est un chemin simple reliant s à i ne traversant aucun module ;
- $l \Rightarrow d$ est un chemin simple reliant l à d ne traversant aucun module ;
- $j_i \Rightarrow k_i$ est un chemin simple reliant j_i à k_i ne traversant aucun module ;
- $x_i \Rightarrow y_i$ est un chemin simple reliant x_i à y_i dans G potentiellement de longueur nulle. Par ailleurs, le chemin $x_i \Rightarrow y_i$ n'intersecte pas les chemins $s \Rightarrow i$ et $l \Rightarrow d$ ni aucun $j_j \Rightarrow k_j$ et aucun autre $x_j \Rightarrow y_j$.

En utilisant le lemme 5, une première borne inférieure, dans le cas où il existe plusieurs modules dans une signature, consiste à faire le produit des bornes inférieures de chaque module.

Les compressions théoriques calculées dans cette section ne sont que des bornes inférieures. Les calculs effectués ne tiennent pas compte de la possibilité d'effectuer des boucles sur un module, car le nombre exact de chemins entre deux sommets d'un module n'est pas calculable sans connaître exactement le graphe. Les gains réels ont donc une forte probabilité d'être supérieurs aux gains théoriques.

3.2.7 Équivalence des systèmes de génération

Dans cette section, nous montrons que, pour une politique de sécurité donnée, les deux systèmes de génération de signatures produisent des bases représentant les mêmes comportements.

Définition 26 (Équivalence des système de génération de signatures) *Deux systèmes de génération de signatures sont équivalents si, pour une même entrée, les bases qu'elles génèrent sont équivalentes.*

Théorème 6 *Le système de génération de signatures de PIGA est équivalent à celui utilisant la décomposition modulaire.*

Preuve : On note $\mathbb{B}_P$ la base de PIGA et $\mathbb{B}_M$ la base générée en utilisant la décomposition modulaire. Cette preuve se décompose en deux parties :

1. Une signature présente dans $\mathbb{B}_P$ est représentée dans $\mathbb{B}_M$:

   Soit $S = s, x_1, ..., x_n, d$ une signature de $\mathbb{B}_P$ de source s et de destination d. S est chemin dans le graphe de flux du système G. On définit $p(S)$ comme la projection de chaque sommet $x_i \in S$ sur le module maximal M_i auquel il appartient dans $MT_G(s, d)$.
   - Soit $\forall i, M_i = \{x_i\}$ alors la signature n'est pas compressée. Elle apparaît telle quelle dans $Quot_G$. Donc $S \in \mathbb{B}_M$.
   - Soit $\forall i \neq j, M_i \neq M_j$ alors la signature est compressée mais ne comporte pas de boucle sur module. Donc $p(S) \in \mathbb{B}_M$
   - Soit $\exists i \neq j$, tel que $M_i = M_j$ alors $p(S) = s, ..., M_i, ..., M_i, ..., d$. On peut alors supprimer la partie du chemin entre les deux occurrences de M_i. Ce processus peut être itéré jusqu'à ce que tous les modules soient différents. La signature ainsi obtenue appartient à $\mathbb{B}_M$ car le chemin correspondant ne contient pas de boucle.

2. Une signature non présente dans $\mathbb{B}_P$ n'est pas représentée dans $\mathbb{B}_M$:

   Soit S une signature de $\mathbb{B}_M$. On définit $p^{-1}(S) = \{S' \in G$ tel que $S = p(S')\}$.

   Supposons qu'il existe une signature $S \notin \mathbb{B}_P$ représentée par la signature $S' \in \mathbb{B}_M$. Alors $S \in p^{-1}(S')$ et $S \notin \mathbb{B}_P$. S est donc un chemin dans le graphe G. Or $\mathbb{B}_P$ et $\mathbb{B}_M$ ont les mêmes couples source/destination donc $S \in \mathbb{B}_P$.

■

On peut faire plusieurs remarques à partir de cette preuve. Dans le cas d'un graphe quotient non propre, la base de signature modulaire peut représenter des comportements n'étant pas représentés dans la base de PIGA. Cependant, ces comportements seront bloqués par le système MAC et n'apparaîtront pas dans les traces analysées par PIGA. Il est également possible qu'un comportement soit représenté par plusieurs signatures modulaires, notamment si des boucles sur module se chevauchent.

3.3 Expérimentation

Les expérimentations ont été réalisées en Java, en utilisant le code original de PIGA. Les nouvelles classes développées lors de cette thèse utilisent la librairie Jung [1] pour visualiser les graphes ainsi que le programme appliquant la décomposition modulaire développé en C par Fabien De Mongolfier [2]. Ce programme a été adapté au format des graphes utilisés par PIGA.

3.3.1 Processus d'expérimentation

Afin de tester l'efficacité de notre méthode de compression de signatures, nous calculons, pour différents graphes, les signatures pour plusieurs couples source/destination. Comme évaluation de l'efficacité de la méthode, nous utilisons deux mesures :

– Compression globale :

$$\rho_s = 1 - \frac{nombre\ de\ signatures\ compressées}{nombre\ de\ signatures\ non\text{-}compressées}$$

ρ_s indique le pourcentage de compression des signatures. Ainsi, si on obtient $\rho = 95\%$, cela signifie que l'on a obtenu 20 fois moins de signatures compressées que de signatures non-compressées.

– Compression mémoire :

$$\rho_m = 1 - \frac{taille\ de\ la\ base\ compressé}{taille\ de\ la\ base\ non\text{-}compressé}$$

ρ_m indique le pourcentage de compression de la base de signatures.

3.3.2 Expérimentation sur un graphe exemple

Tout d'abord, nous avons testé notre méthode sur le graphe de la figure 3.8a. Ce graphe, tiré de [HP10], n'a aucune application en sécurité mais contient des modules séries et parallèles. De plus, l'arbre de décomposition modulaire possède plusieurs niveaux. Ce graphe est composé de 11 sommets et 29 arêtes. Dans ce cas, nous travaillons sur le graphe non-orienté, ce qui revient à considérer le graphe orienté symétrique correspondant. 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. Le taux de compression est calculé deux fois pour chaque paire, une fois sans boucle autorisée sur les modules et une fois avec. 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.

Les résultats sont présentés dans la table 3.1. Dans le cas des signatures compressées sans boucle, nous calculons 1436 signatures contre 103411 avec le mécanisme d'origine, ce qui correspond une compression globale de 98,6%. C'est-à-dire qu'une

1. La librairie Jung est accessible sur `http://jung.sourceforge.net/`
2. Le programme de décomposition modulaire de F. De Montgolfier est disponible sur : `http://www.liafa.jussieu.fr/~fm/algos/index.html`

	Nb sig	ρ_s	Min ρ_s	Max ρ_s
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 3.1 – Taux de compression pour le graphe exemple (figure 3.8a)

signature modulaire sans boucle compresse en moyenne 72 signatures classiques. Les paires source/destination obtenant la moins bonne compression sont les paires 2/6, 2/7, 3/6 et 3/7. La compression obtenue est de 91,3% et le graphe quotient, sur lequel les signatures ont été calculées, est composé de 8 sommets et 36 arêtes. Les paires source/destination obtenant la meilleure compression sont les paires 1/8 et 1/9. La compression obtenue est de 99,9% et le graphe quotient, sur lequel les signatures ont été calculées, est composé de 6 sommets et 14 arcs.

Dans le cas des signatures compressées avec boucle, nous calculons 5780 signatures, ce qui correspond une compression globale de 94,4%. C'est-à-dire qu'une signature modulaire avec boucle compresse en moyenne 18 signatures classiques. Les paires source/destination obtenant la moins bonne compression sont les paires 2/10, 2/11, 3/10 et 3/11. La compression obtenue est de 81,7% et le graphe quotient, sur lequel les signatures ont été calculées, est composé de 9 sommets et 34 arcs. La paire source/destination obtenant la meilleure compression est la paire 1/5. La compression obtenue est de 99,7% et le graphe quotient, sur lequel les signatures ont été calculées, est composé de 5 sommets et 10 arcs.

Sur ces expérimentations, on peut voir que la compression sans boucle produit environ quatre fois moins de signatures que la compression avec boucle.

3.3.3 Expérimentation sur des graphes réels

L'expérimentation sur le graphe exemple a montré qu'il est possible d'obtenir une bonne compression des chemins en utilisant la décomposition modulaire. Cependant, cette méthode est très dépendante de la forme du graphe. Cette section présente les résultats obtenus en utilisant la décomposition modulaire sur des graphes utilisés par PIGA. Premièrement, nous nous intéressons aux graphes de flux utilisés pour la génération des signatures liées aux propriétés de confidentialité. En second, nous appliquons notre méthode sur un graphe de processus afin de tester les possibilités d'implémentation de notre méthode à d'autres propriétés.

Graphes de flux

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 411 sommets et 438 arcs, représente les flux d'informations possibles sur une passerelle sous Gentoo utilisée lors d'un test de performance de PIGA-IDS sur un pot de miel [BRCTZ09]. L'objectif de ce pot de miel est de comprendre le comportement des utilisateurs considérés comme attaquants. L'arbre de décomposition obtenu possède une structure particulière : il est de hauteur 3 et tous ses modules, sauf un, sont des parallèles. De plus, sa racine

est un parallèle contenant un module premier et 297 sommets. La figure 3.12 représente le sous-arbre du module premier, le sommet rose représente ce module et les sommets verts sont des modules parallèles. Ces modules parallèles peuvent refléter une logique sémantique. Par exemple, un parallèle contient `system_u:object_r:shadow_t`, `user_u:object_r:shadow_t` et `root:object_r:shadow_t`. Dans ce cas, la logique sémantique vient du fait que ces contextes représentent tous le fichier `/etc/shadow`. Le graphe de flux a été généré avec un mapping de 42 et nous avons utilisé l'algorithme 5 pour extraire les extrémités de l'arbre, car le graphe n'est pas connexe.

Nous calculons le nombre de signatures générées pour les 6 paires source/destination présentes dans la table 3.2. Le graphe quotient obtenu le plus grand possède 346 sommets et 198 arcs. La première paire (p1) représente tous les moyens de changement de mot de passe de l'utilisateur, tandis que la paire p4 représente toutes les tentatives de l'utilisateur pour obtenir de l'information contenue dans le fichier `shadow`. Le contexte `user_u:user_r:user_t` est utilisé pour toutes les paires et seuls quatre contextes différents sont utilisés comme source ou destination. Parmi ces contextes, seul `system_u:object_r:shadow_t` appartient à un module. Ainsi, pour les paires p2, p3, p5 et p6, nous utilisons le graphe quotient non modifié pour calculer les signatures. Pour les deux paires restantes, le graphe quotient utilisé est composé de 27 sommets et 91 arêtes. On peut noter que le graphe quotient n'est pas *propre*, la ré-orientation du graphe quotient génère 10 arcs supplémentaires.

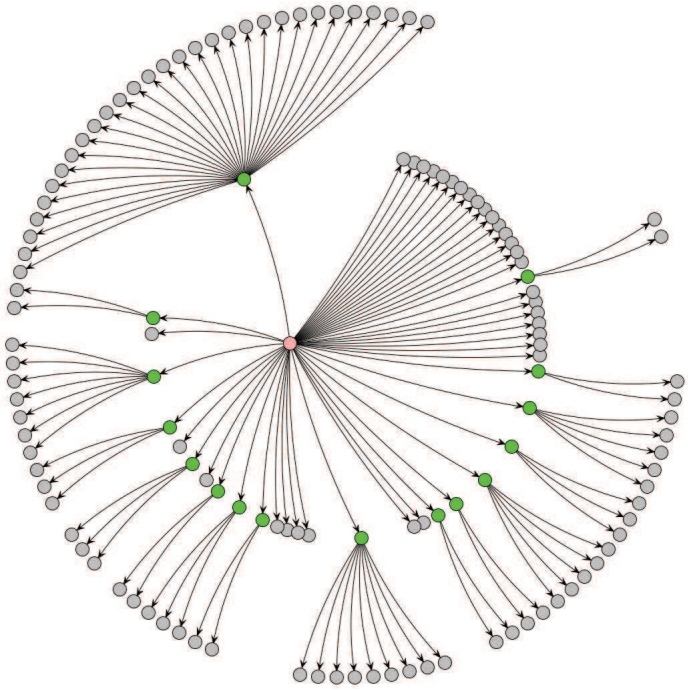


FIGURE 3.12 – Décomposition du module premier du graphe de flux

La table 3.3 présente le nombre de signatures non-compressées et compressées pour chaque paire ainsi que le taux de compression associé. Cette table indique que le nombre de signatures modulaires est de 6870 contre 152513 signatures non modulaires, soit un taux de compression globale de 95,5%. Une signature modulaire

```
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 3.2 – Paires source/destination analysées

	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_s	0%	0%	96.6%	95.3%	95.3%	96.6%	95.5%

TABLE 3.3 – Taux de compression pour le graphe de flux

compresse donc, en moyenne, 22 signatures non-modulaires. Cette expérimentation montre, par ailleurs, qu'une signature modulaire compresse entre 2 et 717 signatures non-modulaires et comporte entre 1 et 3 modules. Certaines signatures, de longueur 4 et ne contenant qu'un seul module, peuvent compresser plusieurs centaines de signatures non-modulaires.

La figure 3.13 montre le nombre cumulé de signatures calculées en fonction de leur longueur. Durant cette expérimentation, toutes les signatures sont calculées, que ce soit avec ou sans utilisation de la décomposition modulaire. Sur cette figure, on peut voir que les deux courbes ont le même comportement et que la longueur maximum des signatures est de 15 avec compression et de 17 sans compression. Certaines signatures non-modulaires plus longues sont donc compressées par des signatures modulaires de longueur inférieure.

Nous avons étudié un autre graphe, mais un mapping de 42 rend ce graphe quasi-identique au graphe utilisé précédemment. On obtient alors le même nombre de signatures pour le graphe quotient. Nous avons donc descendu le mapping à 35 et calculé les signatures non-compressées et compressées allant de `system_u:object_r:etc_t` vers `user_u:user_r:user_t`. La table 3.4 représente les caractéristiques des graphes ainsi que le nombre de signatures non-compressées et compressées. Nous avons limité la longueur (k) à 6 pour les signatures non-compressées et à 8 pour les signatures compressées, car la machine de calcul ne dispose pas d'assez de mémoire pour aller au-delà. Il est difficile de donner un taux de compression pour cette expérience car l'ensemble des signatures n'a pas été généré.

	Sommets	Arêtes	Sign. non-compressées	Sign. compressées	
			k=6	k=6	k=8
Graphe 1	411	729	50060	16694	376681
Graphe 2	428	732	44435	12862	261181

TABLE 3.4 – Résultat du calcul de signatures avec un mapping de 35

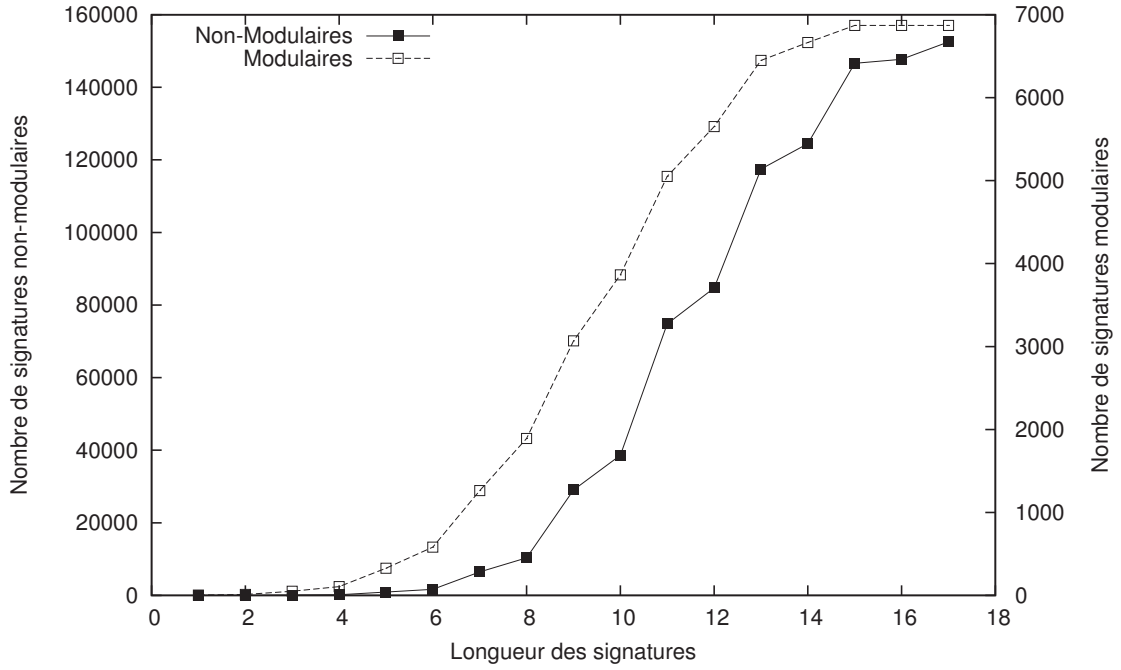


FIGURE 3.13 – Distribution cumulative des signatures en fonction de leur longueur

Graphe de processus

À la section 1.2, nous avons vu que le graphe d'interactions peut être dérivé en différents graphes. Le graphe de processus est obtenu en ne gardant que les arêtes étiquetées par une opération élémentaire de classe `process`. Le graphe de processus, que nous étudions ici, est composé 381 sommets et 21074 arêtes soit une densité de 15%. L'application de la décomposition modulaire génère un arbre possédant un module série comme racine. La table 3.5 montre que l'arbre de décomposition modulaire obtenu est plus profond et plus complexe que celui du graphe de flux. Au niveau 1, les feuilles sont des contextes de la forme `*:sysadm_r:sysadm_t` qui sont en rapport avec l'administrateur du système. Dans la mesure où celui-ci a tous les droits, ces contextes sont connectés à tous les autres. Le graphe quotient est très simple : il correspond à un graphe complet à 5 sommets. On peut noter que le module premier du niveau 4 contient 108 des 112 sommets du niveau 5. Pour la plupart des modules, les contextes qu'ils contiennent ont des similitudes. Par exemple, tous les contextes d'`iptable` appartiennent au même module. Il en va de même pour les contextes de `mozilla`, de `ssh`, de `gcc_config`, etc.

Nous avons calculé le nombre des signatures entre `system_u:system_r:sshd_t` et `user_u:user_r:user_t` sur le graphe de processus en utilisant la décomposition modulaire. Ce calcul vise à définir tous les moyens de connexion à un utilisateur via `ssh`. Lors de cette expérimentation, nous avons extrait les sommets des extrémités en utilisant les deux algorithmes de la section 3.2. Les arbres de décomposition obtenus sont différents, donc les graphes quotients le sont également. Le graphe quotient généré suite à l'extraction par marquage de sommets comporte 53 modules séries, 13 modules parallèles et 46 contextes soient 112 sommets et 902 arêtes dont 89 ne sont pas *propres*. Le graphe quotient généré par modification de l'arbre est composé

	Série	Parallèle	Premier	Feuilles
Racine	1	0	0	0
Niveau 1	0	1	0	4
Niveau 2	1	0	0	1
Niveau 3	0	1	0	17
Niveau 4	1	0	1	3
Niveau 5	51	12	0	49
Niveau 6	8	0	0	270
Niveau 7	0	0	0	37

TABLE 3.5 – Composition des niveaux de l'arbre de décomposition modulaire du graphe de processus

de 52 modules séries, 13 modules parallèles et 50 contextes soient 115 sommets et 1515 arcs dont 92 ne sont pas *propres*.

Sur le graphe d'origine, il n'était pas possible de calculer toutes les signatures de longueur 4 à cause d'un manque de mémoire. Ceci est une conséquence directe de la recherche de chemin dans les modules séries. En effet, elle génère de nombreuses signatures différentes. Sur le graphe quotient, il n'est pas possible de calculer toutes les signatures de longueur 5. En limitant la longueur à 4, nous obtenons 3546 signatures. Afin de pouvoir donner un taux de compression, nous restreignons le calcul aux signatures de longueur 3. Nous obtenons ainsi 190 signatures modulaires contre 1571 signatures non-modulaires. Ceci correspond à un taux de compression de $\rho_s = 87.9\%$.

Le module premier du niveau 4 étant dans un module parallèle, les sommets qui le composent ne sont pas reliés au reste du graphe. Les signatures entre deux contextes appartenant à ce module premier ne peuvent donc pas passer par des contextes qui n'y appartiennent pas. Les contextes `system_u:system_r:sshd_t` et `user_u:user_r:user_t` appartenant au module premier, nous avons recalculé les signatures en prenant uniquement le sous-graphe quotient contenu dans le module premier. Dans ce cas, le graphe initial est composé de 350 sommets et 5930 arcs contre 108 sommets et 506 arcs pour le graphe quotient. Nous avons ainsi pu monter jusqu'à 10 de longueur pour les signatures modulaires et 4 de longueur pour les signatures non-modulaires. Si l'on se limite à 4 de longueur, on obtient seulement 96 signatures modulaires contre 12130 non-modulaires soit un ratio de 99%.

3.4 Simplification par inclusion de signatures

Dans ce qui précède, nous avons considéré les signatures pour chaque paire source/destination. Cependant, dans les cas réels, les propriétés de confidentialité mettent en jeu plusieurs paires de contextes potentiellement sensibles. Nous montrons comment prendre en compte cette spécificité dans cette section.

Afin de réduire le nombre de signatures générées par PIGA, nous avons utilisé une seconde méthode : la suppression par inclusion. Si une signature est incluse dans une autre, la première sera complétée avant la seconde. Or, lorsque PIGA est en mode IPS, la dernière interaction permettant la complétion d'une signature est

	Sans suppression par inclusion				Avec suppression par inclusion			
	Sign.	ρ_s	Taille	ρ_m	Sign.	ρ_s	Taille	ρ_m
Sans DM	152513	-	89,5Mo	-	30107	80,3%	24.2Mo	73%
Avec DM	6870	95,5%	3,4Mo	96,2%	2016	98,7%	1,3Mo	98,6%

TABLE 3.6 – Comparaison du nombre de signatures et de la taille de la base suivant les combinaisons de méthode utilisés

bloquée. Ainsi, la seconde signature ne pourra jamais être complétée car l'interaction finissant la première sera toujours bloquée. Le principe de cette méthode est donc de supprimer une signature s'il existe une autre signature qui est incluse dans la première. Cette méthode est utilisable avec ou sans la décomposition modulaire. Nous avons appliqué cette méthode lors de la génération de signatures du graphe de flux du pot de miel. La table 3.6 représente le nombre de signatures générées, le taux de compression ρ_s, la taille de la base de signatures générée et ρ_m. Ces résultats sont donnés pour chaque combinaison de méthodes possible : avec ou sans décomposition modulaire (DM) et avec ou sans suppression par inclusion.

On peut observer que l'application de la suppression par inclusion seule divise le nombre de signatures générées par 5 et la taille la base par un peu moins de 4. L'utilisation de cette méthode, en plus de la décomposition modulaire, permet de diviser par plus de 3 le nombre de signatures et par plus de 2 la taille de la base comparée à l'utilisation seule de la décomposition modulaire. Cette expérimentation montre que la suppression par inclusion réduit de manière importante le nombre de signatures générées, ainsi que la taille de la base.

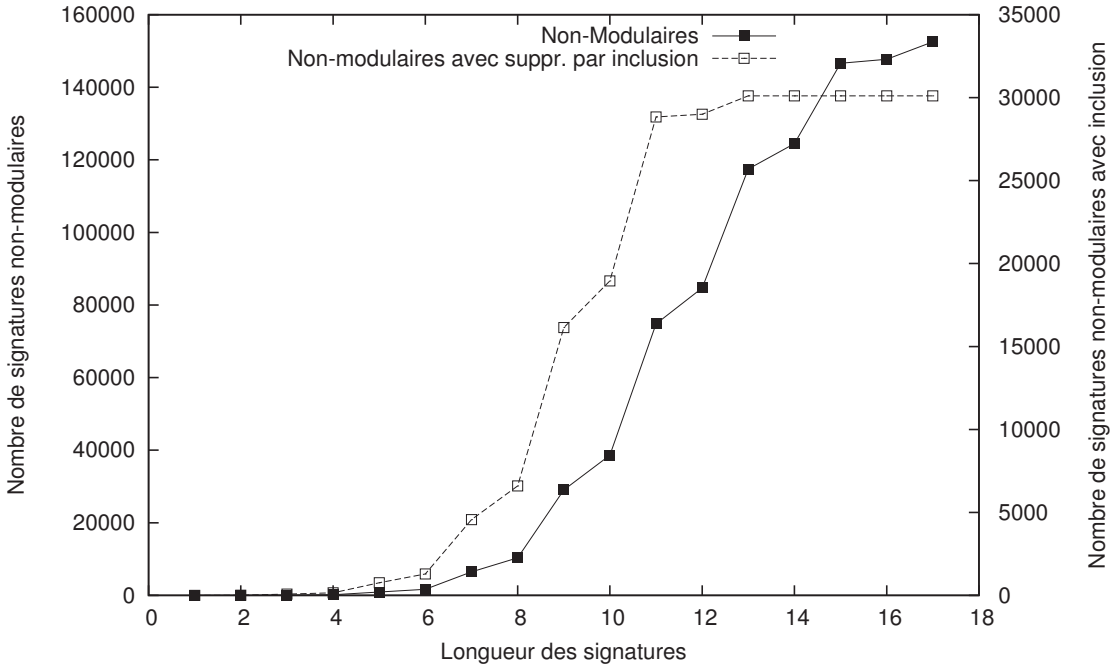


FIGURE 3.14 – Distribution cumulative des signatures en fonction de leur longueur

La figure 3.14 montre le nombre cumulé de signatures non-modulaires avec et

sans suppression par inclusion en fonction de leur longueur. On peut noter que les deux courbes ont un comportement très proche jusqu'à 10 de longueur. En effet, à partir de ce palier, on peut voir que le nombre de signatures non-modulaires avec suppression par inclusion n'augmente quasiment plus. À partir de la longueur 13, toutes les signatures ont été calculées.

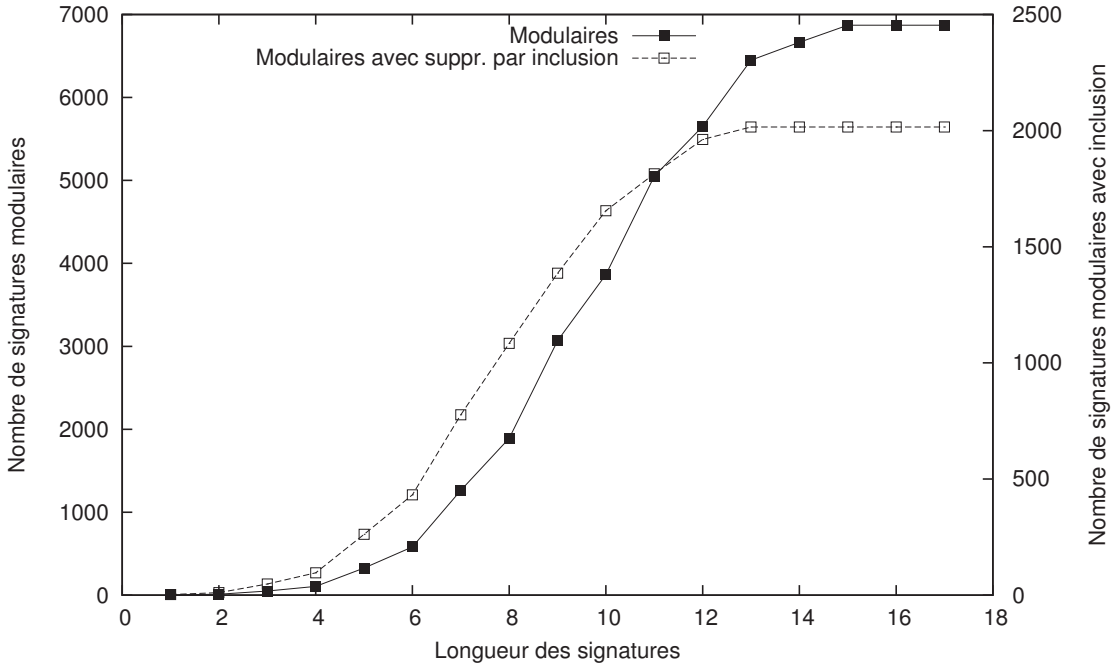


FIGURE 3.15 – Distribution cumulative des signatures en fonction de leur longueur

La figure 3.15 représente le nombre cumulé de signatures modulaires avec et sans suppression par inclusion en fonction de leur longueur. On peut noter que les deux courbes ont un comportement très proche. La principale différence est que l'augmentation des signatures modulaires avec suppression par inclusion diminue à partir de la longueur 10 contre 13 pour les signatures modulaires simples. En effet, à partir de ce palier, on peut voir que le nombre de signatures non-modulaires avec suppression par inclusion n'augmente quasiment plus. A partir de la longueur 13, toutes les signatures ont été calculées.

3.5 Limites d'utilisation

3.5.1 Application de la décomposition modulaire

L'application de la décomposition modulaire a pour effet de créer des boîtes noires dans les signatures. En effet, une fois une signature détectée, il est impossible de savoir, sans utiliser beaucoup de mémoire, comment les modules de cette signature ont été traversés. La gestion des boucles sur modules s'effectuant lors de la détection, le flux d'informations peut passer par de nombreux contextes entre le moment où il rentre dans un module et celui où il en sort. Ainsi, l'écriture de fichiers de log précis,

décrivant les différentes étapes de l'attaque qui a été détectée, est impossible si la décomposition modulaire a été appliquée pour générer les signatures.

3.5.2 Suppression par inclusion

La suppression de signatures par inclusion ne peut être utilisée que lorsque PIGA est en mode protection. En effet, en mode détection, les interactions terminant une signature n'étant pas bloquée, il est possible qu'une signature en incluant une seconde soit complétée.

3.6 Conclusion

Les expérimentations effectuées nous montrent que les taux de compression obtenus grâce aux méthodes développées sont bons, que ce soit pour le nombre de signatures ou pour la taille de la base. Pour 6 propriétés de confidentialité générant une base de 90Mo contenant plus de 150000 signatures sur le système original, on obtient, en combinant les deux méthodes, une base de 1,3Mo contenant un peu plus de 2000 signatures. De plus, l'application de la décomposition modulaire permet d'obtenir des signatures plus longues, soit par le calcul de chemins plus longs dans le graphe, soit en étant représentées par des signatures modulaires. L'expérience sur le graphe de processus montre que l'application de décomposition modulaire peut s'appliquer à d'autres propriétés qu'à la confidentialité.

Pour chaque propriété de confidentialité, le calcul de l'arbre de décomposition modulaire augmente le temps nécessaire au calcul des signatures. Cependant, lorsque toutes les signatures sont calculées en utilisant le système original, le temps de calcul est plus court en appliquant la décomposition modulaire car le graphe utilisé est plus petit.

Il faut maintenant adapter le mécanisme de détection aux signatures modulaires afin d'évaluer les performances en terme de temps ainsi que la quantité de mémoire nécessaire à la gestion des modules lors de la détection. La forme de l'arbre de décomposition, composé de nombreux modules de petite taille, semble positive du point de vue de la mémoire nécessaire, comparée à des modules de grande tailles mais moins nombreux.

Chapitre 4

Modification du système de détection

4.1 Introduction

Le système de détection actuel est fait pour fonctionner avec des signatures non-modulaires. Il peut donc être utilisé pour la base de signatures générée en appliquant uniquement la suppression par inclusion. En effet, cette base est un sous-ensemble cohérent de la base originale. Les signatures y appartenant possèdent donc la même forme que celles de la base originale et sont donc utilisables avec le système de détection actuel. Cependant, les signatures générées par application de la décomposition modulaire ne sont pas utilisables avec ce mécanisme de détection car elles possèdent des informations qui peuvent être traitées à un autre niveau : le module. Nous devons donc modifier celui-ci, afin qu'il puisse traiter les signatures résultant de l'application de la décomposition modulaire.

Dans la section 4.2, nous présenterons les nouveaux cas engendrés par l'utilisation de la décomposition modulaire qu'il faudra traiter lors de la détection. Ces cas sont dus à la présence d'un ou plusieurs modules dans une signature et au fait que ces modules soient consécutifs ou non.

Dans la section 4.3, nous décrirons les algorithmes de détection développés afin de gérer ces nouveaux cas. Nous analyserons leur complexité en temps et en espace. Enfin, nous définirons la notion d'équivalence entre deux systèmes de détection, puis nous prouverons que les algorithmes de détection développés sont équivalents à celui du système original. Cette équivalence s'entend en termes de détections de violations de propriétés de sécurité.

Dans la section 4.4, nous décrirons les expérimentations mises en place afin d'évaluer les performances en termes de temps d'exécution et de consommation de mémoire du nouvel algorithme de détection en fonction de la base de signatures choisie. Nous vérifierons également, sur un cas pratique, l'équivalence du nouveau système de détection avec celui d'origine.

Enfin, dans la section 4.6, nous conclurons ce chapitre par une discussion autour de la manière de gérer les exceptions dans le nouveau système de détection.

4.2 Nouveaux cas engendrés par les méthodes de compression

Les nouveaux cas engendrés par la compression des signatures sont uniquement liés à la décomposition modulaire. En effet, la suppression par inclusion élimine des signatures mais n'en modifie pas la forme. Son utilisation seule ne nécessite donc pas d'adaptation du mécanisme de détection. Le nouveau mécanisme doit pouvoir gérer les signatures modulaires. Dans cette section, nous verrons comment gérer les signatures contenant un module, en détaillant les différents cas induits par la présence de ce module. Nous verrons également que le traitement d'un module s'adapte à la gestion de plusieurs modules non-consécutifs.

La gestion des signatures possédant des modules consécutifs sera présentée dans la section 4.5. Elle n'est pas implémentée dans notre algorithme de détection. En effet, elle nécessite une analyse fine de tous les cas possibles, leur nombre augmentant avec le nombre de modules présents consécutivement. De plus, elle n'est pas nécessaire. En effet, en « cassant » certains modules d'une signature possédant des modules consécutifs, on obtient une signature sans module consécutif, qui peut donc être traitée par notre algorithme.

Le but de l'analyse présentée dans cette section, est de définir le moyen d'être sûr que lorsqu'un comportement malicieux est détecté, la signature levant l'alerte correspond bien à ce comportement.

4.2.1 Analyse des nouveaux cas

Dans cette section, nous traitons une propriété de confidentialité du contexte d envers de le contexte s, c'est-à-dire qu'il ne doit pas y avoir de flux d'informations de s vers d.

Pour les différents cas présentés ici, nous analyserons la signature suivante, correspondant à la propriété traitée :

$$S_1 = s \rightarrow 1 \rightarrow Module\ M \rightarrow d$$
$$M = \{x, y, z\}$$

Nous souhaitons déterminer les scénarios qui mèneront à la levée d'une alerte par cette signature et ceux qui le font pas. Dans ce dernier cas, si le flux d'informations présenté viole la propriété de confidentialité, nous identifions la signature détectant la violation.

Gestion d'un module

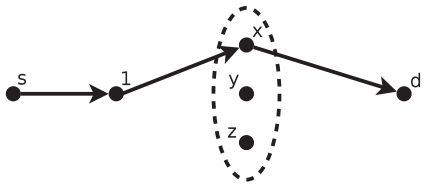
La traversée d'un module présent dans une signature, sans prendre en compte les boucles, est le cas le plus simple. Afin de vérifier si une trace complète une signature, il faut s'assurer que le flux d'informations va bien du sommet source au sommet destination.

Nous concentrons l'étude pour les flux qui traversent le module avec une ou deux interactions possibles avant et après le module. En effet, les autres cas généraux (signatures longues) peuvent être décomposés en utilisant une des situations suivantes.

Voici une liste des cas qu'il faut traiter pour la signature ci-dessus. Pour chaque cas, nous étudierons le flux d'informations d'une trace.

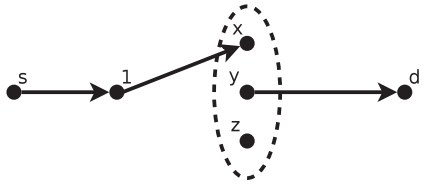
1. **Le cas classique :**

Trace 1
$s \to 1$
$1 \to x$
$x \to d$



Sur cette trace, on entre dans le module M par le sommet x, puis on sort par le même sommet. Le flux d'informations va bien du sommet s au sommet d et correspond à la signature. La propriété est violée et la signature S_1 doit être complétée.

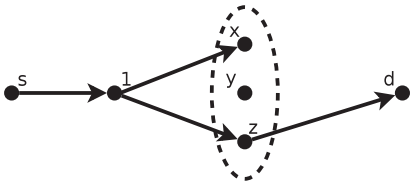
Trace 2
$s \to 1$
$1 \to x$
$y \to d$



Pour cette trace, on entre dans le module M par le sommet x mais on sort par le sommet y. Le flux d'informations ne passe donc pas du sommet s au sommet d. Ceci représente le seul cas de levée de fausse alerte. La propriété n'est pas violée, la signature S_1 ne doit pas être complétée.

2. **Le cas du backtrack :**

Trace 3
$s \to 1$
$1 \to x$
$1 \to z$
$z \to d$



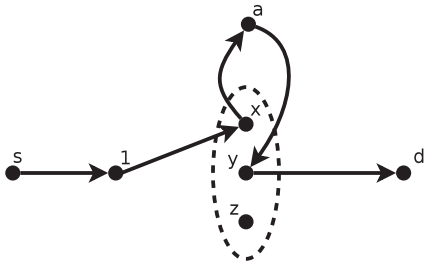
Sur cette trace, on entre dans le module M par x, puis par z et on sort par le sommet z. Ainsi, le flux d'informations va bien du sommet s vers le sommet d en passant par le sommet z, et correspond à la signature. La propriété est violée et la signature S_1 doit être complétée.

Gestion des boucles sur module

Dans la section 3.2, nous avons vu que, lors de la génération, nous n'autorisons pas les boucles sur les modules. Cependant, il est possible de passer plusieurs fois par un module, en passant par des sommets différents à chaque passage. Nous devons donc gérer les boucles sur module durant la détection.

1. **Le cas classique :**

Trace
$s \rightarrow 1$
$1 \rightarrow x$
$x \rightarrow a$
$a \rightarrow y$
$y \rightarrow d$



Le tableau ci-dessus présente une trace appliquée à la signature utilisée pour le cas précédent. Ici, on sort du module par un sommet différent de celui par lequel on est entré. Cependant, l'information a circulé du sommet d'entrée x au sommet de sortie y via le sommet a.

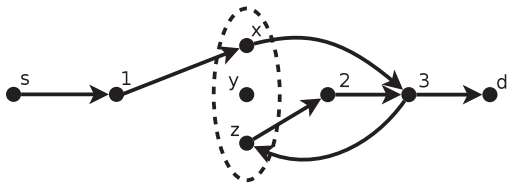
$$\begin{array}{ccccccccccc} s & \rightarrow & 1 & \rightarrow & x & \rightarrow & a & \rightarrow & y & \rightarrow & d \\ s & \rightarrow & 1 & \rightarrow & Module\ M & \rightarrow & a & \rightarrow & Module\ M & \rightarrow & d \end{array}$$

Si l'on remplace les sommets x et z par le module M dans la trace, on peut voir que la trace boucle sur le module. En supprimant la boucle de la trace, on obtient la signature voulue. Le flux d'informations va bien du sommet s au sommet d en respectant la signature. La propriété est violée et la signature S_1 doit être complétée.

2. **Le cas du cycle :**
Pour étudier le cas du cycle, nous considérons la signature, correspondant également à la propriété traitée, suivante :

$$S_2 = s \rightarrow 1 \rightarrow Module\ M \rightarrow 2 \rightarrow 3 \rightarrow d$$
$$M = \{x, y, z\}$$

Trace
$s \rightarrow 1$
$1 \rightarrow x$
$x \rightarrow 3$
$3 \rightarrow y$
$y \rightarrow 2$
$2 \rightarrow 3$
$3 \rightarrow d$



Ici, on effectue une boucle sur le module en passant par le sommet 3. Ce sommet apparaît déjà dans la signature. Voici le flux d'informations correspondant à la trace :

$$s \rightarrow 1 \rightarrow x \rightarrow 3 \rightarrow z \rightarrow 2 \rightarrow 3 \rightarrow d$$

Le flux d'informations va du sommet s au sommet d. La propriété est donc violée. Si on analyse ce flux, on se rend compte qu'il comporte un cycle sur le

sommet 3. Le système de génération de signature original ne peut pas générer cette signature. Si l'on supprime ce cycle, la signature non-modulaire obtenue est la suivante :

$$S_3 = s \quad \rightarrow \quad 1 \quad \rightarrow \quad x \quad \rightarrow \quad 3 \quad \rightarrow \quad d$$

La signature modulaire correspondante est donc :

$$S'_3 = s \quad \rightarrow \quad 1 \quad \rightarrow \quad Module\ M \quad \rightarrow \quad 3 \quad \rightarrow \quad d$$

Le flux d'informations va bien du sommet s au sommet d mais ne correspond pas à la signature S_2. La propriété est violée. Cependant, la signature S_2 ne doit pas être complétée. La détection de ce comportement malicieux est dévolue à la signature modulaire S'_3.

4.2.2 Gestion théorique

Pour illustrer les premiers cas de la gestion théoriques, nous considérons la signature S_1 définie dans la section précédente.

Gestion d'un module

1. **Le cas classique :**
   Grâce aux exemples de trace de la section précédente, on peut voir qu'il est nécessaire de mémoriser le sommet d'entrée d'un module, afin de s'assurer que le flux d'informations traverse le module.

	Trace 1	Sommet d'entrée	Atteint par le flux
1	$s \rightarrow 1$	–	$s, 1$
2	$1 \rightarrow x$	x	$s, 1, x$
3	$x \rightarrow d$	–	$s, 1, x, d$

   Pour la trace 1, le flux atteint le sommet d car lors de l'interaction $x \rightarrow d$, le sommet x est le sommet d'entrée du module M.

	Trace 2	Sommet d'entrée	Atteint par le flux
1	$s \rightarrow 1$	–	$s, 1$
2	$1 \rightarrow x$	x	$s, 1, x$
3	$y \rightarrow d$	x	$s, 1, x$

   Pour la trace 2, le flux n'atteint pas le sommet d car lors de l'interaction $y \rightarrow d$, le sommet y n'est pas le sommet d'entrée du module M.

2. **Le cas du backtrack :**
   Pour gérer le cas du backtrack, il est nécessaire de mémoriser le sommet précédant le module, nommé *prédécesseur*, afin de traiter les interactions qui entrent de nouveau dans celui-ci par un autre sommet.

	Trace 3	Prédécesseur	Sommet d'entrée	Atteint par le flux
1	$s \to 1$	–	–	$s, 1$
2	$1 \to x$	1	x	$s, 1, x$
3	$1 \to z$	1	x, z	$s, 1, x, z$
4	$z \to d$	–	–	$s, 1, x, z, d$

Pour cette trace, le sommet 1 est mémorisé comme prédécesseur de module et le sommet x et z sont tous les deux mémorisés comme sommet d'entrée. Ainsi le flux d'informations peut sortir du module par un de ces sommets.

Du point de vue de l'implémentation, il est donc nécessaire de mémoriser les sommets par lesquels on entre dans un module mais également le sommet précédant le module. Une fois que le flux d'informations a atteint le sommet suivant le module, les informations mémorisées ne sont plus nécessaires. Elles peuvent donc être supprimées de la mémoire.

Gestion des boucles sur module

1. **Le cas classique :**
   Afin de gérer les boucles sur modules, il faut être capable de déterminer, dans le module, les sommets atteints par le flux d'informations à partir de l'entrée. Pour cela, nous utilisons un système de contamination. Lors de la traversée d'un module, un sommet peut être contaminé de deux façons. Premièrement, si le flux induit d'une interaction a pour source le sommet précédant le module, et, pour destination, un sommet y appartenant, alors le sommet destination est contaminé. Deuxièmement, si le flux induit d'une interaction a pour source un sommet déjà contaminé, alors le sommet destination est contaminé. Il n'est cependant pas possible de contaminer un sommet présent dans la signature. Enfin, si le flux induit d'une interaction a pour source, un sommet contaminé appartenant au module, et, pour destination, le sommet suivant le module, alors on sort du module et on supprime de la mémoire la liste des sommets contaminés.

	Trace	Prédécesseur	Sommets contaminés
1	$s \to 1$	–	–
2	$1 \to x$	1	x
3	$1 \to y$	1	x, y
4	$z \to b$	1	x, y
5	$x \to a$	1	x, y, a
6	$a \to d$	1	x, y, a
7	$a \to z$	1	x, y, a, z
8	$z \to d$	–	–

La table ci-dessus présente l'évolution de la contamination sur le module M, pour une trace sur la signature présentée lors de la section précédente. On peut voir que le sommet x (resp. y) est contaminé lorsque l'on entre dans le module lors de l'interaction 2 (resp. 3). La quatrième interaction ne contamine pas b car z n'est pas contaminé, tandis que la cinquième contamine a à partir

de x qui est déjà contaminé. L'interaction 6 complète un flux d'informations de s vers d. Cependant, cette interaction ne finit pas la signature S_1, car, si a est contaminé, il n'appartient pas au module. Une autre signature lèvera une alerte pour cette violation. De plus, cette interaction ne contamine pas 3, car il apparaît dans la signature. Lors de la septième interaction, a contamine z. Enfin, la dernière interaction sort du module et complète la signature, car z est contaminé et appartient à M.

2. **Le cas du cycle :**

   Pour gérer le cas du cycle, il suffit, avant de contaminer un sommet, de vérifier qu'il n'apparaît pas dans la signature. La signature étudiée est la suivante :

$$S_2 = s \to 1 \to Module\ M \to 2 \to 3 \to d$$
$$M = \{x, y, z\}$$

	Trace	Prédécesseur	Sommets contaminés
1	$s \to 1$	–	–
2	$1 \to x$	1	x
3	$x \to 3$	1	x
4	$3 \to z$	1	x
5	$z \to 2$	1	x
6	$2 \to 3$	1	x
7	$3 \to d$	1	x

   Le sommet x est contaminé lors de l'entrée dans le module par le sommet 1. La troisième interaction ne contamine pas 3, car il apparaît dans la signature. La quatrième interaction ne contamine pas z, car 3 n'est pas contaminé. Les interactions restantes ne font pas progresser la signature car z n'est pas contaminé.

La gestion des boucles sur un module implique une augmentation de la mémoire nécessaire pour le système de détection, car celui-ci doit stocker les informations de contamination. Cependant, la mémoire utilisée est récupérée directement après être sortie du module. De plus, plus le nombre et la longueur des boucles possibles sur un module sont grands, plus la compression obtenue par la présence de ce module est importante.

Gestion des modules non-consécutifs

Les mécanismes de gestion des différents cas présentés précédemment peuvent être utilisés pour gérer la présence de modules non-consécutifs. En effet, il existe au moins un contexte simple après chaque module. La traversée d'un module est ainsi validée lorsque ce contexte simple est atteint. Chaque module est donc géré indépendamment des autres.

4.3 Mise en œuvre algorithmique

Dans cette section, nous étudierons les algorithmes de détection pour des signatures possédant un ou plusieurs modules non-consécutifs. Les boucles sur module ne sont gérées que par le second algorithme. La gestion des modules consécutifs n'est pas traitée et n'a pas été implémenté durant cette thèse. En effet, dans les cas concrets, elle ne s'avère pas nécessaire. Nous établirons ensuite la complexité en espace de ces algorithmes, ainsi que celle de l'algorithme de détection original. Enfin, nous prouverons l'équivalence, sous certaines conditions, des algorithmes de détection.

4.3.1 Algorithmes de détection

Pour rappel, la fonction $Pendante(S)$ pointe vers l'interaction pendante de la signature S. La fonction $Suivant(S)$ renvoie l'interaction suivant l'interaction pendante de S. La fonction $SourcePendante(S)$ (resp. $DestPendante(S)$) renvoie la source (resp. la destination) de l'interaction pendante de S. Enfin, la fonction $Alerte(S, I)$ crée une alerte de complétion de la signature S par l'interaction I.

L'algorithme 6 définit la fonction $Avancer(ALERTES, S, I)$, prenant en paramètres un ensemble d'alertes, une signature ainsi qu'une interaction. Cette fonction gère l'avancement ainsi que la complétion des signatures. En effet, celle-ci remplace l'interaction pendante par celle qui la suit dans la signature et lève une alerte si cette signature est complétée. Lorsque qu'une signature est complétée, la dernière interaction de celle-ci reste pendante. Cette signature est en attente d'une nouvelle complétion.

Algorithme 6 Fonction faisant avancer une signature : $Avancer(ALERTES, S, I)$

Entrée: $ALERTES$ un ensemble d'alertes levées, S une signature, I une interaction
Sortie: $ALERTES$ et S

1: **si** $Suivant(S) = \emptyset$ **alors**
2: $\quad ALERTES \leftarrow ALERTES \cup Alerte(S, I)$
3: **sinon**
4: $\quad Pendante(S) \leftarrow Suivant(S)$

L'algorithme 7 implémente la procédure de gestion d'un module décrite dans la section précédente. Comme le système de détection original, il prend en paramètres une trace, c'est-à-dire un ensemble ordonné d'interactions, et une base de signatures. La sortie de cet algorithme est également l'ensemble des alertes levées par la complétion de signatures lors de l'analyse de la trace. La fonction $Prédécesseur(S)$ pointe vers le contexte précédant le module actif de S. La fonction $Entrée(S)$ pointe vers l'ensemble des contextes par lesquels on est entré dans le module actif de S. Enfin, rappelons qu'on considère que $I \in S$, si l'interaction I est une étape de la signature S.

L'algorithme 7 permet de gérer le cas classique de traversée d'un seul module, ainsi que le cas du backtrack. Il va lire chaque interaction de la trace dans l'ordre,

et, pour chaque signature incluant cette interaction, tester plusieurs situations. Tout d'abord, on récupère Src, la source de l'interaction pendante de S, et $Dest$, sa destination (ligne 4). Si l'interaction en cours est l'interaction pendante de la signature, alors celle-ci progresse (lignes 5 et 6). Sinon, si x fait partie des sommets par lesquels on est entré dans le module actif, que x appartient à la source de l'interaction pendante, et que y en est la destination, alors on sort du module et on supprime de la mémoire les informations qui ont permis de la traverser (lignes 7 à 10). Sinon, si y appartient à la destination de l'interaction pendante, on teste si x en est la source (lignes 11 et 12). Si tel est le cas, on entre dans le module : le prédécesseur est défini à x et y est ajouté à la liste des sommets d'entrée (lignes 13 à 15). Sinon, si x est le prédécesseur du module en cours et si y appartient à celui-ci, alors on est entré une nouvelle fois dans le module. Le sommet y est donc ajouté à la liste des sommets d'entrée (lignes 16 et 17).

Algorithme 7 Algorithme de détection des signatures pouvant contenir des modules non-consécutifs

Entrée: $BASE$ un ensemble de signatures,
$TRACE$ un ensemble ordonné d'interactions
Sortie: $ALERTES$ l'ensemble des alertes levées par la complétion des signatures

```
1:  ALERTES ← ∅
2:  pour tout interaction I(x → y) ∈ TRACE faire
3:    pour tout signature S ∈ BASE telle que I' ∈ S faire
4:      Src ← SourcePendante(S), Dest ← DestPendante(S)
5:      si Pendante(S) = I alors
6:        Avancer(ALERTES, S, I)
7:      sinon si x ∈ Entrée(S) et x ∈ Src et y = Dest alors
8:        Avancer(ALERTES, S, I)
9:        Prédécesseur(S) ← ∅
10:       Entrée(S) ← ∅
11:     sinon si y ∈ Dest alors
12:       si x = Src alors
13:         Avancer(ALERTES, S, I)
14:         Prédécesseur(S) ← x
15:         Entrée(S) ← {y}
16:     sinon si Prédécesseur(S) = x et y ∈ Src alors
17:       Entrée(S) ← Entrée(S) ∪ {y}
```

L'algorithme 8 implémente la procédure de gestion des boucles sur module décrite dans la section précédente. Comme le système de détection original, il prend en paramètres une trace et une base de signatures, et a pour sortie l'ensemble des alertes levées par la complétion de signatures lors de l'analyse de la trace. La fonction $Contaminé(S)$ pointe vers l'ensemble des contextes contaminés lors de l'exploration du module actif de S.

L'algorithme 8 permet de gérer les boucles sur module ainsi que le cas du cycle.

Comme pour l'algorithme 7, il va lire chaque interaction de la trace dans l'ordre et tester plusieurs situations pour chaque signature. La seule évolution vient de la gestion des sommets contaminés (sommets d'entrée dans l'algorithme 7). Ici, un sommet devient contaminé quand on entre à nouveau dans le module, mais également lorsque la source de l'interaction en cours d'analyse est déjà contaminée (lignes 15 et 16).

Algorithme 8 Algorithme de détection des signatures gérant les boucles sur module

Entrée: $BASE$ un ensemble de signatures,
$TRACE$ un ensemble ordonné d'interaction
Sortie: $ALERTES$ l'ensemble des alertes levées par la complétion des signatures

1: $ALERTES \leftarrow \emptyset$
2: **pour tout** interaction $I(x \rightarrow y) \in TRACE$ **faire**
3: **pour tout** signature $S \in BASE$ telle que $I' \in S$ ou $x \in Contaminé(S)$ **faire**
4: $Src \leftarrow SourcePendante(S)$, $Dest \leftarrow DestPendante(S)$
5: **si** $Pendante(S) = I$ **alors**
6: $Avancer(ALERTES, S, I)$
7: **sinon si** $x \in Contaminé(S)$ et $x \in Src$ et $y = Dest$ **alors**
8: $Avancer(ALERTES, S, I)$
9: $Prédécesseur(S) \leftarrow \emptyset$
10: $Contaminé(S) \leftarrow \emptyset$
11: **sinon si** $y \neq Dest$ et $x = Src$ et $y \in Dest$ **alors**
12: $Avancer(ALERTES, S, I)$
13: $Prédécesseur(S) \leftarrow x$
14: $Contaminé(S) \leftarrow \{y\}$
15: **si** $x \in Contaminé(S)$ ou $(Prédécesseur(S) = x$ et $y \in Src)$ **alors**
16: $Contaminé(S) \leftarrow Contaminé(S) \cup \{y\}$

4.3.2 Analyse de complexité

Dans cette section, nous procéderons au calcul de complexité en temps et en espace, dans le pire des cas, pour l'algorithme de détection original (algorithme 2), celui gérant la présence de modules non-consécutifs (algorithme 7), et l'algorithme gérant les boucles sur module (algorithme 8). Pour cette analyse, nous utiliserons les notations suivantes :

- t représente la taille de la trace, c'est-à-dire le nombre de lignes qu'elle comporte ;
- s représente le nombre de signatures que la base comporte ;
- n représente le nombre de sommets dans le graphe d'interactions utilisé lors de la génération de la base de signatures.
- $\mathcal{S}$ représente la base de signatures.

Pour le calcul des complexités en espace, on considère que la trace est un flux et qu'il n'y a qu'une ligne à la fois en mémoire. Les alertes sont également un flux, elles ne sont pas conservées en mémoire. De plus, la fonction *Avancer*, décrite dans l'algorithme 6, s'effectue en temps constant.

Algorithme de détection original

Dans la section 1.4, nous avons vu que la complexité en temps de l'algorithme 2 est en $O(s.t)$ et que la complexité en espace est en $O(\mathcal{S})$.

Algorithme de détection gérant les modules non-consécutifs

L'algorithme de détection gérant les modules non-consécutifs (algorithme 7) parcourt toutes les signatures, pour chaque ligne de la trace. Puis, à chaque étape, pour chacune des signatures, on peut soit faire avancer la signature, soit ne rien faire. Si, lors de l'avancée, on entre dans un module, il faut ajouter le contexte par lequel on entre dans le module à la liste d'entrée. Cet ajout se fait en temps constant. Si on sort du module, il faut vérifier que le contexte de sortie appartient bien à la liste d'entrée. Cette vérification est faite en temps constant. La complexité en temps de l'algorithme 7 est donc en $O(s.t)$.

Lors de son déroulement, cet algorithme conserve en mémoire toutes les signatures, leur avancement ainsi qu'une liste d'entrée pour les signatures étant entrées dans un module. L'avancement de chaque signature prend une place fixe en mémoire. Au pire des cas, la liste d'entrée d'une signature peut contenir tous les sommets exceptés les sommets source et destination de la signature. De plus, il est nécessaire de stocker l'arbre de décomposition modulaire de taille $2n - 1$, au pire des cas. La complexité en espace de cet algorithme est donc, au pire des cas, $\mathcal{S} + s.n + 2n - 1$, soit une complexité en $O(\mathcal{S} + s.n)$.

Algorithme de détection gérant les boucles sur module

L'algorithme de détection gérant les modules non-consécutifs (algorithme 8) parcourt toutes les signatures, pour chaque ligne de la trace. Puis, à chaque étape, pour chacune des signatures, on peut soit faire avancer la signature, soit contaminer un contexte, soit ne rien faire. Si, lors de l'avancée, on entre dans un module, il faut ajouter le contexte par lequel on entre dans le module à la liste de contamination. Cet ajout se fait en temps constant. Si on sort du module, il faut vérifier que le contexte de sortie est contaminé. Cette vérification est faite en temps constant. La contamination est un ajout à la liste de contamination. Elle s'effectue donc en temps constant. La complexité en temps de l'algorithme 8 est donc en $O(s.t)$.

Lors de son déroulement, cet algorithme conserve en mémoire toutes les signatures, leur avancement ainsi qu'une liste de contamination pour les signatures étant entrées dans un module. L'avancement de chaque signature prenant une place fixe en mémoire. Au pire des cas, la liste de contamination d'une signature peut contenir tous les sommets exceptés les sommets source et destination de la signature. De plus, il est nécessaire de stocker l'arbre de décomposition modulaire de taille $2n - 1$, au pire des cas. La complexité en espace de cet algorithme est donc, au pire des cas, $\mathcal{S} + s.n + 2n - 1$, soit une complexité en $O(\mathcal{S} + s.n)$.

Lemme 6
Les algorithmes 7 et 8 ont une complexité en temps en $O(s.t)$ et une complexité en espace en $O(\mathcal{S} + s.n)$.

Les nouveaux algorithmes sont de même complexité en temps que l'algorithme original. Cependant, une base de signatures générée en utilisant la décomposition modulaire possède moins de signatures que la base équivalente générée par le système original. Les nouveaux algorithmes devraient donc avoir, pour une même trace et des bases de signatures équivalentes, un temps d'exécution inférieur à celui d'origine.

Complexité	Algo. original	Algo. 7	Algo. 8
En temps	$O(s.t)$	$O(s.t)$	$O(s.t)$
En espace	$O(\mathcal{S})$	$O(\mathcal{S} + s.n)$	$O(\mathcal{S} + s.n)$

Les nouveaux algorithmes consomment, en théorie une quantité de mémoire supérieure de $s.n$ par rapport à celle consommée par l'algorithme original. Cependant, une base de signatures générée en utilisant la décomposition modulaire est moins volumineuse que la base équivalente générée par le système original. De plus, toutes les signatures ne sont pas obligatoirement modulaires. Il est donc très probable que, dans la pratique, les nouveaux algorithmes consomment moins de mémoire que l'algorithme d'origine, pour une même trace et des bases de signatures équivalentes.

4.3.3 Équivalence des systèmes de détection

Dans cette section, nous montrons que les deux systèmes de détection ont la même puissance de détection. Cette validation suppose que les deux bases sont complètes c'est-à-dire qu'elles contiennent exhaustivement toutes signatures possibles.

Définition 27 (Équivalence des systèmes de détection)
On dit que deux systèmes de détection sont équivalents, pour des bases de comportements équivalentes, si, pour toute trace, toute levée d'alertes dans un des systèmes induit au moins une levée d'alerte dans l'autre système au même instant.

Théorème 7 *Le système de détection de PIGA est équivalent à celui défini par l'algorithme 8 dans le cas où aucune signature de la base modulaire ne comporte plusieurs modules consécutifs.*

Preuve : Cette preuve se décompose en deux parties :

1. *Un comportement détecté par le système de détection de PIGA est détecté par celui défini par l'algorithme 8 :*

   Soit S une signature détectée par PIGA et $S' = p(S)$ ($p(S)$ est défini section 3.2.7). Pour $S = s, x_1, ..., x_n, d$ nous avons donc $S' = s, M_1, ..., M_n, d$ où M_i est le module auquel appartient x_i.
   - Soit $\forall i, M_i = \{x_i\}$ alors la signature n'est pas compressée. Elle est alors gérée par l'algorithme 8 de la même manière que par PIGA. Elle est donc détectée.
   - Soit $\forall i \neq j, M_i \neq M_j$ alors la signature est compressée mais ne comporte pas de boucle sur module. La traversée des modules est alors gérée par la sauvegarde du sommet d'entrée.
   - Soit $\exists i \neq j$, tel que $M_i = M_j$ alors la signature comporte au moins une boucle sur module. Les boucles sur module sont alors gérées par le système de contamination.

2. *Un comportement détecté par le système de détection défini par l'algorithme 8 est détecté par celui de PIGA :*

   Par construction, chaque détection de l'algorithme 8 correspond à une signature. Il est possible pour chaque détection de retrouver la signature détectée par PIGA correspondante. Des exemples sont présentés dans la section 4.2.

■

La base de signature de PIGA regroupe tous les comportements malicieux autorisés par le système MAC et son système de détection détecte toutes les signatures de cette base. Ainsi, PIGA détecte tous les comportements malicieux autorisés par le système MAC. Notre système de détection étant équivalent à celui de PIGA, il détecte, également, tous les comportements malicieux autorisés par le système MAC.

4.4 Expérimentation

Les expérimentations présentées dans cette section ont été réalisées en Java, en utilisant le code original de PIGA, ainsi que de nouvelles classes développées lors cette thèse. Toutes les expérimentations ont été effectuées en mode rejeu de trace et les mesures de temps d'exécution et de consommation mémoire ont été faites directement sur la JVM.

4.4.1 Création d'un générateur de trace semi-aléatoire

Afin de tester l'efficacité du nouveau système de détection, nous avons développé un générateur de trace semi-aléatoire. Ce générateur prend, en entrée, un graphe d'interactions ainsi qu'une table de mapping. Il renvoie un fichier de traces d'un nombre fixé de lignes, chacune représentant une interaction possible dans le système et autorisée par le mapping donné.

Le générateur possède une liste de sommets activés. Au début, seul le sommet `system_u:system_r:initrc_t` est activé. À chaque étape de la génération, on choisit, aléatoirement, une arête du graphe représentant au moins une interaction autorisée par le mapping, et dont la source appartient à la liste des sommets activés. Ensuite, on sélectionne, aléatoirement, une interaction autorisée par le mapping parmi celles de l'arête choisie. Enfin, on ajoute le sommet destination de l'arête à la liste des sommets activés. On recommence ce processus jusqu'à avoir atteint le nombre de lignes souhaité.

4.4.2 Expérimentations sur un graphe d'interactions réel

Le nouveau système de détection implémente l'algorithme 8. Les modules consécutifs ne sont donc pas gérés. Néanmoins, les bases de signatures, que nous avons générées lors des expérimentations sur le nouveau système de génération de signatures (section 3.3), ne comportent pas de signatures avec plusieurs modules consécutifs.

Afin d'économiser de la mémoire, seul l'arbre de décomposition modulaire initial est utilisé lors de la détection. Ainsi, une des extrémités peut appartenir à un module

qui apparaît dans la signature. Pour gérer cela, lors de la traversée d'un module, on limite la contamination aux contextes qui ne sont pas présents dans la signature.

Cette expérimentation a plusieurs objectifs. Premièrement, nous voulons vérifier l'équivalence du nouveau système de détection avec le système original. Deuxièmement, nous souhaitons mesurer le temps nécessaire à l'analyse d'une trace, afin de s'assurer que le nouveau système de détection ne peut pas ralentir le système sur lequel PIGA est installé. Finalement, nous voulons comparer la consommation de mémoire du nouveau système avec celle du système orignal. Ce dernier point est important car il correspond à l'objectif de cette thèse.

Afin d'évaluer les performances du nouveau système de détection, nous l'exécutons sur plusieurs traces. Les trois mesures utilisées sont : le temps nécessaire pour analyser la trace, le nombre de comportements détectés et la mémoire maximale utilisée par le système de détection. Les traces sont créées grâce à notre générateur et font 100000 lignes.

Les simulations ont été effectués pour quatre bases de signatures :

1. La base générée par le système original ;
2. La base générée par application de la décomposition modulaire ;
3. La base générée par application de la suppression par inclusion ;
4. La base générée par application de la décomposition modulaire et de la suppression par inclusion.

Pour les bases 1 et 3, le système de détection original a été utilisé, tandis que, pour les bases 2 et 4, nous avons utilisé le système de détection implémentant l'algorithme 8.

Évaluation d'équivalence

Derrière la contrainte de l'équivalence des bases de signatures s'inscrit le souhait que l'IDS implémentant la décomposition modulaire détecte les mêmes attaques que le système original, sans lever de fausse alerte. Cependant, l'équivalence des bases de signatures n'est pas suffisant. Il est également nécessaire que les systèmes de détection soient équivalents. Dans la section précédente, nous avons prouvé que le système implémentant l'algorithme 8 est théoriquement équivalent au système de détection original. Cette expérimentation a pour but de vérifier cette équivalence dans la pratique. Pour cela, nous calculons le nombre d'alertes levées pour une même trace, pour chaque base de signature.

Le tableau ci-dessous représente le nombre d'alertes levées pendant l'analyse deux traces différentes de 100000 lignes sur les quatre bases de signatures.

	Nombre de signatures	Alertes levées	
		Trace 1	Trace 2
Système original	152513	786	820
Avec DM	6870	441	458
Avec suppr. par inc.	30107	405	413
Avec DM et suppr. par inc.	2016	335	329

On peut noter que le nombre d'alertes levées est fortement diminué par la réduction de la base de signatures. Ceci semble prouver que nous n'avons pas d'équivalence des systèmes de détection. Cependant, on peut facilement expliquer ces différences d'alertes levées.

Pour le système utilisant la décomposition modulaire, la diminution du nombre d'alertes s'explique par le fait que plusieurs signatures de la base originale peuvent être représentées par une seule signature modulaire. Ainsi, si plusieurs de ces signatures sont complétées par la même interaction, autant d'alertes seront levées par le système de détection original, tandis qu'une seule sera levée par le système utilisant la décomposition modulaire.

Par exemple, nous prenons les deux signatures de la base originale, sur le point d'être complétées, suivantes :
`shadow_t` → `passwd_d` → `login_d` → `admin_d` → $\boxed{\texttt{bin\_t} \rightarrow \texttt{user\_d}}$
`shadow_t` → `passwd_d` → `ssh_d` → `admin_d` → $\boxed{\texttt{bin\_t} \rightarrow \texttt{user\_d}}$

Ces signatures sont représentées par la signature modulaire suivante :
`shadow_t` → `passwd_d` → *module* → `admin_d` → $\boxed{\texttt{bin\_t} \rightarrow \texttt{user\_d}}$

Ici, dans la cas du système original, si l'interaction `bin_t` → `user_d` est effectuée, les deux signatures de la base originale sont complétées. Deux alertes sont alors levées. Dans le cas du système utilisant la décomposition modulaire, la signature est complétée, mais une seule alerte est levée. Il est donc possible d'avoir moins d'alertes levées sans avoir moins de comportements malicieux détectés.

Pour le système utilisant la suppression par inclusion, la diminution du nombre d'alertes levées s'explique par le fait que nous utilisons PIGA en mode IDS. Les interactions complétant une signature ne sont donc pas bloquées. Ainsi, il est possible qu'une signature soit complétée, et donc lève une alerte, alors qu'elle en inclut une autre. L'alerte ne sera pas levée par le système utilisant la suppression par inclusion car la signature complétée n'apparaît pas dans la base.

Par exemple, nous prenons les deux signatures de la base originale suivantes :
`shadow_t` → `passwd_d` → `ssh_d` → $\boxed{\texttt{admin\_d} \rightarrow \texttt{bin\_t}}$
`shadow_t` → `passwd_d` → `ssh_d` → `admin_d` → $\boxed{\texttt{bin\_t} \rightarrow \texttt{user\_d}}$

La première signature étant incluse dans la seconde, la base de signature générée en utilisant la suppression par inclusion ne contient que la première :
`shadow_t` → `passwd_d` → `ssh_d` → $\boxed{\texttt{admin\_d} \rightarrow \texttt{bin\_t}}$

Ici, dans la cas du système original, si l'interaction `bin_t` → `user_d` est effectuée, la deuxième signature de la base originale est complétée. Une alerte est alors levée. Dans le cas du système utilisant la suppression, aucune signature n'est complétée. Il est donc possible d'avoir moins d'alertes levées. Cependant, les comportements non détectés ne peuvent pas être effectués si PIGA est en mode IPS.

Afin d'évaluer l'équivalence des systèmes de détection, il ne faut donc pas comparer le nombre d'alertes levées mais les comportements détectés.

Le tableau ci-dessous représente les différences de comportements détectés entre

chaque nouvelle base et la base originale. Pour chaque trace et chaque nouvelle base, il indique le nombre de comportements non-détectés par la nouvelle base mais détectés par la base originale (non-détectés) et le nombre de comportements détectés, en plus, par la nouvelle base mais non-détectés par la base originale (sur-détectés).

	Différences avec le système original			
	Trace 1		Trace 2	
	Non-détectés	Sur-détectés	Non-détectés	Sur-détectés
Avec DM	0	0	0	0
Avec suppr. par inc.	381	0	407	0
Avec DM et suppr. par inc.	206	0	250	0

Tout d'abord, nous observons que, pour les deux traces et quel que soit le système de détection utilisé, aucune alerte « sur-détectés » n'est levée. Les comportements acceptés par le système original sont donc acceptés par les nouveaux systèmes.

De plus, le système utilisant uniquement la décomposition modulaire ne lève aucune alerte « non-détectés ». Tous les comportements détectés par le système original sont donc détectés par le système utilisant la décomposition modulaire.

Pour les systèmes de détection appliquant la suppression par inclusion, on observe qu'un grand nombre d'alertes « non-détectés » sont levées. Ceci semble prouver que nous n'avons pas équivalence des systèmes de détection. Cependant, toutes les alertes « non-détectés » obtenues incluent une signature détectée précédemment par le système original, ainsi que ceux utilisant la suppression par inclusion. Ces « non-détectés » apparaissent donc uniquement car PIGA est utilisé en mode IDS.

Cette expérimentation permet de vérifier, dans un cas pratique, que les systèmes de détection utilisant la décomposition modulaire ou la suppression par inclusion sont équivalents au système original. Pour le système utilisant uniquement la décomposition modulaire, cette équivalence est vraie quel que soit le mode dans lequel PIGA est utilisé. Quant aux systèmes utilisant la suppression par inclusion, l'équivalence n'est valable que si PIGA est utilisé en mode IPS.

Évaluation du temps d'exécution

Le temps mis par PIGA pour analyser une ligne de trace est un paramètre très important dans l'évaluation de son efficacité et surtout de sa viabilité. En effet, il est inconcevable que PIGA mette un temps trop important à vérifier qu'une opération ne complète pas une signature. Cette vérification étant effectuée à chaque appel système, un temps de réponse trop élevé engendrerait une chute importante des performances globales observées par un utilisateur. Le but cette thèse est de diminuer la taille de la base de signature, afin d'éviter des ralentissements du système dus à la mémoire utilisée. Il serait donc problématique que la compression opérée implique une augmentation du temps de réponse de PIGA et donc des ralentissements.

Le tableau ci-dessous représente le temps nécessaire pour analyser deux traces différentes de 100000 lignes, ainsi que le gain par rapport au système original. L'analyse a été lancée cinq fois pour chaque base et chaque trace. Les temps inscrits dans le tableau sont une moyenne des cinq temps obtenus.

		Nombre de signatures	Trace 1		Trace 2	
			Temps (en s)	Gain	Temps (en s)	Gain
1	Système original	152513	46,422	-	40,557	-
2	Avec DM	6870	17,476	62,4%	13,487	66,7%
3	Avec suppr. par inc.	30107	12,427	73,2%	9,449	76,7%
4	Avec DM et suppr. par inc.	2016	11,125	76%	8,976	77,9%

La première chose que l'on peut noter est que, quelle que soit la méthode de réduction de la base de signatures utilisée, le temps d'exécution est inférieur à celui utilisant la base originale. Ceci peut facilement s'expliquer par le fait que la complexité en temps des algorithmes de détection dépend principalement du nombre de signatures. En effet, la plus grande des bases de signatures réduites possède 30000 signatures contre 150000 pour la base originale. Pour la base la plus petite, correspondant à un taux compression à 98,6%, le temps nécessaire à l'analyse d'une trace est réduit de 76% pour la trace 1 et de 78% pour la trace 2.

Cependant, les temps des bases 2 et 3 montrent que le temps d'exécution ne dépend pas que du nombre de signatures. La base 2 possède environ quatre fois moins de signatures que la base 3, mais elle prend environ 50% de temps en plus à analyser. Ceci montre que le traitement de la traversée des modules implique une augmentation du temps d'exécution. On peut expliquer cette augmentation par le fait que, dans le cas de la base 2, on traite non seulement les signatures possédant l'interaction en cours d'analyse, mais également les signatures ayant un module en cours de contamination.

Cette expérimentation montre aussi que la réduction de la base de signature réduit significativement le temps d'exécution du système de détection. De plus, on note que la gestion des modules implique une augmentation de ce temps. Cependant, cette augmentation n'est pas suffisamment importante pour contrebalancer la diminution due à la taille de la base. Ainsi, l'application de la décomposition modulaire et de la suppression par inclusion permet, en plus de réduire la taille de la base de signatures, de diminuer le temps d'analyse d'une trace. Enfin, on note que les gains ne varient pas significativement en fonction de la trace analysée.

Évaluation de la consommation de mémoire

La mémoire utilisée, lorsque le système de détection est actif, est un des problèmes majeurs de PIGA. En effet, afin que l'utilisation de PIGA soit viable, il est nécessaire qu'il ne consomme pas une trop grande quantité de la mémoire du système qu'il doit sécuriser. Une trop grande consommation de mémoire aurait pour incidence une diminution des performances du système sécurisé. Le but de cette thèse est de diminuer la taille de la mémoire utilisée par PIGA lors de la détection. Nous avons

vu, dans la section 3.4, que la taille de la base de signatures, est largement réduite par l'utilisation de la décomposition modulaire ou de la suppression par inclusion. Cette expérimentation a pour but de vérifier que la mémoire nécessaire au traitement de ces nouvelles bases, et notamment des modules, ne contrebalance pas le gain sur la taille de la base.

Les données, stockées en mémoire lors de la détection, peuvent être séparées en deux parties. Premièrement, il existe une partie fixe qui ne dépend pas des signatures ni de leur nombre. Cette partie contient notamment le graphe d'interactions et, dans le cas de l'utilisation de la décomposition modulaire, l'arbre de décomposition modulaire. Ainsi, la partie fixe du système original est la même que celle du système utilisant uniquement la simplification par inclusion. Deuxièmement, il existe une partie variable qui est constituée principalement des signatures ainsi que de la table associant à chaque interaction les signatures l'impliquant. Nous pouvons faire deux remarques sur ces parties. Premièrement, la partie fixe est plus volumineuse si on utilise la décomposition modulaire. Ceci est dû au fait qu'il est nécessaire de stocker l'arbre de décomposition modulaire. Deuxièmement, la partie variable est très dépendante des signatures. Elle est donc proportionnelle à la taille de la base de signatures.

Le tableau ci-dessous représente la quantité de mémoire utilisée pour analyser deux traces différentes de 100000 lignes, ainsi que le gain par rapport au système original. L'analyse a été lancée cinq fois pour chaque base et chaque trace. Les quantités de mémoire inscrites dans le tableau sont une moyenne des cinq quantités obtenues.

		Taille de la base (en Mo)	Trace 1		Trace 2	
			Mémoire (en Mo)	Gain	Mémoire (en Mo)	Gain
1	Système original	89,5	564,1	-	544,52	-
2	Avec DM	3,4	340,1	39,7%	311,58	42,8%
3	Avec suppr. par inc.	24,2	238,4	57,7%	242	55,6%
4	Avec DM et suppr. par inc.	1,3	186,2	67%	164,66	69.8%

Les résultats obtenus sur la consommation de mémoire sont équivalents à ceux sur le temps d'exécution. La quantité mémoire utilisée est inférieure à celle utilisée par le système original, quelles que soient les méthodes utilisées pour réduire la base de signatures. De plus, l'utilisation de la suppression par inclusion seule permet un meilleur gain que l'utilisation de la décomposition modulaire seule.

On peut également noter que la trace analysée a un impact plus important sur les systèmes utilisant la décomposition modulaire. En effet, pour le système 1 (resp. 3), la différence de mémoire utilisée entre les deux traces est de 3,5% (resp. 1,5%). Tandis, que pour le système 2 (resp. 4), la différence est de 8,4% (resp. 11,6%). Ceci peut s'expliquer par le fait que, lors des processus de contamination, le nombre de sommets à mémoriser peut différer suivant la trace analysée.

Cette expérimentation montre que, quelle que soit la méthode de réduction de la base de signatures utilisée, la consommation de mémoire du système de détection est significativement réduite. On note également que l'utilisation de la décomposition

modulaire, pour réduire la base, induit une augmentation de la mémoire nécessaire au système de détection à cause de la gestion des modules. Cependant, cette augmentation n'est suffisamment importante pour contrebalancer la diminution due à la taille de la base. Enfin, nous avons pu observer que la trace analysée peut avoir un impact sur la mémoire consommée par les systèmes de détection utilisant la décomposition modulaire.

4.5 Gestion de modules consécutifs

Dans la section 4.2, nous n'avons pas analysé les cas induits par la présence de modules consécutifs dans une signature, car leur traitement n'est pas nécessaire. En effet, il possible de « casser » certains modules afin que ce type de signatures ne contiennent plus de modules consécutifs. Ces signatures seront donc traitées par l'algorithme 8 présenté dans la section 4.3.

Cette méthode a l'inconvénient d'augmenter le nombre de signatures générées et, ainsi, de réduire la compression de la base des signatures. Cette section présente une analyse préliminaire de quelques cas induits par la présence de modules consécutifs. Nous ne nous intéressons ici aux signatures contenant uniquement deux modules consécutifs. De nouveaux cas apparaissent avec l'augmentation du nombre de modules présents consécutivement.

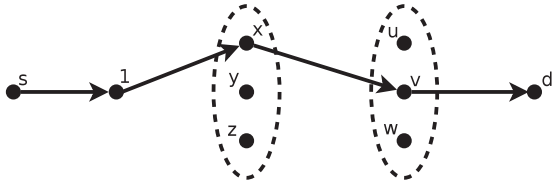
4.5.1 Analyse des nouveaux cas

La présence de plusieurs modules de manière de consécutive dans une signature est le cas le plus complexe à gérer. La principale difficulté vient du fait que les modules soient directement les uns à la suite des autres. En effet, la présence de plusieurs modules séparés par un ou plusieurs contextes simples se gère de la même manière que la présence d'un module seul. Il faut uniquement répéter le processus pour chaque module. Voici une liste des cas qu'il faut traiter pour la signature ci-dessous. Cette liste est non-exhaustive. Il existe d'autres cas où un comportement est détecté par une signature, alors qu'il correspond à une autre signature.

$$S_3 = s \rightarrow 1 \rightarrow Module\ M_1 \rightarrow Module\ M_2 \rightarrow d$$
$$M_1 = \{x, y, z\},\ M_2 = \{u, v, w\}$$

1. **Le cas de la traversée des deux modules sans boucle :**

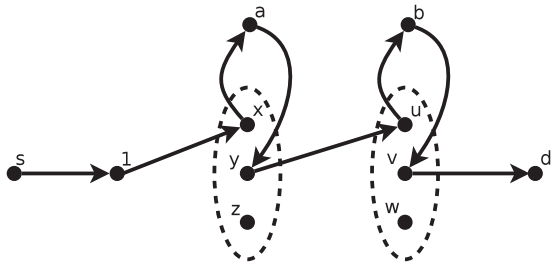
$s \rightarrow 1$
$1 \rightarrow x$
$x \rightarrow v$
$v \rightarrow d$



Pour cette trace, on sort de chaque module directement après y être entré et par le même sommet. Le flux d'informations va bien du sommet s au sommet d en respectant la signature. La signature doit être complétée.

2. **Le cas des boucles :**

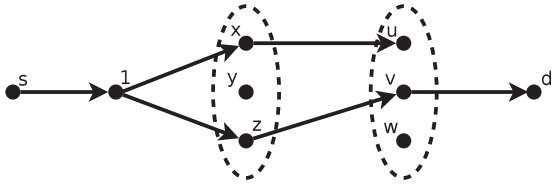
$s \to 1$
$1 \to x$
$x \to a$
$a \to y$
$y \to u$
$u \to b$
$b \to v$
$v \to d$



Pour cette trace, on effectue une boucle sur chaque module avant d'en sortir. Le flux d'informations va bien du sommet s au sommet d, en respectant la signature. La signature doit être complétée.

3. **Le cas du backtrack :**

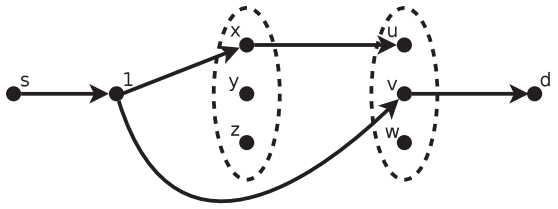
$s \to 1$
$1 \to x$
$x \to u$
$1 \to z$
$z \to v$
$v \to d$



Pour cette trace, on traverse le module M_1 par le sommet x et on entre dans M_2 par le sommet u. Ensuite, on traverse de nouveau M_1 par le sommet z, puis on traverse M_2 par le sommet v. Le flux d'informations va bien du sommet s au sommet d en respectant la signature. La signature doit être complétée.

4. **Le cas de contournement :**

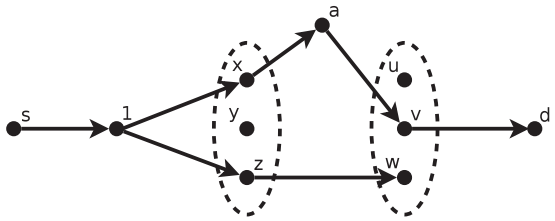
$s \to 1$
$1 \to x$
$x \to u$
$1 \to v$
$v \to d$



Sur cette trace, on entre dans le module M_1 par le sommet x et on entre dans M_2 par le sommet u. Puis, on entre une nouvelle fois dans le module M_2 mais sans être passé par le module M_1. Le flux d'informations va bien du sommet s au sommet d mais ne correspond pas à la signature. La signature ne doit pas être complétée. C'est le rôle de la signature $s \to 1 \to M_2 \to d$.

5. **Le cas du pont :**

$s \to 1$
$1 \to z$
$z \to w$
$1 \to x$
$x \to a$
$a \to v$
$v \to d$



Sur cette trace, on entre dans le module M_1 par le sommet z et on entre dans M_2 par le sommet w. Ensuite, contrairement au cas du contournement, on passe bien par le module M_1, puis par le module M_2. Cependant, entre les deux modules, on passe par le sommet a. Le flux d'informations va bien du sommet s au sommet d mais ne correspond pas à la signature. La signature ne doit pas être complétée. C'est le rôle de la signature $s \to 1 \to M_1 \to a \to M_2 \to d$.

4.5.2 Gestion théorique

1. **Le cas classique :**

   Ce cas se gère de la même manière que le cas classique avec un seul module. On peut supprimer de la mémoire, le sommet d'entrée d'un module lorsque l'on entre dans le suivant.

	Trace	Entrée M_1	Entrée M_2
1	$s \to 1$	–	–
2	$1 \to x$	x	–
3	$x \to v$	–	v
4	$v \to d$	–	–

2. **Le cas des boucles :**

   Ce cas se gère de la même manière que les boucles sur un seul module, par contamination. Cependant, lorsque que l'on entre dans un module qui n'est pas le premier de la signature, il faut mémoriser tous les sommets du module précédent comme prédécesseurs.

	Trace	Prédécesseurs	Sommets contaminés
1	$s \to 1$	–	–
2	$1 \to x$	1	x
3	$x \to a$	1	x, a
4	$a \to y$	1	x, a, y
5	$y \to u$	x, y, z	u
6	$u \to b$	x, y, z	u, b
7	$b \to v$	x, y, z	u, b, v
8	$v \to d$	–	–

3. **Le cas du backtrack :**
   Afin de gérer les backtracks, il est possible d'utiliser le système de contamination, mais en conservant en mémoire les sommets contaminés des modules précédents. Ainsi, les modules consécutifs sont considérés comme un seul méta-module qui s'agrandit au fur et à mesure qu'on avance dans la signature. Le méta-module s'agrandit lorsque qu'une interaction fait passer le flux d'informations du dernier module du méta-module au module suivant. Le méta-module devient alors l'union de lui-même et du module nouvellement atteint par le flux d'informations.

	Trace	Prédécesseur	Sommets contaminés	Méta-module
1	$s \to 1$	–	–	–
2	$1 \to x$	1	x	M_1
3	$x \to u$	1	x, u	$M_1 \cup M_2$
4	$1 \to z$	1	x, u, z	$M_1 \cup M_2$
5	$z \to v$	1	x, u, z, v	$M_1 \cup M_2$
6	$v \to d$	–	–	–

4. **Le cas du contournement :**
   Lors d'un backtrack sur des modules consécutifs, le système de gestion des backtracks autorise un contournement des modules traversés. Dans l'exemple de la section précédente, le module M_2 a été atteint. Il est donc inclus dans le méta-module. Or, lorsqu'un sommet est dans le méta-module, il peut être contaminé par le sommet précédant les modules (ici le sommet 1). Pour éviter que les contournements ne soient considérés comme des complétions de signature, il faut vérifier que, pour chaque transfert dont la source est le prédécesseur, la destination appartienne au premier des modules consécutifs.

	Trace	Prédécesseur	Sommets contaminés	1^{er} module	Méta-module
1	$s \to 1$	–	–	–	–
2	$1 \to x$	1	x	M_1	M_1
3	$x \to u$	1	x, u	M_1	$M_1 \cup M_2$
4	$1 \to v$	1	x, u	M_1	$M_1 \cup M_2$
5	$v \to d$	–	–	–	–

5. **Le cas du pont :**
   Le problème du pont vient du fait que tous les modules ayant été atteints par le flux sont considérés comme un seul méta-module. Or, suite à un backtrack, il devient possible de passer d'un module à l'autre via un sommet intermédiaire, ce qui ne correspond pas à la signature. Il est donc nécessaire de s'assurer qu'on peut entrer dans un module, qui n'est pas le premier des modules consécutifs, uniquement par un sommet du module précédent. Ainsi, un sommet appartenant à un module ne peut être contaminé que par un sommet qui est dans la liste de sommets contaminés de son module, ou par un sommet contaminé du module précédent.

	Trace	Prédécesseur	Contaminés M_1	Contaminés M_2	Méta-module
1	$s \to 1$	–	–	–	–
2	$1 \to z$	1	z	–	M_1
3	$z \to w$	1	z	w	$M_1 \cup M_2$
4	$1 \to x$	1	z, x	w	$M_1 \cup M_2$
5	$x \to a$	1	z, x, a	w	$M_1 \cup M_2$
6	$a \to v$	1	z, x, a	w	$M_1 \cup M_2$
7	$v \to d$	1	z, x, a	w	$M_1 \cup M_2$

Lors de la détection, la gestion de deux modules consécutifs demande des ressources mémoire supplémentaires pour stocker toutes les informations de contamination. Contrairement à la gestion d'un seul module, ces informations peuvent rester longtemps en mémoire et, ainsi, diminuer les performances du système. Cet effet s'accentue si le nombre de modules consécutifs est important. Une analyse plus poussée est donc nécessaire afin de vérifier que le gain en compression, dû à la présence de modules consécutifs, n'est pas contre-balancé par cette consommation mémoire.

Les problèmes engendrés par la présence de modules consécutifs correspondent au fait qu'une signature détecte une violation de propriété censée être détectée par une autre signature. La seule conséquence à cela est la levée de multiples alertes pour une même attaque détectée. Nous ne sommes donc pas dans le cas d'une attaque non-détectée ou d'une fausse alerte. Leur non-traitement engendrerait donc une absence de correspondance entre le comportement détecté et la signature levant l'alerte. Cependant, l'équivalence en terme de détections de violations de propriétés de sécurité serait conservée.

Néanmoins, le traitement des modules consécutifs est nécessaire si l'on souhaite mettre en place une gestion fine des exceptions.

4.6 Gestion des exceptions

Le système de détection original de PIGA permet de définir des exceptions. Ces comportements correspondent à une violation de propriété de sécurité, mais les alertes les concernant sont ignorées lors de la détection. De plus, si PIGA est en mode IPS et, si toutes les alertes levées par une interaction concernent des exceptions, alors l'interaction n'est pas bloquée.

Actuellement, les exceptions sont définies dans un fichier chargé par le système de détection. Le comportement est donc représenté dans la base de signatures. Dans le système original, une signature correspond à un seul comportement. Il est donc facile de définir une exception. En effet, il suffit de spécifier l'identifiant de la signature correspondant au comportement et, si une alerte concernant cette signature doit être levée, elle n'est pas prise en compte.

L'utilisation de la décomposition modulaire ou de la suppression par inclusion implique différents problèmes liés à la gestion des exceptions.

Décomposition modulaire

Lors de la génération de la base signatures, l'utilisation de la décomposition modulaire permet de compresser plusieurs signatures du système original en une signature modulaire. Ainsi, une signature modulaire ne représente plus un comportement mais plusieurs. Si un comportement est représenté par une signature modulaire, il n'est donc plus possible de définir une exception concernant uniquement celui-ci : soit l'exception s'applique à tous les comportements représentés par la signature modulaire, soit aucun comportement n'est considéré comme une exception.

De plus, il est possible qu'un même comportement soit représenté par deux signatures différentes. Par exemple, il est possible, en utilisant le principe des boucles sur module que deux signatures incluant les deux même modules représentent le même comportement. Soit deux modules :

$$M_1 = \{x, y, z\},\ M_2 = \{u, v, w\}$$

Dans le comportement suivant, on remplace les contextes par le module auquel ils appartiennent :

$$1 \to\ x\ \to 2 \to\ u\ \to 3 \to\ y\ \to 4 \to\ v\ \to 5$$

Le comportement modulaire obtenu possède alors deux contextes qui apparaissent deux fois. Il faut supprimer les cycles afin d'obtenir la signature représentant ce comportement :

$$1 \to M_1 \to 2 \to M_2 \to 3 \to M_1 \to 4 \to M_2 \to 5$$

Il y a deux moyens de supprimer les cycles :

- Soit on supprime le cycle sur M_1 :
$$1 \to \boxed{M_1 \to 2 \to M_2 \to 3 \to M_1} \to 4 \to M_2 \to 5$$
$$1 \to \qquad M_1 \qquad \to 4 \to M_2 \to 5$$
- Soit on supprime le cycle sur M_2 :
$$1 \to M_1 \to 2 \to \boxed{M_2 \to 3 \to M_1 \to 4 \to M_2} \to 5$$
$$1 \to M_1 \to 2 \to \qquad M_2 \qquad \to 5$$

Il y a donc deux signatures qui représentent le comportement :

$$1 \to M_1 \to 4 \to M_2 \to 5$$
$$1 \to M_1 \to 2 \to M_2 \to 5$$

Si l'on souhaite ajouter une exception pour un comportement représenté par plusieurs signatures modulaires, il faut ajouter une exception pour chacune des signatures le représentant. Il faut donc tout d'abord, obtenir l'ensemble des signatures représentant le comportement à ajouter aux exceptions.

Ce problème apparaît avec la gestion des boucles sur module. Cependant, il devient plus important si l'on ajoute la gestion des modules consécutifs. En effet, dans la gestion de ces modules, la plupart des difficultés viennent du fait qu'une signatures détecte un comportement qu'elle ne représente pas. Il est donc nécessaire de traiter tous les cas problématiques induits par la présence de modules consécutifs, avant de pouvoir ajouter la gestion des exceptions.

Suppression par inclusion

La méthode de suppression par inclusion consiste à supprimer de la base les signatures qui incluent une autre signature de cette base. Cette méthode n'est utilisable que si PIGA est utilisé en mode IPS, car, dans ce cas, si une interaction complète la signature incluse, cette interaction est bloquée. Les signatures incluant la signature complétée ne peuvent donc jamais être complétées.

Si l'on définit une signature comme étant une exception, la dernière interaction de cette signature n'est plus bloquée par le système de détection. Ainsi, les signatures incluant cette signature peuvent être complétées. Cependant, les exceptions étant définies après la génération de la base de signatures, celles-ci ne sont pas présentes dans la base. Les comportements qu'elles représentent ne peuvent donc pas être détectés, et ainsi permettre des violations de la politique de sécurité.

Afin d'éviter ce problème, il est nécessaire de définir les exceptions lors de la génération de la base de signatures. Les signatures ainsi définies comme des exceptions sont alors supprimées de la base avant l'application de la suppression par inclusion.

4.7 Conclusion

Dans ce chapitre, nous avons tout d'abord étudié la manière d'utiliser les signatures générées par application de la décomposition modulaire. La possibilité d'effectuer des boucles sur module demande une gestion plus fine et plus gourmande en mémoire de la progression de chaque signature, lors de la détection.

À partir de cette étude, nous avons mis au point un algorithme de détection permettant de gérer la présence de modules non-consécutifs et des boucles sur module. La complexité en temps de cet algorithme est en $O(s.t)$ où s représente le nombre de signatures et t le nombre de lignes de la trace analysée. Sa complexité en espace est en $O(S.n)$ où $\mathcal{S}$ représente la base de signatures et n le nombre de sommets du graphe d'interactions utilisé pour la génération de la base.

Nous avons ensuite prouvé l'équivalence de notre nouveau système de détection avec le système de détection orignal de PIGA. Cette équivalence signifie que tout comportement détecté par PIGA sera détecté par notre système de détection et inversement. Pour les bases de signatures utilisant la suppression par inclusion, l'équivalence avec le système original est valable uniquement dans le cas où PIGA est utilisé en mode IPS.

Les expérimentations effectuées montrent que l'application de la décomposition modulaire et de la suppression par inclusion permet de réduire, à la fois le temps et la mémoire nécessaires à la détection. La réduction de mémoire utilisée est inférieure à la compression obtenue sur la base de signature. Cependant, elle reste suffisante pour confirmer l'intérêt de l'utilisation de la décomposition modulaire et de la suppression par inclusion, lors de la génération de la base de signature.

Ces expérimentations ont également permis de vérifier l'équivalence entre le nouveau système de détection et le système original.

Enfin nous avons présenté deux améliorations possibles de notre système. Tout

d'abord, la mise en place du traitement des modules consécutifs permettrait, théoriquement, d'améliorer la compression de la base de signature. Pour cela, il est nécessaire d'étoffer l'analyse présentée dans ce chapitre.

Une seconde amélioration consiste à intégrer la gestion des exceptions. Chacune des méthodes de compression utilisée induit des problèmes pour gérer des exceptions précises.

L'application de la décomposition modulaire rend difficile la possibilité d'isoler un comportement précis, si celui-ci est représenté par une ou plusieurs signatures modulaires. Il devient donc complexe d'ajouter des exceptions.

Lors de l'utilisation de l'application de suppression par inclusion, l'ajout d'exceptions peut conduire à rendre non détectables, car étant considérés comme irréalisables, des comportements violant la politique de sécurité.

Conclusion

Durant cette thèse, nous nous sommes attachés à réduire la mémoire utilisée par PIGA, lorsque son système de détection est activé. La principale source d'utilisation de la mémoire est le stockage de la base de comportements violant la politique de sécurité définie pour PIGA. Ces comportements malicieux sont représentés par une base de signatures, chaque signature correspondant à un comportement. Les signatures étant générées à partir d'un graphe, nous avons cherché des méthodes de réduction de la base sur deux axes : réduire directement la base de signatures et réduire le graphe utilisé pour la génération de signatures.

Le premier chapitre a exposé le fonctionnement général de PIGA et s'est attardé sur les systèmes de génération de signatures et de détection.

Tout d'abord, nous avons présenté des notions classiques de sécurité, telles que les systèmes de prévention et détection d'intrusions, ainsi que les grandes familles de propriétés de sécurité. Puis, nous avons expliqué la manière dont PIGA utilise le système de contrôle d'accès mandataire, sur lequel il se greffe, afin de générer un graphe d'interactions représentant les actions autorisées sur le système d'exploitation.

Nous avons alors détaillé comment PIGA utilise ce graphe pour générer les signatures, afin de créer une base de comportements violant les propriétés de sécurité décrites dans la politique. Pour cela, nous avons vu que PIGA génère différents graphes à partir du graphe d'interactions suivant le type des propriétés. Ensuite, nous avons décrit le système de détection de PIGA. Ce système fait progresser les signatures à mesure que des actions sont effectuées sur le système. Quand une signature est complétée, PIGA, suivant son mode de fonctionnement, bloque la dernière action ou lève une alerte.

Enfin, nous avons défini la problématique de cette thèse : réduire la taille de la base de signatures. Pour cela, nous proposons deux approches. La première consiste à travailler directement sur les signatures en supprimant celles qui ne peuvent pas être complétées. La seconde consiste à réduire la taille du graphe d'interaction afin de diminuer le nombre et la taille des signatures. Pour cela, nous regroupons les sommets du graphe qui partagent le même comportement. L'outil de la théorie des graphes effectuant cette action se nomme la décomposition modulaire.

Dans le deuxième chapitre, nous avons expliqué le fonctionnement de la décomposition modulaire. Nous avons tout d'abord rappelé les fondements théoriques de cet outil. Notamment, trois objets de la théorie des ensembles : les partitions, les familles partitives et les permutations factorisantes.

À partir de cela, nous avons défini les notions de modules d'un graphe, d'arbre de

décomposition modulaire et de graphe quotient modulaire. L'intérêt de ce graphe est qu'il permet de diminuer la taille de notre graphe en limitant la perte d'information. Puis, nous avons vu les différents types de modules ainsi que les propriétés qui leurs sont associées. Nous avons ensuite analysé les algorithmes de décomposition modulaire orienté et non-orienté, ainsi que leur complexité en temps.

Enfin, nous avons présenté quelques applications de la décomposition modulaire. Nous avons pu noter que l'on trouve de nombreuses applications sur des problèmes de la théorie des graphes, tels que les ensembles stables ou la coloration. Cependant, les applications sur des problèmes pratiques, telles que le dessin de graphe ou les réseaux d'interactions protéine-protéine, sont plus rares.

Le chapitre 3 présente la première partie de notre contribution. Nous avons appliqué deux méthodes de réduction de la base de signatures : l'application de la décomposition modulaire sur le graphe d'interactions et la suppression par inclusion. Pour tester l'efficacité de nos méthodes, nous nous sommes concentrés sur les propriétés de confidentialité. Celles-ci génèrent le plus de signatures.

Tout d'abord, nous avons établi les deux problèmes principaux de l'application de la décomposition modulaire. Le premier problème est que les différents graphes utilisés par PIGA ne donnent pas la même décomposition orientée. La génération des signatures utilisant plusieurs graphes par propriété, la décomposition modulaire orientée ne peut pas être utilisée. Pour résoudre ce problème, nous avons utilisé la décomposition modulaire non-orientée. Le second problème est que les extrémités des signatures n'apparaissent pas obligatoirement sur le graphe quotient modulaire. Nous avons donc présenté deux algorithmes permettant d'extraire des sommets de leur module, tout en gardant un graphe quotient aussi compact que possible.

Ensuite, nous avons évalué l'efficacité de l'application de la décomposition modulaire sur la génération de signatures. Tout d'abord, nous avons calculé la compression théorique de la présence d'un module dans une signature, c'est-à-dire le nombre de signatures représentées par une signature modulaire. Puis, nous avons présenté plusieurs expérimentations. Pour la première, sur un graphe jouet, nous avons obtenu une compression moyenne de plus de 98% pour toutes les paires source/destination possibles. Pour la deuxième, nous avons travaillé sur un graphe de flux réel tiré d'un pot de miel utilisant PIGA. Nous avons obtenu une compression moyenne de plus de 95% pour six paires source/destination. Enfin, la troisième expérimentation a été effectuée sur un graphe de processus pour une seule paire. En limitant la longueur des signatures à 4, nous avons obtenu un taux de compression de 99%.

Puis, nous avons évalué les performances de la suppression par inclusion. Le principe de cette méthode est de supprimer de la base toutes les signatures qui incluent une autre signature de la base. En effet, si le comportement représenté par cette dernière est bloqué, alors toutes les signatures qui l'incluent ne peuvent être complétées. Nous avons appliqué cette méthode seule et avec la décomposition modulaire sur le graphe de flux. Dans le premier cas, nous avons obtenu un taux de compression de 80%. Dans le second cas, le taux de compression obtenu est de plus de 98%.

Enfin, nous avons discuté des limites d'utilisation de ces méthodes. L'application de la décomposition modulaire ne permet pas d'avoir une analyse précise des com-

portements violant la politique de sécurité. En effet, chaque module se comporte comme une boîte noire. La mémorisation du chemin qui le traverse nécessite une quantité de mémoire conséquente. Le suppression par inclusion se base sur le fait que la dernière interaction d'une signature est bloquée. Elle ne peut donc être utilisée que si PIGA est en mode IPS. Dans le cas contraire, les signatures supprimées peuvent être complétées sans que le système ne les détecte.

Ce chapitre a mis en évidence que les méthodes proposées permettent de compresser, de manière efficace, une base de signatures. Le taux de compression est très dépendant de la forme du graphe. Cependant, les modules obtenus possèdent une cohérence sémantique. On peut donc supposer que les graphes utilisés par PIGA se prêtent bien à l'application de la décomposition modulaire. Il nous a ensuite fallu adapter le système de détection à la forme des nouvelles signatures.

Le quatrième chapitre présente la deuxième partie de notre contribution : l'adaptation du système de détection de PIGA aux signatures modulaires. Les expérimentations, décrites dans ce chapitre, ont pour but de confirmer l'efficacité de nos méthodes de compression. En effet, le but initial de cette étude n'était pas de réduire le nombre de signatures, mais de réduire la mémoire nécessaire à PIGA lors de la détection. Il nous fallait obtenir un temps de réponse à chaque trace au plus égal à celui de l'ancien système.

Tout d'abord, nous avons présenté une liste des nouveaux cas engendrés par la présence de modules dans les signatures. Nous avons analysé en détail les problèmes induits par la présence d'un module ou de plusieurs modules non-consécutifs. Pour chacune des cas, nous avons présenté les problèmes qui pouvaient apparaître, puis nous avons proposé une solution pour les gérer.

Ensuite, nous avons décrit deux algorithmes de détection des signatures. Le premier permet de gérer certains des cas engendrés par la présence d'un module isolé, tandis que le deuxième permet de gérer tous les cas. Nous n'avons pas présenté d'algorithme gérant les modules consécutifs, car notre liste de cas les concernant n'était pas exhaustive. De plus, nous n'avons généré aucune signature possédant des modules consécutifs. Pour chacun de ces algorithmes, nous avons analysé leurs complexités en temps et en espace. Les deux nouveaux algorithmes ont une complexité en temps identique à l'algorithme original, c'est-à-dire en $O(s.t)$. Leur complexité en espace est en $O(\mathcal{S} + s.n)$, tandis que l'algorithme orignal exécute en $O(\mathcal{S})$. Nous avons ensuite prouvé l'équivalence des systèmes de détection.

Afin de vérifier les performances du deuxième algorithme, nous avons mesuré ses performances grâce à des traces générées semi-aléatoirement. Les critères étaient : le nombre d'alertes levées, le temps d'exécution et la mémoire consommée. Nous avons comparé quatre bases de signatures différentes : la base originale, la base appliquant la décomposition modulaire, la base appliquant la suppression par inclusion et la base appliquant les deux méthodes. Tout d'abord, quelle que soit la base utilisée, les mêmes alertes ont été levées au même moment. Pour le temps d'exécution, nous avons noté une diminution de celui-ci. Cette diminution est plus forte si on utilise la suppression par inclusion. Les mêmes remarques ont été faites concernant la mémoire utilisée. Notre algorithme est donc plus efficace en temps et en mémoire consommée, tout en gardant la même efficacité en terme de détection de comportements

malicieux.

Nous avons ensuite présenté une analyse préliminaire des cas problématiques induits par la présence de modules consécutifs dans une signature. Le traitement de ce type de signatures, s'il n'est pas nécessaire, peut, théoriquement, permettre d'augmenter la compression.

Enfin, nous avons soulevé un problème lié à l'utilisation de la décomposition modulaire et de la suppression par inclusion. En effet, la gestion d'exception devient plus complexe si l'on utilise l'une ou l'autre de ces méthodes. Pour la première méthode, il devient difficile d'isoler un comportement d'une signature modulaire. Pour la seconde, si une signature incluse dans une autre est une exception, alors il ne faut pas supprimer cette dernière.

L'étude effectuée durant cette thèse a obtenu de bons résultats. Sur le graphe flux et pour six paires source/destination, le fichier contenant la base de signatures a diminué de 98,6% en utilisant les deux méthodes de réduction. Lors de la détection, nous avons noté une diminution moyenne de plus 75% du temps d'exécution et une diminution moyenne de plus de 68% de la mémoire consommée. De plus, l'utilisation de ces méthodes ne diminue pas l'efficacité de PIGA en terme de détection de comportements malicieux. Enfin, notre système est encore améliorable tant d'un point de vue algorithmique que d'un point de vue de l'intégration à PIGA.

Perspectives

Suite à cette étude, des perspectives s'ouvrent sur trois axes : l'amélioration de l'intégration des méthodes de réduction à PIGA, l'amélioration des algorithmes utilisés et l'application de nos méthodes à d'autres problèmes.

Améliorations de l'intégration à PIGA

Actuellement, nos méthodes ne sont appliquées qu'aux propriétés de confidentialité avec quelques limitations.

La première amélioration possible est d'appliquer nos méthodes de réduction aux autres propriétés de sécurité utilisées par PIGA. Les deux méthodes peuvent être appliquées de manière simple aux autres propriétés générant des signatures simples. Cependant, pour les propriétés générant des signatures composées, l'utilisation de nos méthodes est plus compliquée. En effet, la suppression par inclusion ne peut être utilisée que sur la dernière partie des signatures composées.

La seconde amélioration consiste à intégrer la gestion des exceptions. L'ajout d'exceptions se fait actuellement en spécifiant à PIGA l'identifiant des signatures représentant les comportements à ignorer. Ce système a la particularité de permettre l'ajout d'exceptions a posteriori de la génération de signatures. Avec l'utilisation de la décomposition modulaire, une signature peut représenter plusieurs comportements. L'éclatement d'un ou plusieurs modules, afin d'isoler un comportement précis, demande de recalculer une partie de la base ce qui est assez contraignant. Une solution pourrait être d'ajouter une base de signatures représentant l'ensemble des comportements à ignorer. Les deux bases progresseraient alors en parallèle.

Améliorations algorithmiques

Les systèmes de génération de signatures et de détection développés lors de cette thèse fonctionnent de manière efficace avec les données testées. Cependant, des optimisations ainsi qu'un élargissement du spectre des données traitables sont possibles.

Actuellement, notre système de génération de signatures n'utilise qu'une partie de l'arbre de décomposition modulaire. En effet, seul le premier niveau de cet arbre est utilisé pour générer le graphe quotient. Lors de l'analyse de l'algorithme de décomposition modulaire non-orienté, nous avons observé qu'à la fin de la première itération de l'algorithme calculant la permutation, on obtient une décomposition modulaire où le sommet choisi comme centre est isolé. En choisissant ce centre comme une des extrémités des signatures, puis en faisant une seconde itération sur le module contenant l'autre extrémité en prenant celle-ci pour centre, on obtient alors une décomposition modulaire avec les deux extrémités isolées. En utilisant cette méthode, il n'est plus nécessaire de calculer une première fois la décomposition modulaire, puis de modifier l'arbre pour en extraire les extrémités.

Pour le moment, le système de détection ne gère pas la présence de modules consécutifs dans une signature. Nous avons proposé des moyens de modifier ce type de signatures afin de ne plus avoir de modules consécutifs. Cependant, afin de conserver une compression optimale, il nous faut conserver ces signatures. Une amélioration de notre système serait de modifier l'algorithme de détection afin qu'il gère la présence de modules consécutifs. Cette modification nécessiterait un approfondissement de l'étude des cas problématiques induits par ce type de signature, présentée dans cette thèse.

Applications

Notre problématique initiale était de diminuer la mémoire nécessaire lorsque le système de détection de PIGA est actif. Rapidement, nous avons réduit ce problème à une diminution de la taille de la base de signatures générées par PIGA. Or, une signature est soit un chemin dans un graphe, soit une composition de plusieurs chemins. Les solutions proposées dans cette thèse sont donc des moyens de réduire le nombre de chemins entre deux sommets d'un graphe. Notre méthode d'application de la décomposition modulaire peut être appliquée à différents problèmes nécessitant une compression des chemins sur un graphe. Dans le cas d'un graphe orienté, une implémentation de l'algorithme de décomposition modulaire orienté est nécessaire.

Bibliographie

[ABB+12] Z. Afoulki, A. Bousquet, J. Briffaut, J. Rouzaud-Cornabas, and C. Toinard. PIGA-Cloud : une protection obligatoire des environnements d'informatique en nuage. In *NOTERE CFIP 2012*, 2012.

[AEKBT+05] A. Abou El Kalam, J. Briffaut, C. Toinard, M. Blanc, and L. Oudot. MIDS : Multi level Intrusion Detection System. In *The 4th Conference on Security and Network Architectures*, pages 145–155, 2005.

[BBC+06a] M. Blanc, J. Briffaut, P. Clemente, M. Gad El Rab, and C. Toinard. A Collaborative Approach for Access Control, Intrusion Detection and Security Testing. In *The 2006 International Symposium on Collaborative Technologies and Systems, Special Session on Multi Agent Systems and Collaboration*, pages 270–278, 2006.

[BBC+06b] M. Blanc, J. Briffaut, P. Clemente, M. Gad El Rab, and C. Toinard. A Multi-Agent and Multi-Level Architecture to Secure Distributed Systems. In *First International Workshop on Privacy and Security in Agent-based Collaborative Environments*, 2006.

[BBC12] P. Berthomé, J. Briffaut, and P. Clairet. Compacting security signatures for PIGA IDS. In *The Sixth International Conference on Emerging Security Information, Systems and Technologies*, pages 126–133, 2012.

[BBLT06] M. Blanc, J. Briffaut, J-F. Lalande, and C. Toinard. Distributed control enabling consistent MAC policies and IDS based on a meta-policy approach. In *Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 153–156, 2006.

[Ber85] C. Berge. *Graphs and Hypergraphs.* Elsevier Science Ltd, 1985.

[BGBT12] M. Blanc, D. Gros, J. Briffaut, and C. Toinard. PIGA-Windows : contrôle des flux d'information avancés sur les systèmes d'exploitation Windows 7. In *9ème édition de la conférence MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication - MajecSTIC 2012 (2012)*, 2012.

[Bis03] M. Bishop. *Computer Security : Art and Science.* Addison-Wesley, 2003.

[BM08] J.A. Bondy and U.S.R. Murty. *Graph Theory.* Springer, 2008.

[BPT+11] J. Briffaut, M. Peres, C. Toinard, J. Rouzaud-Cornabas, B. Venelle, and J. Solanki. PIGA-OS : Retour sur le Système d'Exploitation

Vainqueur du Défi Sécurité. In *RenPar'20 / SympA'14 / CFSE 8, 8ème Conférence Francaise en Systèmes d'Exploitation*, 2011.

[BRCTZ09] 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*, pages 184–192, 2009.

[Bri07] J. Briffaut. *Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions*. PhD thesis, Université d'Orléans, 2007.

[Cap96] C. Capelle. *Décompositions de graphes et permutations factorisantes*. PhD thesis, Université Montpellier II, 1996.

[CHdM02] C. Capelle, M. Habib, and F. de Montgolfier. Graph Decompositions and Factorizing Permutations. *Discrete Mathematics and Theoretical Computer Science*, pages 55–70, 2002.

[CHM81] M. Chein, M. Habib, and M.C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, pages 35 – 50, 1981.

[Cid04] D.B. Cid. OSSEC : Open source host-based intrusion detection system. `http://www.ossec.net/`, 2004.

[Co80] James P. Anderson Co. *Computer Security Threat Monitoring and Surveillance*. 1980.

[Den87] D.E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, pages 222–232, 1987.

[Dep85] Department of Defense. *Trusted Computer System Evaluation Criteria*. 1985.

[dM03] F. de Montgolfier. *Décomposition modulaire des graphes. Théorie, extensions et algorithmes*. PhD thesis, Université des Sciences et Techniques du Languedoc, 2003.

[Epp99] D. Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, pages 652–673, 1999.

[GKBC04] J. Gagneur, R. Krause, T. Bouwmeester, and G. Casari. Modular decomposition of protein-protein interaction networks. *Genome Biology*, page R57, 2004.

[HdMP04] M. Habib, F. de Montgolfier, and C. Paul. A simple linear-time modular decomposition algorithm for graphs, using order extension. In *Algorithm Theory - SWAT 2004*, pages 187–198. Springer Berlin Heidelberg, 2004.

[HLCMM92] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX : Software architecture and rule-based language for universal audit trail analysis. In *Computer Security — ESORICS 92*, pages 435–450. Springer Berlin Heidelberg, 1992.

[HLR92] P. Helman, G. Liepins, and W. Richards. Foundations of intrusion detection. In *Computer Security Foundations Workshop V, 1992. Proceedings.*, pages 114–120, 1992.

[HP10] M. Habib and C. Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, pages 41–59, 2010.

[HPV99] M. Habib, C. Paul, and L. Viennot. Partition refinement techniques : An interesting algorithmic tool kit. *Int. J. Found. Comput. Sci.*, pages 147–170, 1999.

[KR02] C. Ko and T. Redmond. Noninterference and intrusion detection. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 177–187, 2002.

[MdM05] R. McConnell and F. de Montgolfier. Linear-time modular decomposition of directed graphs. pages 198–209, 2005.

[MR84] R.H. Möhring and F.J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. In *Algebraic and Combinatorial Methods in Operations Research Proceedings of the Workshop on Algebraic Structures in Operations Research*, pages 257 – 355. North-Holland, 1984.

[Off91] Office for Official Publications of the European Communities - Commission. ITSEC : Information Technology Security Evaluation Criteria (Provisional Harmonised Criteria, Version 1.2). pages 92–826, 1991.

[PAF+09] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee. McPAD : A multiple classifier system for accurate payload-based anomaly detection. *Computer networks*, pages 864–881, 2009.

[Pau06] C. Paul. *Aspects algorithmiques de la décomposition modulaire*. Hdr, Université Montpellier II - Sciences et Techniques du Languedoc, 2006.

[Pax99] V. Paxson. Bro : A system for detecting network intruders in real-time. *Comput. Netw.*, pages 2435–2463, 1999.

[PV05] C. Papadopoulos and C. Voglis. Drawing graphs using modular decomposition. In *Graph Drawing. Volume LNCS 3843*, pages 343–354. Springer, 2005.

[Rao06] M. Rao. *Décompositions de graphes et algorithmes efficaces*. PhD thesis, Université Paul Verlaine - Metz, 2006.

[Roe99] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, pages 229–238. USENIX Association, 1999.

[WPRW05] B. Wotring, B. Potter, M. Ranum, and R. Wichmann. *Host Integrity Monitoring Using Osiris and Samhain*. Syngress Publishing, 2005.

Pierre CLAIRET

Approche algorithmique pour l'amélioration des performances du système de détection d'intrusions PIGA

PIGA est un outil permettant de détecter les comportements malicieux par analyse de trace système. Pour cela, il utilise des signatures représentant les comportements violant une ou plusieurs propriétés de sécurité définies dans la politique. Les signatures sont générées à partir de graphes modélisant les opérations entre les différentes entités du système et sont stockées en mémoire pendant la détection d'intrusion. Cette base de signatures peut atteindre une taille de plusieurs Mo et ainsi réduire les performances du système lorsque la détection d'intrusion est active. Durant cette thèse, nous avons mis en place plusieurs méthodes pour réduire la mémoire nécessaire pour stocker les signatures, tout en préservant leur qualité.

La première méthode présentée est basée sur la décomposition modulaire des graphes. Nous avons utilisé cet outil de la théorie des graphes pour réduire la taille du graphe et, ainsi, diminuer le nombre de signatures, ainsi que leur longueur. Appliquée à des propriétés de confidentialité sur un système servant de passerelle, cette méthode divise par 20 le nombre de signatures générées.

La seconde méthode réduit directement la base de signatures en supprimant des signatures inutiles lorsque PIGA est en mode IPS. Appliquée sur les mêmes propriétés, cette méthode divise par 5 le nombre de signatures générées. En utilisant les deux méthodes, on divise le nombre de signatures par plus de 50.

Ensuite, nous avons adapté le mécanisme de détection afin d'utiliser les nouvelles signatures générées. Les expérimentations que nous avons effectuées montrent que notre système est équivalent à l'ancien système. De plus, nous avons réduit le temps de réponse de PIGA.

Mots clés : détection d'intrusions, décomposition modulaire, compression, signatures, sécurité système

Algorithmic approach for perfomance improvement of the intrusion detection system PIGA

PIGA is a tool for detecting malicious behaviour by analysing system activity. This tool uses signatures representing illegal behaviours that violate security properties defined in the policy. The signatures are generated from graphs modelling the operation between different system entities and stored in the memory during the intrusion detection. The signature base can take up several MB (Megabytes). This will reduce system performance when the intrusion detection is running. During this thesis, we set up two methods to reduce the memory used to store the signatures while also preserving their quality.

The first method is based on the modular decomposition of graphs. We used this notion of graph theory to reduce the size of the graph and lower the number and length of signatures. Applied to confidentiality properties on a gateway system, this method divides by 20 the number of generated signature.

The second method reduces directly the signature base by deleting useless signatures when PIGA is used as an IPS. Applied to the same properties, this method divides by 5 the number of generated signatures. Using both methods together, the number of signatures is divided by more than 50.

Next, we adapted the detection mechanism to use the new generated signatures. The experiments show that the new mechanism detects the same illegal behaviours detected by the previous one. Furthermore, we reduced the response time of PIGA.

Keywords : intrusion detection, modular decomposition, compression, signatures, system security

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS

Laboratoire d'Informatique Fondamentale d'Orléans Bâtiment IIIA Rue Léonard de Vinci B.P. 6759 F-45067 ORLEANS Cedex 2

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE