INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

PROJET APPLICATION

QUOTIENT GRAPH

Résumé

Policy Interaction Graph Analysis (PIGA) est un outil permettant de détecter et de prévenir les actions illicites dans un système d'exploitation. Et ce, à partir de suites d'interactions représentant un possible transfert d'informations, nommées signatures. Ces signatures étant nombreuses nous souhaitons en réduire leur nombre. Une fois cette réduction effectuée par la décomposition modulaire, nous devons trier le résultat. Pour cela nous utiliserons un algorithme de pivot afin de subdiviser le plus possible les tas obtenus.

Charles LE REUN – Jason NGOSSO – Cyril SEGRETAIN

3A STI – Juin 2014

Tuteur : Pascal BERTHOME

REMERCIEMENTS

Nous tenons à remercier tout d'abord M. Pascal BERTHOME pour nous avoir encadrés tout au long de la réalisation de ce premier projet d'application, mais aussi M. Pierre CLAIRET, auteur de la Compression de signatures pour PIGA IDS pour sa disponibilité. Enfin nous souhaitons remercier l'ensemble des trois auteurs de PIGA, M. Pierre CLAIRET, M. Pascal BERTHOME et M. Jérémy BRIFFAUT pour nous avoir partagé ces informations complètes sous différents formats, que ce soit le poster résumé ou la thèse complète et les fichiers issus de la décomposition modulaire.

INTRODUCTION

Avant de commencer, qu'est-ce que PIGA ? Il s'agit d'un outil permettant de détecter les comportements malicieux sur une machine, par analyse de trace. Plus simplement, PIGA va utiliser des signatures, autrement dit des références à une action violant une propriété de sécurité, afin de vérifier si un comportement au sein de la machine est illicite. Mais pour obtenir ces signatures il est nécessaire de passer par des graphes ; ici nous utiliserons la décomposition modulaire qui réduit significativement le nombre de signatures. En quelques mots il s'agit de savoir par où peuvent s'introduire des programmes malveillants afin de les contrer. Mais vous imaginez évidemment que ce nombre d'endroits est gigantesque et que le nombre de chemins les liant est d'autant plus grand qu'il y a d'accès. Or il faut bien stocker ces signatures en mémoire et les utiliser. La finalité de cet outil est de réduire ce nombre conséquent de signatures afin de gagner de la place et de permettre une utilisation moins gourmande en ressources physiques.

Ce dossier va présenter notre démarche de compréhension de l'outil PIGA, ainsi que ce qui en découle, telle que la Théorie des Graphes ou plus simplement la notion de Graphe Quotient. De plus, nous vous ferons part du travail que nous avons effectué de notre côté, suite à la décomposition modulaire. Et enfin comment interpréter les résultats retournés par notre programme.

Table des matières

1. Travail préliminaire

1.1. PIGA

Dans le cadre de ce projet, nous avons dû nous familiariser à certaines branches de la théorie des graphes, mais aussi à un système de détection d'intrusions précis : PIGA.

PIGA est un acronyme de Policy Interaction Graph Analysis qui signifie en français : L'interaction des Politiques d'Analyse Graphique. En somme il s'agit d'un IDS, ou Système de Détection d'Intrusions en français, développé durant la thèse de J. Briffaut au Laboratoire d'Informatique Fondamentale d'Orléans (LIFO).

Un IDS a pour but de détecter une intrusion effectuée par un utilisateur ou un service sur le système surveillé par l'IDS. Il existe deux types d'IDS : les systèmes de détection d'intrusion réseau et les systèmes de détection d'intrusion système. Ce dernier, utilisé par PIGA, analyse les données fournies par le système, comme les appels systèmes par exemple. Aussi, il existe deux principales méthodes de détection : par comportement et par signature. La détection par signature a pour but de détecter les attaques connues, en utilisant une base de signatures d'attaques. Mais cela nécessite une mise à jour fréquente de cette base de signatures car cette méthode ne permet pas de détecter les attaques inconnues. PIGA utilise une méthode se rapprochant de la détection par comportement, mais paramétrée par une politique qui considérera qu'il y a attaque s'il y a violation de cette politique. Il s'agit de la propriété de sécurité de PIGA. D'où la nécessité d'une parfaite connaissance du système sur lequel l'IDS est installé. De plus, la notion de confidentialité est tout aussi importante. En effet cette propriété doit empêcher les entités non autorisées à accéder à l'information confidentielle tout en permettant l'accès aux entités autorisées. Mais cette propriété va de paire avec celle d'intégrité, qui garantie l'impossibilité de modifier ou détruire une information, si elle est intègre.

Savoir qu'il existe des comportements allant à l'encontre des propriétés de sécurité c'est bien, pouvoir les identifier, c'est mieux. Mais pour cela, il faut pouvoir se les représenter, c'est ici le rôle que jouera la base des comportements malicieux. Cette base est un objet qui identifie l'ensemble des comportements illicites pour un système et une politique de sécurité donnés. Elle indique si le comportement est licite ou non. Le système de signatures de PIGA, générées à partir de plusieurs graphes représentant le système et la politique de sécurité, en est un exemple. Contrairement au système d'accès discrétionnaire (DAC), dans lequel chaque utilisateur définit les droits d'accès des objets qu'il possède. PIGA, elle, fonctionne comme une surcouche de contrôle d'accès mandataire (MAC) et permet donc de définir une politique d'accès qui ne peut être modifiée par les utilisateurs. Ce qui lui permet de récupérer des interactions, autrement dit l'ensemble des actions autorisées par le système.

Grâce aux vecteurs d'interactions, PIGA génère un graphe nommé graphe d'interaction qui représente l'ensemble des interactions autorisées. À partir des différents graphes, PIGA génère des signatures correspondant aux comportements violant les propriétés de sécurité décrites dans sa politique et les classe suivant le type de propriétés de sécurité. La base de signatures obtenue permet à PIGA de détecter les comportements violant une ou plusieurs signatures. Enfin, 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 étant 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.

Mais la base de signatures étant gigantesque, il est nécessaire de trouver des moyens de la réduire. Pour cela, deux approches, combinables, sont possibles. La première par modification des graphes utilisés lors de la génération de signatures. La seconde par suppression de signatures qui ne sont pas complétables à cause de la présence d'autres signatures dans la base.

1.2. Décomposition modulaire

Nous savons désormais qu'il est important de faire attention aux différents contextes de sécurité. Ces contextes sont représentés dans la théorie des graphes par des sommets. La décomposition modulaire est l'outil théorique permettant de mettre en évidence ces sommets sur des graphes quelconques.

Tout d'abord, qu'est-ce qu'un graphe quotient ? C'est un graphe qui va regrouper les sommets sous forme de tas, ayant comme point commun le type de contexte de sécurité. Nous vous proposons l'exemple ci-dessous afin que cela vous soit plus parlant.

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

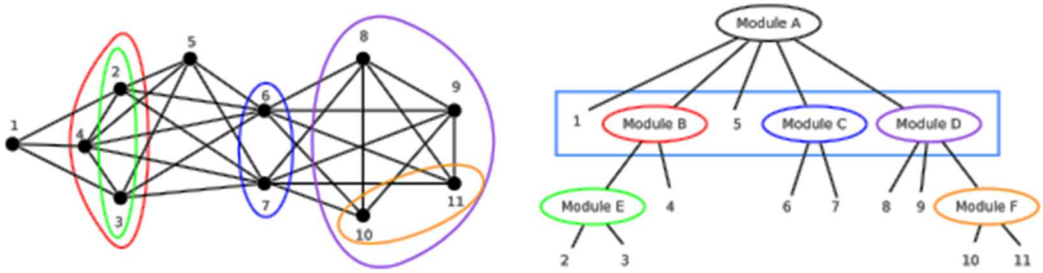
Même si le passage sous cette forme est indispensable pour une meilleure visualisation du système, cela met en avant la perte de deux types d'informations. Premièrement, sous cette forme il est désormais difficile de connaître précisément les interactions entre chaque tas, communément appelées arêtes dans la théorie des graphes. La deuxième découle directement de la première : nous n'avons plus connaissance des interactions précises à l'intérieur de chaque tas.

C'est essentiellement à cause de cette première perte d'informations que la décomposition modulaire est utilisée. En effet, le principe de cette décomposition est de regrouper les sommets ayant le même comportement vis-à-vis du reste du graphe. Si deux éléments d'une partition sont adjacents sur le graphe quotient, alors tous les sommets du premier élément sont adjacents à ceux du deuxième et réciproquement. De plus, chaque sous-ensemble est partitionné de manière récursive, suivant le type des éléments de chaque tas. Il est donc possible de connaître les arêtes entre deux éléments ou tas, d'un même graphe. Même s'il existe deux types de graphes, nous travaillerons, par commodité, sur les graphes non-orientés.

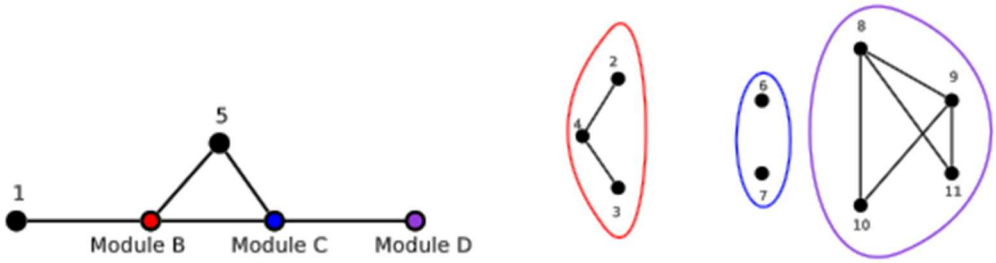
Le regroupement des sommets se base sur l'idée de module de graphes. Un module se base quant à lui, sur le concept d'uniformité et ce, pour un graphe non-orienté. En quelques mots, un sommet S est uniforme par rapport à un sommet x si x est voisin de tous les sommets de S ou qu'il est voisin d'aucun sommet de S. Un module correspond au sous ensemble des sommets uniformes, pour lesquels ce sous ensemble est inclus dans l'ensemble des sommets adjacents à un sommet précis au sein du graphe, ou bien inclus dans l'ensemble complémentaire de ce dernier. Autrement dit, l'idée de module d'un graphe est de regrouper les

sommets qui ont un comportement identique, vis-à-vis du reste du graphe, sous forme de méta-sommets.

Enfin la décomposition modulaire peut se représenter sous forme d'arbre. En effet, si un module ne chevauche aucun autre module, alors il sera fort. L'ensemble de ces modules forts peut donc être organisé sous forme d'un arbre, nommé arbre de décomposition modulaire et représente l'ordre d'inclusion des modules forts du graphe. Dans l'illustration ci-dessous, vous trouverez à gauche un graphe avec les modules forts et à droite son arbre de décomposition modulaire.



La partition modulaire maximale d'un graphe est l'unique partition de l'ensemble des modules composée des modules forts maximaux. Cette décomposition étant unique, il est possible de s'en servir pour générer un graphe quotient modulaire, en regroupant les sommets d'un même module maximal sous forme d'un méta-sommet, comme l'illustre l'image ci-dessous.



L'utilisation de la décomposition modulaire permet de diminuer la perte d'informations lors du passage au graphe quotient. Par définition même d'un module, 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.

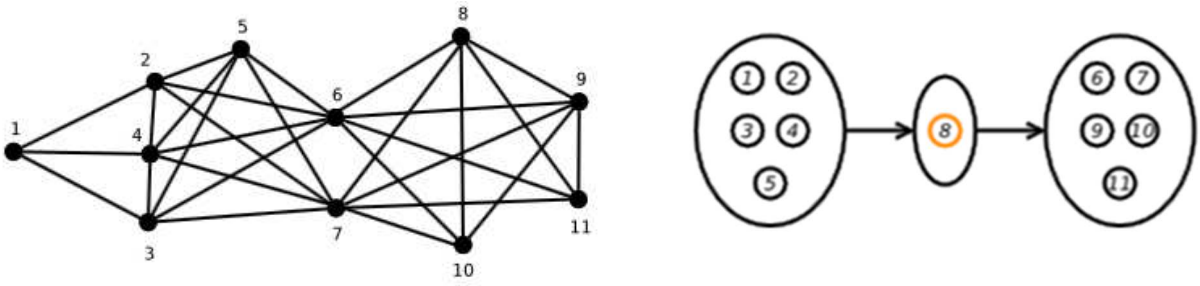
L'utilisation de la décomposition modulaire sur les graphes utilisés par PIGA permet d'obtenir un graphe quotient de taille réduite par rapport au graphe d'interactions initial. Les signatures étant des chemins dans les différents graphes utilisés par PIGA, la structure des modules implique la compression efficace des signatures originales, en limitant la sur-approximation.

1.3. La règle du pivot

La fonction *pivot* est la fonction qui va nous permettre de créer les tas. Cette fonction prend en paramètre la matrice générale créée précédemment et la matrice « tas » créée dans la fonction *centre*.

Étant donné qu'au retour de la fonction *centre* nous avons déjà 2 lignes dans notre matrice « tas », la fonction *pivot* va donc à chaque case testée se référer à la matrice générale pour voir quels sont les adjacents de la valeur de la case testée, et voir si ses adjacents se trouvent sur la même ligne de la case dans la matrice « tas ». Dans cette fonction on test bien évidemment toutes les cases de « tas » par ordre croissant (tas[0][0], tas[0][1], …, tas[0][j], tas[1][0], …) jusqu'à arriver à une première case d'une ligne vide (tas[i][0] est vide).

Voici le graphe exemple qu'on va utiliser pour montrer ce que fait la fonction *pivot*.



Comme vous pouvez voir sur l'exemple ci-dessus, notre première ligne de notre matrice « tas » serait le tas 1,2,3,4,5 et la deuxième ligne serait le tas 6,7,9,10,11, car sur cette exemple on a choisi pour centre la valeur « 8 ». Pour simplifier, nous allons utiliser un affichage des tas comme ci-dessous :

tas1

1	2	3	4	5

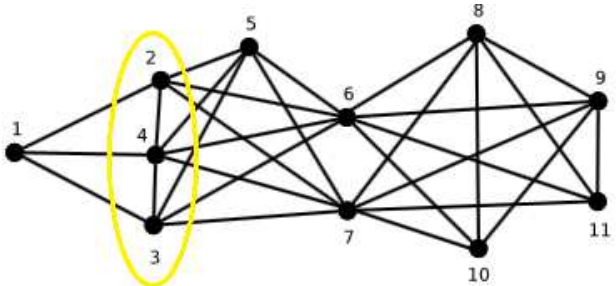
tas2

6	7	9	10	11

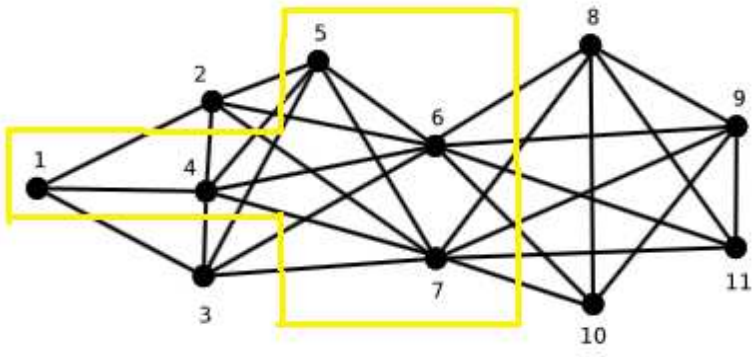
Ici on commencerait par testé « 1 ». Tous les adjacents de « 1 » qui sont 2,3,4 que l'on peut voir clairement sur les illustrations ci-dessous se trouvent sur la même ligne. Il n'y a pas de nouveau tas qui se crée, la règle du pivot n'a pas lieu d'être appliqué dans ce cas.

adjacent_1

2	3	4



Lorsque l'on tombe sur une valeur dont tous ses adjacents ne sont pas sur la même ligne (pour l'exemple on peut se référer à la valeur « 2 » ou « 3 » qui ont les mêmes adjacents, illustrés ci-dessous) le pivot peut alors être appliqué : après avoir regardé tous les adjacents de la valeur testée, on cherche la ligne de la matrice « tas » à laquelle se trouve l'adjacent qui n'est pas sur la même ligne de la valeur testée dans cette matrice.



adjacent_2

1	4	5	6	7

adjacent_3

1	4	5	6	7

Après avoir trouvée la ligne on teste toutes les cases de la ligne afin de trouver un ou plusieurs adjacents que l'on stockera dans un tableau provisoire. A la fin de ce test, on va comparer le nombre de cases que possède le tableau provisoire au nombre de cases que possède la ligne de la matrice « tas » que l'on a utilisé.

tab_provisoire

6	7

Si ces derniers sont égaux, cela signifierait que c'était déjà un tas finis donc à ce moment-là le pivot ne s'applique pas (c'est pour éviter une répétition en boucle).

Exemple pour la valeur « 3 » après que le pivot effectué avec la valeur « 2 » a généré un nouveau tas :

tas1

1	2	3	4	5

tas2

9	10	11

tas3

6	7

tab_provisoire

6	7

On voit bien ici que ce serait inutile de refaire un nouveau tas avec encore les mêmes valeurs que le tas3.

Mais par contre si elles ne sont pas égales alors on crée un nouveau tas composé des adjacents que l'on a stocké, puis on supprime les cases de la ligne de la matrice « tas » utilisée qui comporte les valeurs des adjacents isolés.

tas1

1	2	3	4	5

tas2

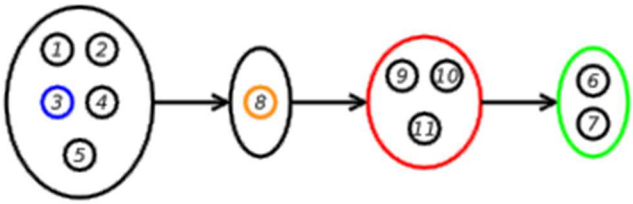
9	10	11

tas3

6	7

tab_provisoire

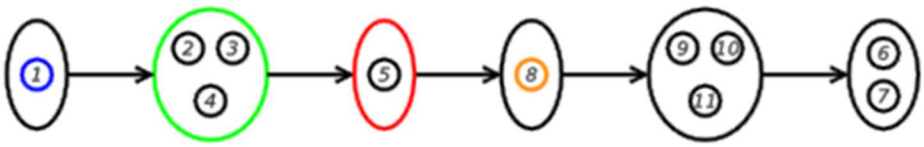
6	7



Puis on supprime tous les éléments du tableau provisoire, que le pivot ait été effectué ou non pour faire place à la nouvelle valeur testée.

tab_provisoire

Et enfin la fonction *pivot* prendra fin après qu'aucun élément d'un tas ne pourra en couper un autre, ce qui nous donnera ci-dessous dans les 2 vues en ce rapportant toujours à l'exemple cité :



tas1

1

tas2

9	10	11

tas3

6	7

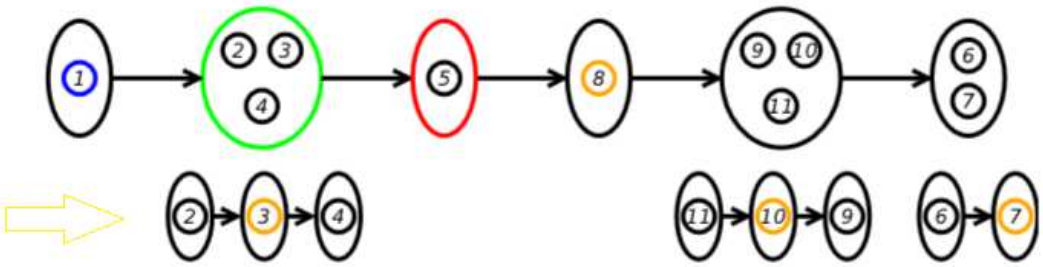
tas4

5

tas5

2	3	4

Notre fonction pivot s'arrête à ce niveau-là, si on veut encore casser des tas, il va falloir appliquer la fonction *centre* a l'intérieur d'un tas, pour nous retourner ceci (voir ci-dessous au niveau de la flèche) :



2. De la théorie à la pratique

2.1. L'utilisation du fichier d'entrée

Maintenant que vous vous êtes familiarisé avec le principe de la décomposition modulaire. Nous vous proposons de vous expliquer brièvement à quoi correspondent les éléments du fichier d'entrée que notre programme va traiter.

A la ligne 1, « sysadm_u:sysadm_r:bootloader_t » correspond à un sommet. A la ligne suivante puis une ligne sur deux, se trouvent les adjacents. Ici « sysadm_u:sysadm_r:bootloader_t », « system_u:system_r:bootloader_t » … ainsi de suite. Et ce, jusqu'à tomber sur une ligne vide, qui marque la séparation entre deux tas. Vous aurez aussi remarqué les lignes « file { write}; » qui correspondent à l'attribut liant le nœud à chaque adjacent au-dessus de cette fameuse ligne. L'image ci-dessous est une capture du début du fichier d'entré de notre programme.

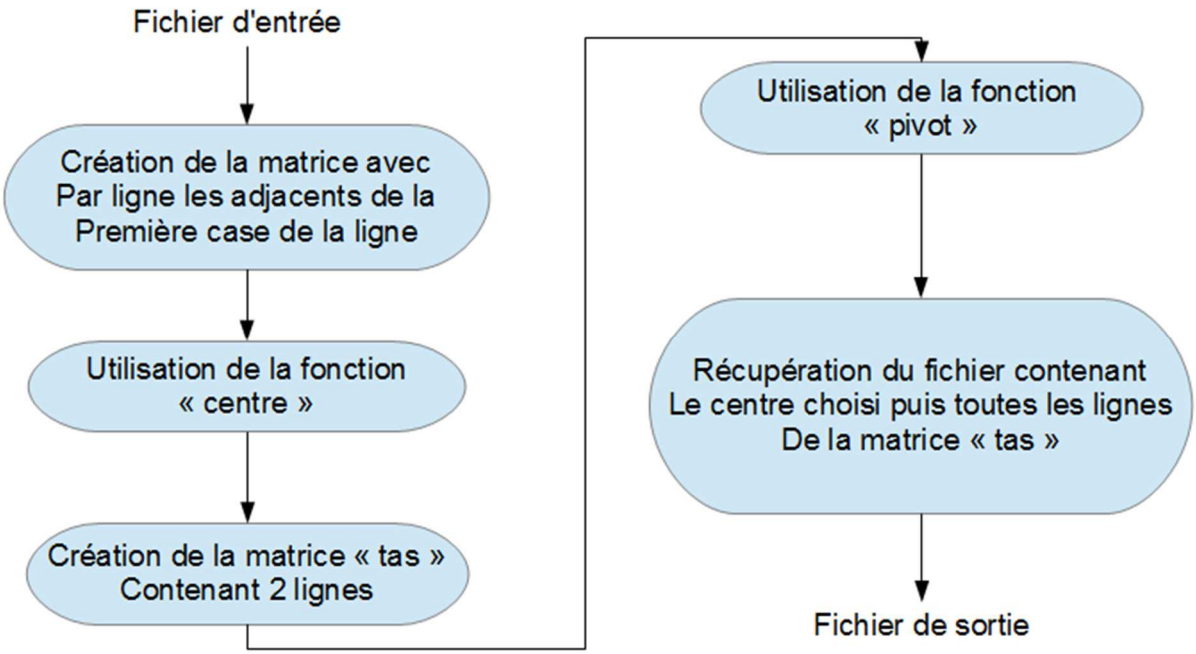
```
sysadm_u:sysadm_r:sysadm_dbusd_t
    sysadm_u:sysadm_r:sysadm_t
            file  { write};
    system_u:system_r:sysadm_t
            file  { write};
    root:sysadm_r:sysadm_t
            file  { write};
    staff_u:sysadm_r:sysadm_t
            file  { write};
    sysadm_u:sysadm_r:sysadm_dbusd_t
            file  { write};
    root:sysadm_r:sysadm_dbusd_t
            file  { write};
    staff_u:sysadm_r:sysadm_dbusd_t
            file  { write};
    system_u:system_r:sysadm_dbusd_t
            file  { write};
    system_u:system_r:init_t
            file  { write};
    sysadm_u:sysadm_r:sysadm_t
            file  { write};
    system_u:system_r:sysadm_t
            file  { write};
    root:sysadm_r:sysadm_t
            file  { write};
    staff_u:sysadm_r:sysadm_t
            file  { write};

sysadm_u:sysadm_r:run_init_t
    system_u:system_r:initrc_t
            file  { write};
    sysadm_u:sysadm_r:sysadm_t
            file  { write};
```

Ce fichier d'entré va correspondre dans notre cas d'utilisation à toutes les signatures générées par PIGA dans un système. Le but est maintenant d'appliquer l'algorithme de décomposition modulaire tiré de la thèse de Pierre CLAIRET afin d'améliorer la lecture d'un tel fichier.

2.2. Le cœur du programme

Comme le montre le schéma de notre algorithme, lorsque l'on récupère notre fichier d'entrée on crée la matrice principale définissant les adjacents grâce à la fonction creation_matrice(). Ensuite on choisit le centre grâce à la fonction centre(). Dès lors qu'on a notre centre on crée la matrice qui va avoir la valeur du centre comme première ligne, puis dans les 2 lignes suivantes d'une part les adjacents du centre et d'autre part les non adjacents du centre grâce à la fonction creation_table_tas(), où ces 2 dernières lignes vont devenir les 2 premières lignes de notre matrice « tas » que l'on va utiliser dans notre fonction pivot. Puis on utilise cette fonction pivot pour avoir tous les tas formés comme il se doit. Et donc au retour de cette fonction nous allons retrouver notre fichier de sortie avec la matrice contenant le centre puis tous les tas formés.



Une fois notre algorithme terminé, nous avons choisis un langage de programmation adapté à notre problématique : le Python. Ce choix s'explique tout d'abord par la simplicité de manipulation des chaines de caractères dans ce langage. D'autre part le python est un langage de haut niveau donc les conventions sont facilement utilisables. De plus ce langage est portable, ce qui était une nécessité dans notre cas car l'utilisation de notre programme n'est pas limitée à une plateforme.

2.3. Et en sortie

Comme expliqué dans les parties précédentes, nous pouvons voir dans ce fichier qui se réfère à l'exemple pris de la thèse la première ligne contenant le centre choisi (donc ici 8:8:8) puis tous les tas bien formés après avoir passé la fonction *pivot* au fichier d'entré (donc deuxième ligne le tas « 9:9:9, 10:10:10, 11:11:11 », troisième ligne « 1:1:1 » et ainsi de suite).

```
[cyril@new-host algo]$ cat fichier_sortie.txt
8:8:8
9:9:9 10:10:10 11:11:11
1:1:1
8:8:8
2:2:2 3:3:3 4:4:4
6:6:6 7:7:7
5:5:5
[cyril@new-host algo]$
```

3. Bilan

3.1. Difficultés rencontrées

Nous avons opté pour un code en Python afin de pouvoir traiter les chaînes de caractères facilement. Mais un problème majeur nous a rapidement bloqué lors du stockage du fichier dans notre liste nommée « matrice ». En effet en python le « EOF » du langage C n'existe pas. Nous avons tout d'abord tenté de faire comprendre au programme que la fin du fichier était un certain nombre de saut de lignes. Puis nous avons créé une fonction qui va écrire à la fin du fichier l'expression « End_Of_Line », mais en vain. Nous avons enfin utilisé une astuce basée sur une boucle « while » qui s'exécute tout le temps sauf quand la lecture d'une ligne n'est en fait pas une ligne. Ci-contre la partie du code qui nous a permis de terminer l'exécution de notre fonction à la fin du programme.

```
f=open('entree_graphe_these','r')
with f:
        while True:
                line=f.readline()
                if not line: break
```

La fonction pivot a également été une difficulté à laquelle nous avons fait face. Le plus dur dans cette fonction s'est fait ressentir au niveau des conditions. Cela a commencé dès la 1ère boucle de notre fonction qui consiste à tester chaque case de notre matrice « tas ». En effet, pour pouvoir effectuer le pivot il fallait déjà savoir quels étaient les adjacents de la valeur qu'on teste, puis de voir ou se situe ces adjacents dans notre matrice « tas » pour savoir si la valeur testée permettait bien de faire un nouveau tas. Il fallait donc penser à se référer à la ligne ou été contenue l'adjacent qui n'était pas dans le même tas. Il fallait aussi penser à une structure dans laquelle on pourrait stocker les adjacents qui ne se trouvent pas dans le même tas que la valeur testée, car essayer de séparer les tas directement étaient vite compliqués à faire (car il y aurait un gros risque de répétition des tas). De ce fait on a opté pour utiliser un tableau (le fameux tableau provisoire) pour stocker ces adjacents pour pouvoir former des nouveaux tas. On a trouvé plus simple de faire ça car il suffisait juste de *pop* les éléments du tableau provisoire dans une nouvelle ligne (tas) qu'on allait créer, puis de parcourir de nouveau la ligne où les adjacents (qui étaient aussi stockés dans le tableau provisoire) se trouvaient puis de les supprimer directement de la ligne (tas) tout en évidemment supprimant toutes les cases du tableau provisoire dans le cas où on n'a pas eu besoin de *créer* une nouvelle ligne (tas).

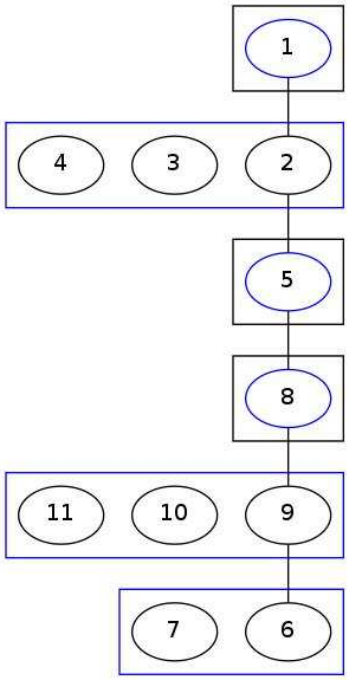
Enfin, lors de l'exécution de notre programme, sans qu'il y ait d'erreurs apparentes sur notre code nous avions des incohérences lors de l'affichage de notre liste principale, avant tout traitement. En effet le fichier de sortie fourni au début du projet était à certains endroits étranges. Plus précisément au début du fichier chaque tas était séparé d'un unique saut de ligne, puis au milieu du fichier par 2 lignes et parfois même 3 lignes vides, sans raisons apparentes. Nous avons donc dû changer notre façon de voir le ficher et donc nos conditions de changement de tas. Aussi, entre chaque adjacents était normalement écrit une ligne « file { write}; » mais parfois il n'y avait pas cette alternance normale, comme le montre l'image ci-contre.

```
system_u:system_r:mplayer_t
                file  { write};
sysadm_u:sysadm_r:mplayer_t
                file  { write};
                file  { write};
root:staff_r:mplayer_t
                file  { write};
                file  { write};
root:sysadm_r:mplayer_t
                file  { write};
                file  { write};
```

3.2. Prolongement

Après utilisation de notre programme, nous obtenons un fichier texte comme montré dans la partie précédente. Mais plutôt que l'utilisateur perde un temps précieux à comprendre le fichier et à la lecture du fichier, pourquoi ne pas utiliser une version graphique ?

Graphviz est un outil de programmation dédié aux graphes. L'autre groupe a travaillé sur cet aspect du projet. En utilisant le langage C, ils ont codé un programme utilisé par Graphviz afin d'obtenir un fichier image en sortie. Voici ce que donne l'utilisation de leur programme avec notre fichier de sortie.



CONCLUSION