Analyse morphologique en temps réel pour la détection de malware

Aurélien Thierry

Rapport de stage
Encadrant : Jean-Yves Marion (équipe CARTE, LORIA)

Analyse morphologique en temps réel pour la détection de malware

Aurélien Thierry

Nancy, juin 2011

1 Introduction

La rapide démocratisation de l'informatique a transformé notre rapport à l'ordinateur et aux systèmes d'information. Les premiers tendent à regrouper une vaste quantité de données personnelles sur leurs utilisateurs tandis que les systèmes d'information sont devenus une composante opérationnelle et décisionnelle nécessaire à toute organisation moderne. Ces systèmes apparaissant alors comme des cibles potentielles, des solutions ont été développées afin d'en protéger la confidentialité, l'intégrité et la disponibilité.

La première ligne de défense consiste à filtrer les entrées du système (ports usb, interfaces réseau, etc.) grâce à un pare-feu et à bloquer les menaces potentielles. Si cette défense échoue et qu'une ou plusieurs parties du système sont compromises, avoir défini et appliqué une politique de sécurité constitue une sécurité supplémentaire. En effet celle-ci restreint l'accès de chaque utilisateur, et donc des applications qu'il utilise, aux ressources du système. Ce cloisonnement permet de ralentir et limiter la compromission du système. Les détecteurs de logiciels malveillants, ou *antivirus* sont le dernier rempart dans cette approche. Ils vont permettre la détection et la classification des menaces présentes sur la machine, afin de stopper l'attaque.

La technique de détection la plus largement utilisée consiste à comparer un programme analysé à l'ensemble d'une base de logiciels malveillants connus. Elle dispose de nombreux avantages : elle est rapide et génère peu de faux positifs, c'est à dire de programmes légitimes faussement classés comme malveillants. Néammoins, se basant uniquement sur des attaques connues, elle n'est pas adaptée à la détection de nouvelles souches de programmes malveillants ni à la détection de techniques d'attaque innovantes. De ce fait, jusqu'à ajout dans la base de signatures, certains nouveaux malware ne seront pas détectés (faux négatif). Nous nous intéressons ici aux alternatives à cette approche par signature, et particulièrement à l'analyse morphologique des programmes en utilisant le graphe de flot de contrôle de celui-ci. Dans le but de détecter ou de stopper l'attaque avant l'infection de la machine, il serait souhaitable d'avoir conscience de l'arrivée d'un programme malveillant dès sa présence sur une entrée du système. L'objectif du stage est d'étudier la possibilité d'utiliser un détecteur morphologique au niveau réseau, puis d'en réaliser une implémentation sur un pare-feu. Pour ce faire, une étude de la complexité des algorithmes mis en œuvre est nécessaire afin d'améliorer les performances temporelles du détecteur.

Ce rapport est organisée comme suit. Une définition des logiciels malveillants selon leur charge active et leur forme est proposée dans la partie 2. Puis, dans la partie 3 nous étudierons l'approche par signature et ses limites, et proposerons différentes alternatives dont les analyses comportementales et morphologique. Les aspects théoriques et pratiques de cette dernière approche seront détaillés dans la partie 4. Enfin, dans la partie 5, des aspects d'optimisation de cette dernière approche portant sur le problème d'isomorphisme de sous-graphe seront développés.

Dans la mesure où ce rapport a été rendu après deux mois d'un stage de cinq mois, j'expliquerai comment je compte utiliser le travail déjà réalisé dans la suite du stage, et les axes à développer.

2 Terminologie des logiciels malveillants

Les travaux de Cohen [1] en 1986 et Adleman [2] en 1988 constituent les fondements de la virologie. Un virus, au sens de Cohen, peut être formalisé par un mot sur le ruban (zone mémoire) d'une machine de Turing, qui se duplique ou se modifie sur ce même ruban lorsqu'il est activé. La notion de réplication automatique est une caractéristique primordiale du virus. Adleman propose une notion plus générale d'infection informatique. La réplication n'entre pas dans sa définition qui est alors élargie à tout programme nuisible pour la machine ou l'utilisateur. Ses travaux ont été repris et complété par les auteurs de [3] qui établissement un cadre formel et les démonstrations des théories d'Adleman. Malgré un fondement théorique original solide, les approches fondamentales au problème des programmes malveillants sont assez rares. Dans son livre, Filiol [4] propose la définition 1 suivante pour une infection informatique ou *malware*.

Définition 1. *Programme simple ou auto-reproducteur, à caractère offensif, s'installant dans un système d'information, à l'insu du ou des utilisateurs, en vue de porter atteinte à la confidentialité, l'intégrité, ou la disponibilité de ce système, ou susceptible d'incriminer à tort son possesseur ou l'utilisateur dans la réalisation d'un crime ou d'un délit.*

Il est à noter que certains aspects de l'attaque virale comme le mode d'infection ne seront pas détaillés ici, mais ils sont abordés dans la littérature, notamment dans le livre de Szor [5].

3 Différentes méthodes de détection

Nous ne nous intéresserons ici qu'aux mécanismes de classement d'un fichier cherchant à déterminer s'il est bénin ou malveillant. En particulier, les aspects d'implémentation du programme d'analyse, le moment où le fichier est analysé (avant d'être modifié ou exécuté par exemple) comme les techniques d'optimisation utilisées (pour la recherche de signature par exemple) ne seront pas évoqués. Différents raisonnements ont été faits dans le domaine de la détection des logiciels malveillants et deux approches générales s'opposent. La première consiste en la recherche systématique de fichiers connus par avance et dont une signature a été extraite. Cette signature, comparée au fichier analysé, permet de déterminer si celui-ci est malveillant ou non. Cette technique est bien rodée et reste la principale méthode utilisée par les solutions commerciales. La seconde, encore principalement du domaine de la recherche, est une approche par comportement. Elle s'attache particulièrement à l'observation des actions entreprises par le programme analysé afin de déterminer s'il présente un risque ou non.

3.1 Analyse formelle du problème de détection

Le problème de détection des malware peut se formaliser. Nous présentons ici un aperçu de différents travaux dans ce domaine.

Les travaux fondateurs de Cohen [1] se basant sur la définition d'un virus comme programme auto-reproducteur sur une machine de Turing aboutissent

à un théorème fondateur. Celui-ci stipule que le problème de détection est indécidable. C'est à dire qu'il n'existe pas de programme capable sans erreur de décider si tel programme passé en entrée est malveillant. Et ce quelles que soient les connaissances à priori sur des virus ou des programmes bénins connus. Par conséquence, aucune approche, qu'elle soit par signature ou comportementale, ne peut être complète.

L'approche d'Adleman [2] aboutit au résultat suivant dans le cas plus général des infections informatiques. La détection de certains malware et de leurs variantes, est un problème Π_2 complet, c'est à dire soit non calculable, soit semi-calculable. Il s'agit de ce fait d'un problème plus difficile que celui de l'arrêt d'un programme, qui est déjà indécidable.

Le problème de la détection, quelle que soit l'approche théorique réalisée est donc, formellement, très difficile et souvent non calculable. Les approches pratiques ne peuvent donc pas être parfaites, même si de nombreuses voies d'amélioration seront étudiées.

3.2 Approche par expressions rationnelles

Une signature est simplement une suite d'assertions sur les octets constituant le code à analyser. Le processus de création des signatures nécessite une bonne connaissance du code analysé (binaire ou code interprêté) et de ce fait a recours à des analystes humains. A partir d'un programme connu comme malveillant, l'analyste tente d'extraire la plus petite suite d'octets caractérisant ce programme et éventuellement ses variantes. Cette signature doit être suffisamment discriminante afin de ne pas classer à tort un fichier sain comme malveillant. Une technique simple consiste à prendre une suite d'octets consécutifs dans le fichier. Une base de signatures est remplie avec les signatures des différents malware connus à détecter. Scanner un fichier revient à rechercher les différentes signatures au sein du fichier, classant le fichier comme malveillant lorsqu'une correspondance a été trouvée. Dans son livre [5], Szor établit un ensemble de variantes à cette signature standard dans le but d'atteindre les deux objectifs de minimisation des faux-positifs et la détection de variantes.

Utilisation de joker. En général, seules peu d'instructions sont déterminantes pour classer un programme ; il peut aussi s'agir d'une suite d'instructions disséminées dans le fichier (par exemple toutes les instructions modifiant la valeur d'un registre spécifique). Afin d'ajouter plus de souplesse au processus de détection, la notion de *joker* a été introduite. Tout octet remplit la condition ajoutée par un joker, et donc chaque signature définit un ensemble de chaînes la respectant. La figure 1 expose une partie d'un programme en assembleur dont la signature contient des joker. La colonne de doite indique les instructions en assembleur telles qu'interprêtées lors de l'analyse du programme, celle du milieu les octets (décomposés en code hexadécimal) constituant le fichier. Les chaînes *08 8d 57* ***ae*** *b0* ***14*** *cd 80 e8 c3* comme *08 8d 57* ***43*** *b0* ***db*** *cd 80 e8 c3*, si présentes dans un fichier, conduiront à la détection du virus. L'utilisation de joker peut être généralisée en utilisant à la place une expression rationnelle comme signature, permettant encore plus de souplesse.

Tolérance d'exceptions. Afin de détecter les variantes immédiates de certains virus, certaines signatures autorisent le fichier à différer légèrement de la

FIGURE 1 – Extrait de programme et sa signature

```
90 :   b0 3f          mov $0x3f,%al
92 :   41             inc %ecx
93 :   cd 80          int $0x80
95 :   89 fb          mov %edi,%ebx
97 :   8d 4f 08       lea 0x8(%edi),%ecx
9a :   8d 57 1a       lea 0x1a(%edi),%edx
9d :   b0 0b          mov $0xb,%al
9f :   cd 80          int $0x80
a1 :   e8 c3 ff ff ff call 69 <peer0+0x69>
a6 :   2f             das
a7 :   62 69 6e       bound %ebp,0x6e(%ecx)
aa :   2f             das
ab :   73 68          jae 115 <peer0+0x35>
```

Signature : 08 8d 57 ** b0 ** cd 80 e8 c3

signature définie. Un nombre de différences est défini. Par exemple, à partir de la signature à 6 octets *07 3d cd 20 e8 c3*, si l'on autorise deux différences, les chaînes *07* ***5c fd*** *20 e8 c3* comme ***13*** *3d cd 20* ***6b*** *c3* seront également reconnues.

Cette approche par signature présente des avantages en termes de vitesse de détection et induit peu de faux positifs. De plus elle est indépendante du type de malware analysé. Nous verrons cependant par la suite les limites de ce système.

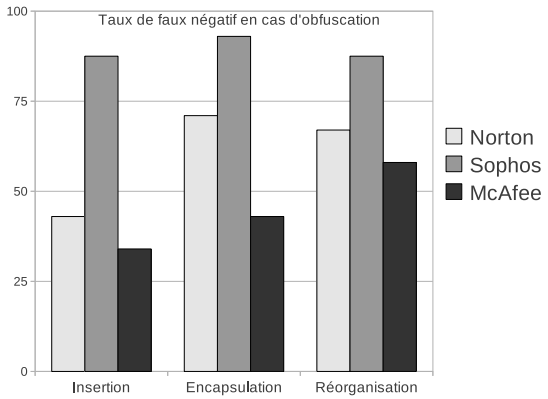
3.3 Limites de l'approche par signature

L'analyse de produits du commerce faite dans [6] en 2004 montre l'utilisation massive faite de cette approche ainsi que son manque de robustesse face à des techniques simples d'obfuscation. L'obfuscation de code consiste à appliquer des transformations sur le code afin de le rendre suffisamment différent du code original pour qu'il ne soit plus détecté par un antivirus reconnaissant l'original. Ces transformations doivent cependant respecter l'exécution du programme initial. En particulier l'ensemble des sorties du programme original doit être inclus dans celui du programme obfusqué.

Insertion de code inutile. Cela consiste à intercaler simplement dans le fichier à obfusquer des instructions supplémentaires sans modifier le comportement du programme initial. Ces ajouts peuvent être faits dans un code de haut niveau comme dans un programme binaire, seule la syntaxe des instructions ajoutées changeant.

Réorganisation du code. Cette opération revient à modifier l'ordre physique du code sans modifier son ordre d'exécution. En assembleur, l'ajout d'instructions de saut inconditionnel le permet.

FIGURE 2 – Évaluation de l'approche par signatures, exemples d'obfuscation



Encapsulation du code. Il s'agit d'écrire les instructions sous une forme ne dévoilant pas leur sens. Elle peut être réalisée à l'aide d'un encodage particulier (en hexadécimal par exemple), de compression (au format ZIP par exemple), ou de chiffrement. Le cas des packers permettant le chiffrement du programme n'est pas traité par les auteurs de [6]. Ces transformations modifient grandement la syntaxe du programme sans en modifier les instructions qui seront exécutées au final. Souvent une partie du programme sera dédiée à l'interprétation de la partie obfusquée.

Les auteurs ont ensuite généré, à partir de 8 virus connus, entre 31 et 5 592 variantes. Les antivirus testés sont Norton Antivirus, Sophos Antivirus, et McAfee Virus Scan. Les versions évaluées sont les dernières disponibles (Juillet 2004), avec leurs signatures de virus mises à jour. Leurs résultats sur chacune des techniques d'obfuscation sont les suivants.

Les résultats obtenus présentés dans la figure 2 donnent le pourcentage de virus non détectés après obfuscation pour chaque antivirus. Il est à noter qu'avec les deux méthodes simples d'obfuscation que sont l'insertion de code et la réorganisation, le pourcentage de faux négatifs est très élevé. Le cas de la réorganisation est particulièrement intéressant. En effet, les taux de faux négatif sont de 67%, 88%, et 58% alors que la transformation effectuée ne modifie ni les instructions ni leur ordre réel d'exécution.

Il apparaît donc clair que la méthode de détection par signature simple telle qu'elle est largement pratiquée encore aujourd'hui n'est pas suffisante. Cet effet est d'autant plus gênant que ces modifications du code peuvent être faites de manière automatique par un virus cherchant à se rendre furtif aux yeux d'un antivirus. Le temps nécessaire à la génération et la diffusion de la signature de chaque nouvelle souche laisse au virus une fenêtre temporelle suffisamment large pour se répandre. Le papier [6] datant de 2004 justifie la nécessité de développer de nouvelles méthodes de détection, et explique donc l'abondance de recherches en ce sens ces dernières années.

3.4 Approche comportementale

Les méthodes de détection dites comportementales cherchent à caractériser les actions *normales* pour un type de programme étudié. Deux approches sont donc possible. La première tente de modéliser les comportements malveillants et mesure l'écart entre le programme évalué et les modèles connus. La seconde au contraire consiste à définir un ensemble de profils correspondant à des programmes légitimes (client mail, navigateur internet, etc.) à l'aide d'outils statistiques et détermine l'écart avec le programme testé. Les modèles basés sur des profils d'applications légitimes sont complexes à mettre en oeuvre, par la grande diversité d'applications de différentes natures sur un système. Ils sont de fait très sensibles aux faux positifs et dépendant de l'environnement sur lequel ils sont déployés. C'est pour ces raisons que nous ne détaillerons ici que la première approche. Néammoins, étant basée sur le comportement des malware connus, elle n'est pas adaptée pour détecter des logiciels malveillants utilisant des techniques innovantes. Le risque ici concerne le nombre possiblement grand de faux négatifs.

Les auteurs de [7] se sont employés à extraire la structure d'un détecteur comportemental, ainsi que les différentes possibilités explorées jusqu'ici. Trois tâches consécutives sont nécessaires. La première consiste à récupérer une base de malware et à en extraire les instructions ou les actions qu'ils réalisent. La seconde transforme ces données en données intermédiaires, plus abstraites. Le dernier composant contient l'algorithme de détection indiquant si tel programme analysé est malveillant.

3.4.1 Collecter des malware

La collection de malware utilisée peut provenir de deux sources. Ils peuvent être récupérés sur les ordinateurs personnels de clients (un antivirus peut avoir une fonction d'envoi d'un fichier suspect pour analyse), ou provenir de *pots de miel*. Un pot de miel ou *honeypot* est un système volontairement vulnérable, dans le but de subir des attaques susceptibles d'être par la suite analysées afin de mieux connaître l'attaquant (ici les malware).

3.4.2 Extraction et analyse des actions d'un malware

Pour extraire les actions du programme étudié, il est possible de n'effectuer qu'une analyse statique, mais on cherche souvent à obtenir plus d'informations. Dans la plupart des cas, le programme est exécuté, soit sur une machine physique en enregistrant les appels système, soit sur une machine virtuelle simulant un environnement réel. Ce deuxième cas est celui rencontré le plus souvent car il permet l'analyse du malware sans risquer de compromettre la machine physique. L'analyse dynamique n'est pas toujours réaliste car certains malware parviennent à détecter qu'ils sont exécutés sur une machine virtuelle et se comportent alors différemment, rendant leur analyse inefficace [8] .

Les actions sont rendues moins spécifiques en les regroupant par catégories. Certaines opérations sont plus sensibles que d'autres. On peut citer l'accès à des secteurs système (partition ou démarrage), ou des références vers une zone mémoire réservée au noyau.

3.4.3 Algorithme de reconnaissance des malware

Le composant réalisant la détection peut implémenter différents alogorithmes d'évaluation de la menace. Deux de ces alogorithmes sont présentés ci-dessous.

Algorithme par seuil. Une stratégie simple consiste à extraire des malware récoltés les actions les plus susceptibles d'être malveillantes. Une coefficient est affecté à chaque action, et un *score de malveillance* peut être calculé pour chaque programme à analyser en effectuant la somme des coefficients associés à chaque action qu'il entreprend. Si ce score dépasse un certain seuil, le programme sera détecté comme malveillant. Une étude comparative de logiciels légitimes peut être bénéfique pour éviter un taux trop élevé de faux positifs car beaucoup d'actions sont fréquentes pour les malware comme pour les programmes légitimes et donc ne doivent pas rentrer excessivement en ligne de compte.

Algorithme par automate. Cette approche s'intéresse plus à des séquences d'actions. Des profils malveillants vont être établis et un langage leur est associé. Pour permettre la détection de ces profils, des automates sont utilisés. Un programme sera considéré comme malveillant si, partant d'un état initial d'un des automates, la séquence des actions qu'il entreprend le fait arriver dans un état final de cet automate. Cette méthode tend à considérer un malware comme effectuant une suite d'actions, malveillantes ou non, qui, effectuées séquentiellement, ont un comportement malveillant.

3.4.4 Résultats

Certains travaux montrent l'efficacité de l'approche comportementale. Pour la détection d'une catégorie particulière de logiels espions par exemple [9]. Les auteurs se sont intéressés aux programmes s'intégrant dans un navigateur internet (Internet Explorer sous Windows) et offrant des fonctionnalités supplémentaires (barres d'outils intégrées, etc.). Certains de ces programmes tiers sont en fait des logiciels destinés à collecter des informations de navigation des utilisateurs pour leur proposer de la publicité ciblée. Ils ont caractérisé un logiciel espion par l'observation des sites visités et de leur contenu via Internet Explorer, suivie de l'appel à des API Windows destinées à enregistrer (dans un fichier) ou envoyer (à un serveur distant) ces informations. Afin de détecter ces comportements, ils ont combiné analyse statique et analyse dynamique. Dans ce cas, et sur 51 fichiers étudiés (dont 33 sont malveillants), leur approche a un taux de détection optimal (100% des 33 malware sont détectés) et un taux de faux positifs faible (deux logiciels bénins sont classés comme malveillants).

Il apparaît ici que l'analyse des programmes en se basant sur leur comportement plutôt que sur des signatures est prometteuse et donne déjà des résultats probants dans certains cas. Dans le paragraphe suivant, nous aborderons une approche originale, par analyse morphologique des malware. Cette approche repose sur un nouveau type de signatures, par le graphe de flot de contrôle. Ce graphe est construit à partir des instructions exécutées par le malware.

4 Analyse morphologique : de la création du CFG à sa détection

L'analyse morphologique des virus se base sur la reconnaissance de graphes de flot de contrôle (*control flow graph*, ou CFG) de virus connus. En ce sens il s'agit d'une approche par signatures. Dans cette partie, nous allons définir le champ d'application de cette étude, les graphes de contrôle de flots, les aspects théoriques et pratiques de leur détection, et discuter les résultats déjà obtenus avec cette méthode. Nous verrons alors en quoi cette approche se place entre une approche standard par signature et une approche comportementale.

4.1 Cadre de l'étude

L'analyse se réduira ici aux programmes non structurés tels les exécutables en code assembleur x86. La principale difficulté dans leur analyse réside dans l'existence d'instructions de saut dynamiques (instruction *jmp* par exemple). Un graphe de flot de contrôle est un graphe orienté dans lequel les nœuds sont les adresses des instructions et les arcs représentent les différents chemins que l'exécution peut suivre.

Le graphe de flot de contrôle peut-être obtenu de deux manières. La première consiste à réaliser une analyse statique (après désassemblage du programme), on a alors un CFG qui peut être incomplet dans le cas où des adresses cible de saut dynamiques n'ont pas pu être déterminées. C'est notamment le cas lors de l'utilisation d'un packer qui va déchiffrer les instructions, donc le programme, au fur et à mesure de son exécution. La seconde approche consiste à faire une analyse dynamique du programme en instrumentant le code (à l'aide de *Pin Tool* par exemple [10]) tandis qu'il est exécuté dans une machine virtuelle. La trace ainsi obtenue est ensuite transformée en CFG. Cette approche donne un graphe de flot de contrôle dans tous les cas, mais ce graphe n'est pas forcément représentatif du programme analysé dans la mesure où elle ne donne qu'une exécution donnée, dans un environnement fixé, du programme. Si la charge virale ne s'active qu'à une certaine date par exemple, le CFG ne sera caractéristique d'aucune action malveillante.

Dans la suite, nous ne nous intéresserons pas à la manière dont les instructions sont obtenues (analyse statique ou dynamique), mais seulement à la construction du CFG et à sa reconnaissance.

4.2 Construction du CFG

L'étude faite dans [11] introduit un langage modélisant l'assembleur x86. Il différentie principalement deux types d'instructions. D'une part les instructions de flot ont une influence sur le graphe de flot et sont au nombre de 4 (dans un jeu d'instructions x86 réel, il y a un grand nombre de variante au saut *jcc* mais elles ont toutes la même influence sur le CFG). D'autre part les instructions purement séquentielles n'ont pas d'influence sur le graphe de flots. De ce fait leur présentation ne cherche pas à être exhaustive. Dans ce modèle, un programme est simplement une suite d'instructions.

Adresses	$\mathbb{N}$
Offset	$\mathbb{Z}$
Registres	$\mathbb{R}$
Expressions	$\mathbb{Z}\|\mathbb{N}\|\mathbb{R}\|[\mathbb{N}]\|[\mathbb{R}]$
Instructions de flot	$\mathbb{I}^f ::= jmp\ \mathbb{E} \mid call \mid ret \mid jcc\ \mathbb{Z}$
Instructions séquentielles	$\mathbb{I}^d ::= mov\ \mathbb{E}\ \mathbb{E} \mid comp\ \mathbb{E}\ \mathbb{E} \mid ...$
Programmes	$\mathbb{P} ::= \mathbb{I}^d \mid \mathbb{I}^f \mid \mathbb{P};\mathbb{P}$

Chaque instruction se caractérise par son nombre de successeurs.

- Les instructions séquentielles ont unique successeur, il s'agit de l'adresse de l'instruction suivante.
- L'instruction de saut inconditionnel *jmp* a un unique successeur : l'adresse de destination du saut.
- L'instruction de saut conditionnel *jcc* a deux successeurs : l'adresse du saut dans le cas où la condition est vérifiée, et celle de l'instruction suivante dans le cas contraire.
- L'instruction *call* a deux successeurs : l'adresse de la fonction à appeler et l'adresse de retour (celle de l'instruction qui suit).
- L'instruction *end* n'a pas de succeseur. C'est ici que le programme (ou la fonction) s'arrête.

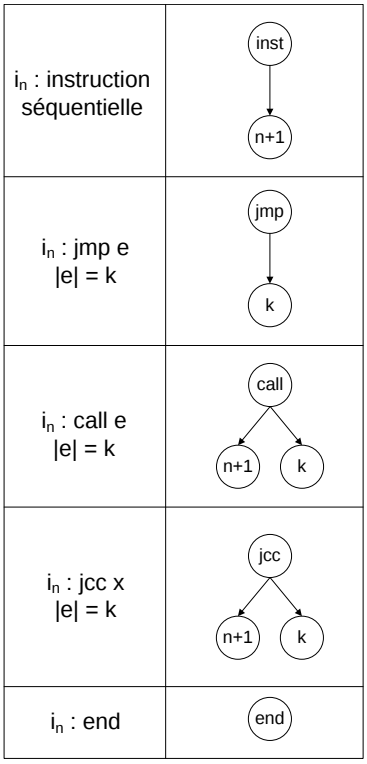
La figure 3(a) présente ces cinq cas et leurs influences sur le CFG selon l'instruction i_n à la ligne n. Lorsqu'un paramêtre e d'une instruction définit la cible d'un saut dynamique, il peut être évalué à l'aide d'une méthode d'analyse statique à $k = |e|$ afin de connaître si possible la ligne (et l'instruction correspondante) cible du saut. Dans de nombreux cas, une telle analyse n'est pas faite ou n'est pas probante, on note alors $|e| = \perp$ et on considère la cible du saut comme inconnue.

4.2.1 Normalisation : protection contre l'obfuscation

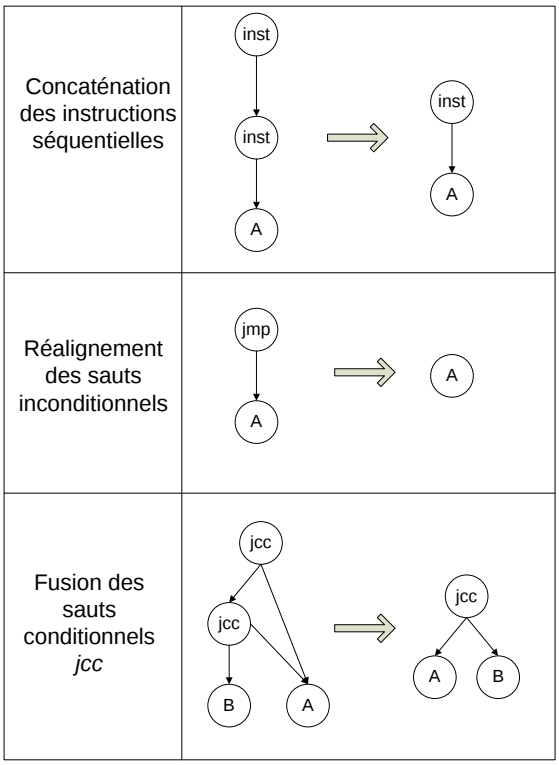
Une fois un premier graphe réalisé, il est utile d'en normaliser le résultat afin que les suites d'instructions ayant le même comportement aient la même représentation. Ces transformations rendent le CFG moins dépendant de l'agencement des instructions dans le code, et donc fournissent une protection contre des techniques habituelles d'obfuscation. Les opérations effectuées sont les suivantes.

- Concaténer les instructions séquentielles consécutives en un seul bloc.
- Réaligner le code en supprimant les sauts inconditionnels.
- Fusionner des sauts conditionnels consécutifs.

La figure 3(b) présente la réécriture du graphe effectuée sur le CFG. La figure 4 présente un exemple de programme en assembleur (à gauche), sa représentation initiale en graphe de flot de contrôle (au centre), et le CFG réduit (à droite). Les seules instructions ayant une influence sur le graphe de flot sont aux lignes 1, 6, 7 et 9. L'instruction *jne* est une variante du saut conditionnel *jcc*. La seule réduction possible consiste à regrouper les instructions séquentielles 3, 4 et 5.



(a) Construction du CFG



(b) Techniques de normalisation

FIGURE 3 – Construction et normalisation du graphe de flot de contrôle

FIGURE 4 – Un programme avec son CFG initial, et le CFG réduit

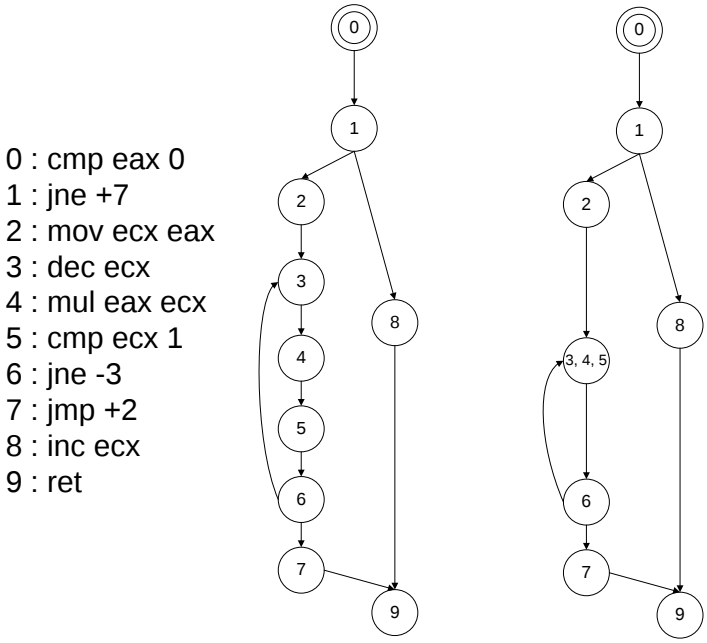


FIGURE 5 – Exemple de code avec instruction inutile (ligne 11)

```
1   assign x := x mod 3
2   assign x := 10(x+1)
3   cgoto(x)
10  stop
11  goto 2
20  stop
30  stop
```

4.2.2 Raffinement par analyse statique

Le graphe de flot peut être dérivé directement du code binaire dans des cas simples. Cela dit, très souvent, et d'autant plus si des méthodes d'obfuscation sont employées, il devient nécessaire d'avoir une connaissance à priori sur le flot d'exécution du code. Pour cela, les approches effectuées dans [11] et [12] préconisent le recours à l'analyse statique. Le but de cette analyse est d'identifier les instructions qui ne sont pas accessibles, de simplifier certaines suites d'instructions dans le but d'évaluer les conditions de transition de flot de contrôle pour supprimer les branches inutiles, et au final d'approximer au mieux le comportement réel du code avant de construire le graphe de flot.

Une approche classique s'intéresse aux valeurs des variables tout au long du programme. Soit le programme de la figure 5, une approche par valeurs ne s'intéressant qu'aux bornes considèrera l'instruction de la ligne 11 comme exécutable et le CFG en prendra compte car après la ligne 1, $x \in [0, 2]$ et après la ligne 2, $x \in [10, 30]$. Cela dit, compte tenu des valeurs possibles de x, cette instruction n'est pas atteignable. L'approche faite dans [13] est par raffinement successif. Dans une première approximation, les valeurs possibles de x ne sont pas connues, et donc l'analyse n'est pas suffisante pour déterminer si l'instruction de la ligne 11 est utile ou non. Les valeurs de x seront donc recherchées, en remontant à partir de l'instruction de la ligne 11 via les possibles instructions qui la précédent. De cette manière, il sera déterminé que $x \in \{10, 20, 30\}$ et donc que cette instruction est inutile. Le graphe de flot de contrôle est donc simplifié.

Bien que ce ne soit pas l'objet de cette étude, il est ainsi à noter que la précision du graphe de flot de contrôle dépend grandement de la qualité de l'analyse statique effectuée au préalable. Nous chercherons par la suite à comparer un programme analysé à l'ensemble de la base de malware connus, dont le CFG a été extrait.

On est à présent capable de représenter un programme sous la forme d'un CFG réduit dont on va se servir pour d'une part remplir une base de malware et d'autre part pour reconnaître un fichier infecté en le comparant à cette base. Les programmes que l'on souhaite reconnaître sont ceux qui intègrent le comportement d'un malware de la base. En termes de graphes de flot de contrôle, on cherche donc à savoir si le CFG réduit d'un des malware de la base représente un sous-graphe du CFG réduit du programme à analyser. Dans la prochaine partie, je présenterai les aspects théoriques des problèmes d'isomorphisme de graphe et

de sous-graphe, puis quelques algorithmes de résolution. Enfin je détaillerai l'approche effectuée jusqu'ici pour le détecteur et les pistes d'améliorations étudiées au cours du stage.

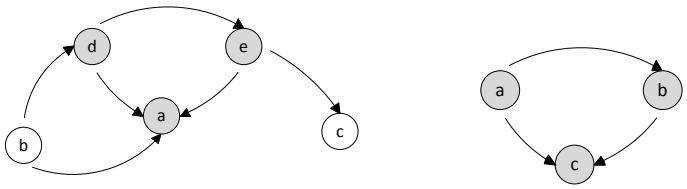
5 Détection de sous-graphes

5.1 Graphes et sous-graphes

Les notations et définitions standard d'un graphe ainsi que certains résultats généraux proviennent du livre *Graph Theory and its applications* [14]. Le problème de détection d'isomorphismes de graphes et de sous-graphes peut se poser de la manière suivante. Les graphes orientés sont définis comme suit (définition 2) :

Définition 2. *Un graphe orienté est G=(V, E) est composé par les deux ensembles finis V et E. V est l'ensemble des sommets de G, E est l'ensemble des arcs de G, c'est à dire un ensemble de couples (t, h) où t et h sont deux sommets distincts de G.*

Deux graphes orientés sont isomorphes s'ils vérifient les propriétés de la définition 3. On définit le sous-graphe d'un graphe G selon la définition 4, il est important de noter qu'un sous-graphe a des sommets pris arbitrairement au graphe de base mais que ses arcs sont exactement tous ceux du graphe de base qui s'appliquent aux sommets choisis. Notre problème consiste donc, à partir d'un graphe à analyser, à déterminer si un de ses sous-graphes est isomorphe à un des graphes de la base de malware. Intuitivement, et comme sur la figure 6, on cherche à savoir si le graphe H peut-être vu comme identique à un sous-graphe de G en préservant sa structure. Ici le graphe H est isomorphe au sous-graphe constitué des sommets $\{d, e, a\}$ comme à celui formé par les sommets $\{b, d, a\}$.



(a) Graphe G

(b) Graphe H, isomorphe à un sous-graphe de G

FIGURE 6 – Exemple de graphe et d'un sous-graphe

Définition 3. *Soit G et H deux graphes orientés. Ils sont dits isomorphes s'il existe une bijection* $f : V_G -> V_H$ *entre les sommets de G et ceux de H et si f préserve la structure d'adjacence des graphes, c'est à dire si*
$\forall u, v \in V_G, (u, v) \in E_G \Leftrightarrow (f(u), f(v)) \in E_H$.

Définition 4. *Un graphe orienté $H = (V_H, E_H)$ est un sous-graphe du graphe $G = (V_G, E_G)$ si et seulement si $V_H \subset V_G$, et $E_H = \{(t,h) \in E_G / t, h \in V_H\}$*

5.2 Représentation sous forme matricielle

Afin de mieux appréhender les problèmes liés aux graphes et pour faciliter l'utilisation de notions d'algèbre linéraire dans leur résolutions, il est pratique de les écrire sous forme de matrices. On définit ainsi la matrice d'adjacence M d'un graphe G suivant les arcs qu'il contient en ordonnant ses sommets selon la définition 5. On commence par ordonner les sommets $V = \{v_1, v_2, ..., v_n\}$ du graphe, puis s'il y a un arc du sommet v_i vers le sommet v_j alors $M_{i,j} = 1$, sinon $M_{i,\ j} = 0$.

Définition 5. *Soit G=(V, E) un graphe à $n \in \mathbb{N}$ sommets. La matrice d'adjacence M associée à G est la matrice $n*n$ définie par :* $M_{i,j} = \begin{cases} 1 & si\ (i,j) \in E \\ 0 & sinon. \end{cases}$

Par exemple, les graphe H et G de la figure 6 ont pour matrice d'adjacence les matrices de la figure 7.

$M_{i,j} =$

	a	b	c	d	e
a	0	0	0	0	0
b	1	0	0	1	0
c	0	0	0	0	0
d	1	0	0	0	1
e	1	0	1	0	0

(a) M_G

$M_{i,j} =$

	a	b	c
a	0	1	1
b	0	0	1
c	0	0	0

(b) M_H

FIGURE 7 – Matrices d'adjacence des graphes G et H

Le problème de l'isomorphisme peut alors se reformuler. Deux graphes G et H sont isomorphes si et seulement si il existe une permutation σ des sommets du graphe G telle que $M_{\sigma(G)} = M_H$. Il est donc nécessaire et suffisant qu'il existe une matrice de permutation P telle que $M_H = P.M_G.P^t$ (car $P^{-1} = P^t$ pour une matrice de permutation).

Pour l'isomorphisme de sous-graphe on cherche un sous-ensemble $\{s_1, s_2, ..., s_m\}$ de m sommets avec $m = Card(V_G)$ de G tel que le sous-graphe qu'il définit soit un isomorphisme du graphe H. C'est à dire une permutation σ de $\mathfrak{S}(n)$ avec $n = Card(V_G)$ telle que $\forall i \leq m,\ \sigma(s_i) \leq m$. Les autres éléments n'ont pas d'importance puisqu'ils ne participent pas à l'isomorphisme recherché. Soit P la matrice de permutation associée à σ. La matrice transformée $M_{\sigma(G)} = P.M_G.P^t$ est identique à M_H pour les sommets qui participent à l'isomorphisme, soit les m premiers sommets. Il suffit alors de réduire la taille de la matrice en utilisant une matrice $I^{n,m}$ définie par $I^{n,m}_{i,j} = \begin{cases} 1 & si\ i = j \\ 0 & \text{sinon} \end{cases}$.

Dans ce cas, et si on pose $W = (I^{n,m})^t.P$ avec $W \in \mathbb{M}_{n,m}$, on a bien $M_H = W.M_G.W^t = I^t_{n,m}.P.M_G.P^t.I_{n,m}$.

Dans l'exemple de nos graphes G et H, H est isomorphe au sous-graphe de G composé des sommets $\{d, e, a\}$ en faisant correspondre les couples (a, c), (d, a) et (e, b) entre G et H, soit la permutation σ telle que $\sigma(1) = 3$, $\sigma(4) = 1$ *et* $\sigma(5) = 2$. La matrice de permutation P est donnée en figure 8(a). La transformation qu'elle

induit est donnée par la matrice $M_{\sigma(G)}$ en figure 8(b). Il paraît alors évident qu'en ne prenant que la sous-matrice 3*3, on obtient M_H, opération réalisée à l'aide de $I_{n,m}$ (figure 8(c)).

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(a) Matrice de permutation P

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(b) $M_{\sigma(G)}$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

(c) $I_{n,m}$

FIGURE 8 – Matrices pour la démonstration sur l'exemple de G et H

On peut ainsi montrer que H est isomorphe à un sous-graphe de G si et seulement si il existe une matrice $W \in \mathbb{M}_{n,m}$ vérifiant $M_H = W.M_G.W^t$.

Il apparaît clair que l'approche par force brute qui consiste à essayer toutes les possibilités d'isomorphismes demande de parcourir toutes les permutations possibles d'une matrice n * n. Elle est donc en $O(n!)$ puisqu'il y a un nombre factoriel de matrices de permutation n * n. Une telle approche dans la recherche d'isomorphisme de sous-graphe serait encore moins efficace car il faudrait rechercher un isomorphisme entre tout sous-graphe d'un graphe d'une part, et un autre graphe fixe d'autre part. D'autant que cette détection de sous-graphe devra être effectuée pour chaque graphe contenu dans la base de malware. Nous nous intéresserons maintenant aux aspects de calculabilité des problèmes d'isomorphime de graphe et de sous-graphe, puis à différents algorithmes proposés pour leur résolution, et à leur complexité.

5.3 Calculabilité

Le problème de l'isomorphisme de sous-graphes est NP-complet.

Il a été montré que le problème de l'isomorphisme de graphes est un problème NP qui n'est pas NP-complet [15]. Cependant dans de nombreux cas particuliers il existe des algorithmes polynomiaux (classe P), comme nous le verrons par la suite.

5.4 L'algorithme d'Ullman

L'algorithme le plus connu pour résoudre le problème d'isomorphisme de sous-graphe est celui d'Ullman [16]. Il repose sur le concept de retour sur traces (ou *backtracking*) pour trouver successivement tous les sous-graphes isomorphes au graphe de base. Cette approche a ensuite été largement améliorée par Ullman à l'aide de l'utilisation d'une technique de vérification à priori permettant d'éliminer de nombreux candidats sans avoir à les tester entièrement.

5.4.1 Backtracking simple

Soit $G = (V_G, E_G)$ un graphe et $H = (V_H, E_H)$ un graphe dont on cherche à connaître les sous-graphes de G avec lesquels il est isomorphe. On note $m = Card(V_G)$ et $n = Card(V_H)$. L'idée consiste à partir d'un sommet de H, à

l'associer à un sommet de G, de vérifier si l'association créée est bien un isomorphisme de sous-graphe. Si c'est le cas, on prend un autre sommet de H à associer avec un sommet de G pas encore inclus dans la transformation. Si ce n'est pas un isomorphisme de sous-graphe, on retire la dernière association faite, et on repart en incluant un autre sommet. Et ainsi de suite. Lorsqu'on a une transformation utilisant tous les sommets de H et définissant un isomorphisme avec un sous-graphe de G, on la met de coté (c'est une solution), et on revient en arrière pour trouver les autres solutions.

La manière de tester si on a bien obtenu un isomorphisme est celle décrite précédemment à l'aide de la représentation matricielle. On définit la matrice de permutation P de la transformation, puis la matrice $I_{n,m}$ permettant le redimensionnement d'une matrice, et on regarde si la relation $M_H = I^t_{n,m}.P.M_G.P^t.I_{n,m}$ est vérifiée.

La première remarque à faire sur le principe donne la validité de cet algorithme. Si deux graphes sont isomorphes et ont exactement ou plus de 2 sommets, alors il existe deux sous-graphes respectifs de chacun des graphes, avec un sommet de moins, tels que les deux sous-graphes sont isomorphes. De ce fait, en augmentant progressivement la taille de chaque sous-graphe de H donnant un isomorphisme, on atteint tous les sous-graphes isomorphes à G.

Une deuxième observation a été faite. On peut dès le départ savoir que certaines associations ne donneront pas lieu à un sous-graphe. Avec les graphes H et G de la figure 6, il est clair que le sommet c du graphe H ne peut pas être associé au sommet c du graphe G. En effet c_H a stricement plus d'arcs entrant que c_G, la structure autour de c_H ne pourrait donc pas être préservée par un isomorphisme. On construit une matrice $M^0 \in \mathbb{M}_{m,n}$ telle que, pour $i \leq n, j \leq n$, on n'essaie d'associer les sommets i et j que si $M^0_{i,j} = 1$. En prenant en compte la remarque précédente, on définit alors

$$M^0_{i,j} = \begin{cases} 1 & \text{si le nombre d'arcs entrant est plus grand pour i dans G que pour j dans H,} \\ & \text{et si le nombre d'arcs sortant est plus grand pour i dans G que pour j dans H} \\ 0 & \text{sinon.} \end{cases}$$

Les conditions sur les inégalités sont prises au sens large.

Le calcul effectif du nombre d'arcs entrant (respectivement sortant) d'un sommet d'un graphe se fait simplement à partir de sa matrice d'adjacence en additionnant, à colonne (respectivement ligne) constante correspondant au sommet, les valeurs de la matrice sur toutes les lignes (respectivement colonnes).

L'algorithme est alors détaillé dans la figure 9. On note i un entier représentant le nombre de sommets dans le sous-graphe considéré plus 1. À l'origine on a donc i=1. On conserve la permutation σ sous la forme d'un ensemble F d'éléments de couples de la forme (i, j) avec $i \in G$, et $j \in H$. La première étape consiste donc à initialiser les matrices que l'on va utiliser (M_G, M_H, $I_{n,m}$, *et* M^0). Une fois l'initialisation faite, pour parcourir toutes les possibilités, on utilise la procédure *Backtrack* en partant d'une matrice de possibilités M^0, de i=1, et d'un ensemble F vide. Cette procédure commence par vérifier si la situation courante est une solution d'isomorphisme de sous-graphe. Pour cela il faut d'une part que le sous-graphe soit de la taille de H, et d'autre part que ce soit bien un isomorphisme avec H, c'est à dire que $M_H = I^t_{n,m}.P.M_G.P^t.I_{n,m}$ soit vérifiée.

FIGURE 9 – Algorithme de backtracking simple

1. Initialiser $M_G,\ M_H,\ I_{n,m},\ et\ M^0$
2. $Backtrack(M^0, 1, \emptyset)$

3. procedure Backtrack(M, i, F) :
   - *a)* Si i > n et la condition $M_H = I^t_{n,m}.P.M_G.P^t.I_{n,m}$ est vérifiée, alors afficher(F) et *return*.
   - *b)* $\forall j \in H$ tel que $M_{i,j} = 1$,
     - *i.* $F \cup \{(i,j)\}$
     - *ii.* $M' = M\ et\ \forall k > i,\ M'_{k,j} = 0$
     - *iii.* $Si\ S_H = P.S_G.P^t,\ Backtrack(M',\ i+1,\ F)$
     - *iv.* $F = F - \{(i,j)\}$

Si la situation courante est une solution, on arrête l'algorithme et on renvoie la solution. Si ce n'était pas une solution, pour chaque association possible de i dans G avec un élément j de H (possibilité inscrite dans M^0), on ajoute (i, j) à F, on met à jour M (les associations choisies ne pourront plus l'être ensuite), et on appelle *backtrack* avec la matrice M, l'entier i+1, et F mis à jour dans le cas où l'élément ajouté donne toujours un isomorphisme entre les sous-graphes S_G et S_H considérés par ajouts successifs. Ensuite il suffit de retirer (i, j) et de tester l'association suivante. Il est à noter qu'on peut déterminer la matrice de permutation P à partir de l'ensemble F en fixant les éléments de F dans la permutation : $\forall (i,\ j) \in F,\ \sigma(i) = j$, et les autres éléments n'ont pas d'importance. Il suffit que la fonction ainsi définie soit bien une permutation, on peut trouver des valeurs de $\sigma(k)$ en prenant à chaque fois le plus petit entier non encore attribué. Les sous-graphes S_G et S_H de G et H considérés sont respectivement les sous-graphes définis à partir des sommets i tels que $\exists j,\ (i,\ j) \in F$ et les sommets j tels que $\exists i,\ (i,\ j) \in F$.

5.4.2 Raffinement et vérification à priori

L'algorithme détaillé précédemment est une application directe du concept de backtracking au problème d'isomorphisme de sous-graphes, et Ullman propose un deuxième algorithme basé sur une vérification à priori, ou *forward checking*. L'idée est, à chaque ajout d'une correspondance dans la transformation, de vérifier qu'il existe une correspondance supplémentaire possible. L'intérêt est d'éliminer des branches impossibles en les vérifiant à l'avance. La vérification de l'existence d'une permutation est donc faite avant l'appel récursif à la procédure *Backtrack* grâce à la fonction *Forward_ Checking*. Cette vérification tient de la définition d'un isomorphisme en tant que graphe plutôt qu'en termes matriciels. Pour que le sous-graphe S_G de G choisi soit isomorphe au sous-graphe S_H de H auquel il est associé, il est nécessaire et suffisant qu'ils aient la même structure d'arcs. Ainsi si deux sommets sont reliés par un arc dans S_G alors leurs correspondants dans S_H doivent aussi être reliés par un arc orienté de la même manière, et inversement. Si cette condition est vérifiée alors la matrice M conservant les possibilités d'association gardera possible cette association, sinon elle l'interdira. Enfin si plus aucune association n'est possible pour le sommet à

FIGURE 10 – Algorithme de backtracking avec forward checking

1. Initialiser $M_G,\ M_H,\ et\ M^0$
2. $Backtrack(M^0, 1, \emptyset)$

3. procedure Backtrack(M, i, F) :
   - *a)* Si i > n alors F représente un isomorphisme de sous-graphe : afficher(F) et *return*.
   - *b)* $\forall j \in H$ tel que $M_{i,j} = 1$,
     - *i.* $F \cup \{(i, j)\}$
     - *ii.* $M' = M\ et\ \forall k > i,\ M'_{k,j} = 0$
     - *iii.* Si Forward_Checking(M', i, F)=true, alors Backtrack(M', i+1, F)
     - *iv.* $F = F - \{(i, j)\}$

tester au vue des modifications faites à M, la fonction *Forward_ Checking* renvoie *false*. Elle renvoie *true* dans le cas contraire. Le nouvel algorithme est donné dans la figure 10, et la fonction *Forward_ Checking* dans la figure 11.

La complexité dans le pire des cas [17] arrive quand toutes les associations sont possibles (M^0 n'est composée que de 1). Dans ce cas, à chaque niveau de la procédure *Backtrack*, il y aura $O(n)$ appels récursif à *Backtrack*. Puisqu'il y a m niveaux d'appels de *Backtrack* (m sommets dans G), le nombre d'appels récursifs est en $O(n^m)$. Pour chaque appel de la procédure *Forward_ Checking*, $O(m^2)$ opérations sont effectuées. Puisque nous utilisons cet algorithme pour comparer un graphe à une base de L graphes, la complexité maximale dépend linérairement de L, et est donc $O(L.n^m.n^2)$ où n est le nombre de sommets maximal des graphes de la base et m le nombre de sommets du graphe à tester.

Il est à noter que ce pire cas n'est pas celui que nous rencontrons, et donc que la complexité en réalité est bien moindre. Dans la suite du stage, il sera donc important de l'évaluer pour la classe particulière de graphes dont nous nous occupons.

Cet algorithme reste une référence dans la recherche d'isomorphismes de sous-graphe. Je vais maintenant présenter une variante de l'algorithme d'Ullman allant plus loin dans le processus de vérification à priori, puis un algorithme de recherche d'isomorphismes adapté à une utilisation sur de grandes bases de graphes déjà connus.

5.5 Algorithme VF

L'algorithme dit VF [18] reprend les grandes lignes de celui d'Ullman. Ils ont en commun l'utilisation du *backtracking* en combinaison de mécanismes de *forward checking*. L'algorithme VF propose d'appliquer des conditions supplémentaires pour éliminer des candidats. Ces nouvelles conditions s'ajoutent à celle de l'algorithme précédent (qui étaient nécessaires et suffisantes) et servent à accélérer le processus.

Les auteurs définissent deux ensembles T^{in} et T^{out} relatifs à un sous-graphe S_G d'un graphe G. T^{in} contient les sommets de $G - S_G$ dont un arc orienté vers S_G les relie à un sommet de S_G, tant dis que T^{out} contient ceux dont un

FIGURE 11 – Fonction Forward_Checking(M, i, F)

1. $\forall k = (i+1)..m$ *et* $l = 1..n$
   Si $M_{k,l} = 1$ *alors* :
   Vérifier qu'il y a isomorphisme, c'est à dire : $\forall (v, w) \in F$,
   a) S'il y a un arc $e_G = (k, v)$ (*respectivement* $(v,\ k)$) $\in E_G$,
   alors il y a un arc $e_H = (l, w)$ (*respectivement* $(w,\ l)$) $\in E_H$
   b) S'il y a un arc $e_H = (l, w)$ (*respectivement* $(w,\ l)$) $\in E_H$,
   alors il y a un arc $e_G = (k, v)$ (*respectivement* $(v,\ k)$) $\in E_G$

   Si une des conditions n'est pas vérifiée, alors $M_{k,l} = 0$.

2. S'il existe k $\leq$ m tel que $\forall l = 1..n, M_{k,l} = 0$ alors *return false* sinon *return true.*

arc orienté depuis S_G les relie à un sommet de S_G. Par exemple les ensembles T^{in} et T^{out} du sous-graphe de G constitué des sommets $\{d, e, a\}$ sont donnés figure 12. On note $T = T^{in} \cup T^{out}$. Il est à noter que T^{in} et T^{out} ne sont pas nécessairement disjoints. Les règles portant sur T^{in} et T^{out} cherchent à éliminer les candidats de l'étape suivante car T^{in} et T^{out} contiennent tous les potentiels sommets du tout suivant. Ces régles stipulent que le nombre de successeurs et de prédécesseurs dans T^{in} (respectivement T^{out}) d'un sommet i de S_G associé à un sommet j de S_H doivent être le même pour i et j. Une dernière règle s'intéresse aux sommets qui ne sont ni dans S_G, ni dans T, elle permet d'éliminer des sommets qui auraient été étudiés deux tours plus loin. Les règles sont détaillées dans la figure 13. Il est clair que les deux premières règles sont celles de la procédure *Forward_ Checking* de l'algorithme d'Ullman tandis que les autres sont spécifiques à l'algorithme VF.

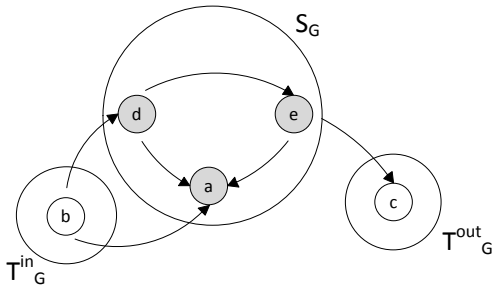


FIGURE 12 – Ensemble des sommets entrant et sortant pour S_G

La complexité dans le pire des cas de l'algorithme VF n'est pas meilleure que celle pour celui d'Ullman. Cela dit, et comme l'indiquent certains travaux [19], dans le cas plus réaliste de grands graphes peu connectés (comme ceux relatifs

à un CFG), et sur des applications pratiques, il est plus efficace que les autres algorithmes testés, dont celui d'Ullman.

Dans la suite du stage il sera nécessaire d'étudier la complexité temporelle de ces algorithmes ainsi que de réaliser des implémentations et de les tester sur les bases de CFG provenant des malware récoltés. Une fois ces tests effectués, nous serons plus à même de choisir l'algorithme le plus efficace.

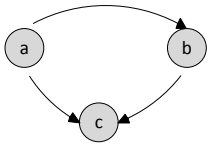
Utile à l'étape	Nom	Condition
i+0	R_pred	Si et seulement si pour chaque prédécesseur v' de v dans S_G, le sommet w' correspondant dans S_H est un prédécesseur de w, et vice versa
	R_succ	Si et seulement si pour chaque successeur v' de v dans S_G, le sommet w' correspondant dans S_H est un successeur de w, et vice versa
i+1	R_termout	Si et seulement si le nombre de prédécesseurs (resp. successeurs) de v qui sont dans T^{in}_G est égal au nombre de prédécesseurs (resp. successeurs) de w qui sont dans T^{in}_H
	R_termin	Si et seulement si le nombre de prédécesseurs (resp. successeurs) de v qui sont dans T^{in}_G est égal au nombre de prédécesseurs (resp. successeurs) de w qui sont dans T^{in}_H
i+2	R_new	Si et seulement si le nombre de prédécesseurs (resp. successeurs) de v qui ne sont ni dans S_G ni dans T_G est égal au nombre de prédécesseurs (resp. successeurs) de w qui ne sont ni dans M_H ni dans T_H.

FIGURE 13 – Règles de forward checking pour l'algorithme VF

5.6 Construction d'un arbre de décision

Une autre approche consiste à calculer au préalable tous les isomorphismes possibles pour chaque graphe de la base et à les ranger sous la forme d'un arbre de décision [20]. Chaque graphe est représenté sous la forme de sa matrice d'adjacence et décomposé en représentation ligne-colonne. On peut noter que, quel que soit le graphe, les termes diagonaux de sa matrice sont tous nuls (il n'y a pas d'arc d'un sommet vers lui-même d'après la définition 2). Pour plus de lisibilité, on note ces termes tous nuls -. La définition de la représentation ligne-colonne est donnée à la définition 6. Par exemple, le graphe H, sa matrice d'adjacence et sa représentation ligne-colonne est à la figure 14.

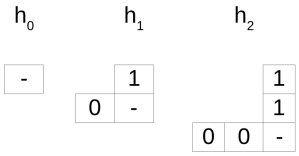
Définition 6. *Soit H un graphe orienté à n sommets et M sa matrice d'adjacence. Le i^e vecteur de la représentation ligne-colonne de H est, avec* $0 \leq i \leq n-1$, $h_i = (M_{i+1,1}, M_{i+1,2}, ..., M_{i+1,i}, -, M_{i,i+1}, M_{i-1,i+1}, ..., M_{1,i+1})$.



(a) Graphe H

$$\begin{pmatrix} - & 1 & 1 \\ 0 & - & 1 \\ 0 & 0 & - \end{pmatrix}$$

(b) Matrice d'adjacence de H



(c) Représentation ligne-colonne

FIGURE 14 – Graphe H et représentation ligne-colonne

Pour chaque graphe de la base, on trouve tous ses isomorphismes en déterminant d'une part toutes les matrices de permutation P de taille n * n puis en effectuant la transformation $P^t.M.P$. On place ensuite tous les isomorphismes dans un arbre de décision selon leur représentation ligne-colonne. En partant de la racine, le choix du premier fils est fait suivant h_1, puis h_2 et ainsi de suite. Une fois qu'on a fait le choix h_n donc que le graphe est correctement placé, on ajoute un nom à la feuille atteinte selon le graphe atteint (ici H). Il est de cette manière à la fois aisé de rajouter des graphes à la base, et de placer un graphe à analyser dans l'arbre. Selon la branche atteinte on pourra dire s'il s'agit d'un isomorphisme avec un des graphes de la base ou non, et on sera capable grâce au label placé sur la feuille de savoir avec quel(s) graphe(s) il y a correspondance. Pour le graphe H à 3 sommets, il y a 3 !=6 permutation, données par les matrices P_i de la figure 15. Les matrices $M_{\sigma(H)}$ correspondants, notées H_i sont données figure 16.

L'arbre de décision engendré par H (et ses isomorphismes) est donné figure 17. Ici les seuls graphes reconnus sont les isomorphismes de H. Si on cherche une correspondance entre H_3 et H, en parcourant l'arbre de décision à l'aide de la représentation ligne-colonne de H_3, on arrive aisément à voir que H_3 permet de terminer en une feuille dont le label est H_3 et qu'il est donc isomorphe à H. La détermination du problème d'isomorphisme de graphe se fait donc ici en un temps linéaire à partir du moment où l'arbre de décision a été déterminé.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$ (a) P_1 $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$ (b) P_2 $$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$ (c) P_3 $$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$ (d) P_4

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$ (e) P_5 $$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$ (f) P_6

FIGURE 15 – Matrices de permutation 3*3

$$\begin{pmatrix} - & 1 & 1 \\ 0 & - & 1 \\ 0 & 0 & - \end{pmatrix}$$ (a) H_1 $$\begin{pmatrix} - & 1 & 1 \\ 0 & - & 0 \\ 0 & 1 & - \end{pmatrix}$$ (b) H_2 $$\begin{pmatrix} - & 0 & 1 \\ 1 & - & 1 \\ 0 & 0 & - \end{pmatrix}$$ (c) H_3 $$\begin{pmatrix} - & 1 & 0 \\ 0 & - & 0 \\ 1 & 1 & - \end{pmatrix}$$ (d) H_4

$$\begin{pmatrix} - & 0 & 0 \\ 1 & - & 1 \\ 1 & 0 & - \end{pmatrix}$$ (e) H_5 $$\begin{pmatrix} - & 0 & 0 \\ 1 & - & 0 \\ 1 & 1 & - \end{pmatrix}$$ (f) H_6

FIGURE 16 – Matrices $M_{\sigma(H)}$

De plus la complexité de construction de cet arbre est linéaire par rapport au nombre d'arbres dans la base et la complexité du problème d'isomorphisme dépend uniquement de la taille du graphe à tester et pas de la taille de la base.

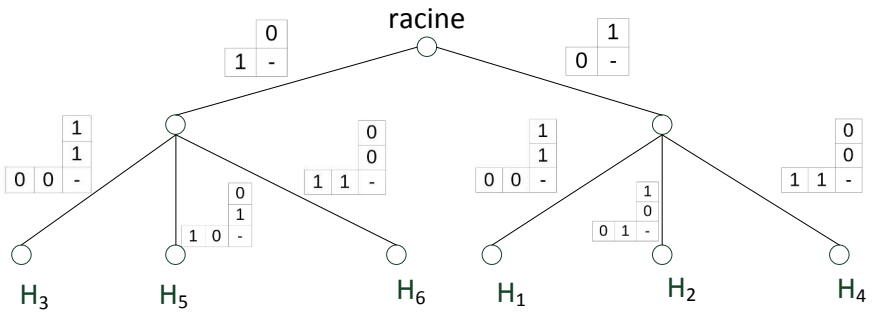


FIGURE 17 – Arbre de décision pour la base $\{H\}$

La version présentée précédemment de l'algorithme ne fonctionne que pour détecter des isomorphismes de graphe. On peut la modifier pour y ajouter la détection des isomorphismes de sous-graphes en augmentant artificiellement la taille des graphes de la base jusqu'à la taille maximale des graphes que nous aurons à traiter (environ 30 sommets). On est alors capable de détecter des isomorphismes de sous-graphes à partir du moment où le parcours de l'arbre de décision s'arrête sur une feuille dont le label correspond à un graphe de la base.

La grande faiblesse de cette approche est qu'elle demande de calculer au préalable toutes les permutations d'un graphe, et cette opération ne peut se faire qu'en un temps en $O(n!)$. Dans le cas de la recherche d'isomorphismes de sous-graphes, la taille des graphes étudiés est de 30 sommets avec $30! = 2.65 * 10^{32}$. Or ni le calcul de toutes les matrices de permutation ni la sauvegarde d'autant de matrices de taille 30*30 ne nous semble raisonnablement accessible. Pour cette raison cet algorithme serait plus adapté dans le cas de nombreux graphes plus petits (jusqu'à une quinzaine de sommets), et ne sera pas étudié dans la suite du stage. En effet en dessous de 15 sommets, les signatures ne sont pas assez discriminantes pour la détection de malware et le risque est de produire trop de faux positifs.

5.7 Parcours en profondeur

En gardant à l'esprit qu'un CFG représente un programme, on peut émettre deux hypothèses sur les graphes que nous manipulons. La première est de dire que de tels graphes ont un point d'entrée, c'est à dire qu'il existe un sommet du graphe à partir duquel on peut atteindre tous les autres. Par exemple, le sommet b du graphe G (figure 6(a)) est un point d'entrée puisqu'on peut atteindre tous les autres sommets en suivant les arcs orientés. De plus chaque sommet a au plus deux arcs sortant en raison des sommets possibles dans un CFG donnés figure 3(a). On appellera CFG simple un tel graphe, dont la définition formelle est donnée à la définition 7. On note, pour un sommet v d'un graphe G, l'ensemble des sommets adjacents à v, $A_v = \{w \in V,\ (v,\ w) \in E\}$.

Définition 7. *Un CFG simple est un graphe $G = (V, E)$ orienté vérifiant :*
- *Il a (au moins) un point d'entrée :*
  $\exists p \in V, \forall v \neq p \in V, \exists e_1, e_2, ..., e_k \in E$ *avec* $\forall i \leq k, e_i = (u_i, v_i)$ *vérifiant* $u_1 = p,\ v_k = v,$ *et* $\forall\ 1 \leq i \leq k-1,\ v_i = u_{i+1}$.
- *Chaque sommet a au plus deux arcs sortant :*
  $\forall u \in V, Card\{(u,\ v) \in E, v \in V\} \leq 2$.

À partir de cette configuration de graphe, on peut effectuer un parcours en profondeur dans le but de construire un arbre recouvrant (*spanning tree*) du CFG simple selon une structure utile à la découverte d'isomorphismes. J'ai repris l'algorithme de Tarjan [21] et l'ai modifié afin que la transformation induite par celui-ci donne à la fois un arbre recouvrant mais que la structure construite soit isomorphe au graphe initial. L'algorithme donne en sortie un graphe constitué de deux types d'arcs, donc deux ensembles E_1 et E_2 distincts, dont le second type est représenté par un arc en pointillés. Cet algorithme, partant d'un sommet donné en entrée, permet de numéroter les sommets du graphe de manière unique comme suit. On numérote l'argument par l'entier 1. Puis on regarde tous ses fils qui ne sont pas déjà numérotés, on associe les deux par une flèche normale (E_1) et on appelle l'algorithme pour ce fils. Pour les fils déjà numérotés, on associe les deux par une flèche en pointillés (E_2) et on s'arrête. L'algorithme est détaillé figure 18. On notera $T_s(G)$ le graphe à deux types d'arcs résultant de la transformation précédent à partir du sommet s du graphe G. Par exemple, pour le graphe G donné à la figure 19(a) en partant du sommet b, le graphe $T_b(G)$ est donné figure 19(b).

La proposition 1 indique que le graphe $T_i(G)$ est isomorphe au graphe G si l'on confond ses deux types d'arcs dans le cas où i est un point d'entrée de G.

Soit i=0
Soit T le graphe tel que $V_T = V_G$ et $E_T = \emptyset$.
Soit N la fonction nulle sur V_T.
DFS(s).
return T, N.

Fonction DFS(v) :
    i++ ;
    N(v)=i ;
    $\forall w \in A_G(v)$,
        Si w n'est pas encore numéroté ($N(w) = 0$)
            Associer v et w normalement : $E_{T,1} = E_{T,1} \cup (v,\ w)$;
            DFS(w) ;
        Si w est déjà numéroté ($N(w) \neq 0$)
            Associer v et w en pointillés : $E_{T,2} = E_{T,2} \cup (v,\ w)$;

FIGURE 18 – Algorithme de numérotation et construction du graphe à partir du sommet s d'un graphe G

Cette proposition s'explique à la lecture de l'algorithme. Soit $H = T_i(G)$. Les deux graphes ont les mêmes sommets. Le sommet i est un point d'entrée donc en appliquant l'algorithme, tous les sommets de G seront atteints. Quel que soit l'arc $e = (v_1, v_2) \in E_G$, v_1 est atteint dans l'algorithme et v_1 est adjacent à v_2 donc soit $e \in E_{H,1}$ soit $e \in E_{H,2}$. Donc $e \in E_{H,1} \cup E_{H,2}$. Tous les arcs du graphe G sont dans sa transformation. Inversement si un arc $e = (w_1, w_2)$ est dans $E_{H,1}$ ou $E_{H,2}$, c'est que w_1 et w_2 sont adjacents dans G, donc $e \in E_G$. On a bien montré que les arcs des graphes G et $(V_H, E_{H,1} \cup E_{H,2})$ ont les mêmes sommets et les mêmes arcs. Ils sont identiques donc isomorphes.

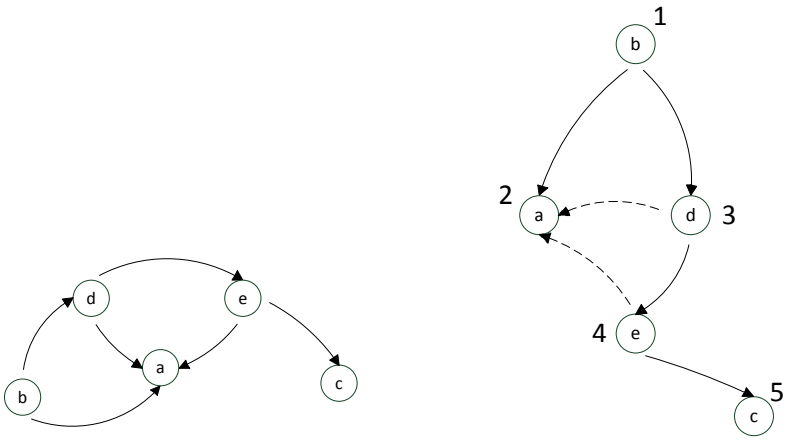
Proposition 1. *Soit G un graphe et i un point d'entrée de G. Notons* $H = T_i(G)$. *Les deux graphes G et* $(V_H, E_{H,1} \cup E_{H,2})$ *sont isomorphes.*

La proposition 2 indique que le graphe $T_i(G)$ muni de la première catégorie d'arcs uniquement avec i point d'entrée de G est un arbre recouvrant de G si l'on considère que le fait que l'arc $(i,\ j) \in E$ soit dans le graphe signifie que le sommet j est un fils du sommet i dans l'arbre. La démonstration de cette proposition est faite dans l'article de Tarjan [21].

Proposition 2. *Soit G un graphe et i un point d'entrée de G. Notons* $H = T_i(G)$. *Le graphe* $(V_H, E_{H,1})$ *est un arbre recouvrant de G.*

Le résultat intéressant de cette approche est le théorème 3 qui donne une condition nécessaire et suffisante pour avoir un isomorphisme entre deux graphes ayant un point d'entrée. On considère ici que les sommets de $T_i(G_1)$ et $T_j(G_2)$ sont les numéros associés à chacun des sommets par la fonction N renvoyée par la transformation.

Si G_1 et G_2 sont isomorphes, il existe une bijection entre les sommets de G_1 et ceux de G_2. Si on prend i point d'entrée de G_1 et $j = \sigma(i)$, on aura $T_i(G_1) = T_j(G_2)$. Inversement, s'il existe i et j tels que $T_i(G_1) = T_j(G_2)$ avec i et j points



(a) Graphe G (b) $T_b(G)$

FIGURE 19 – Graphe G et sa transformation $T_b(G)$ avec numérotation des sommets

d'entrée de G_1 et G_2. On a, en vertu de la proposition 1, $(V_H, E_{H,1} \cup E_{H,2})$ isomorphe à G_1 avec $H = T_i(G_1)$. Or $T_i(G_1) = T_j(G_2)$ donc $(V_H, E_{H,1} \cup E_{H,2})$ est isormophe à G_1 avec $H = T_j(G_2)$. Identiquement $(V_H, E_{H,1} \cup E_{H,2})$ est isomorphe à G_2 avec $H = T_j(G_2)$. On en déduit, car l'isomorphisme de graphes est une relation d'équivalence, que G_1 et G_2 sont isomorphes.

Théorème 3. *Soit G_1 et G_2 deux graphes ayant chacun un point d'entrée. Les deux propositions suivantes sont équivalentes :*

- *$\exists\ i,\ j$ points d'entrée de G_1 et G_2 respectivement / $T_i(G_1) = T_j(G_2)$*
- *G_1 et G_2 sont isomorphes*

On peut alors formuler un algorithme simple permettant la détection d'isomorphisme de graphes entre deux graphes ayant au moins chacun un point d'entrée. Un critère simple pour savoir si un sommet est point d'entrée d'un graphe est d'en déterminer sa transformation par l'algorithme 18 et de vérifier que le nombre de sommets numérotés est égal au nombre de sommets du graphe. Pour savoir si G_1 et G_2 sont isomorphes il suffit alors de déterminer les transformations de chaque graphe à partir de chacun de ses sommets et de rechercher $i \in V_{G_1}$ et $j \in V_{G_2}$ avec i et j points d'entrée tels que $T_i(G_1) = T_j(G_2)$. Si un tel couple existe alors G_1 et G_2 sont isomorphes. Dans le cas contraire ils ne sont pas isomorphes. Cet algorithme est détaillé figure 20.

La complexité temporelle dans le pire des cas de cet algorithme peut être évaluée. Elle arrive dans le cas où tous les sommets de G_1 et G_2 sont des points d'entrée de ces graphes. Notons n le nombre de leurs sommets. Ils ont le même nombre de sommets car c'est une condition nécessaire à l'isomorphisme. On considère que le calcul des transformations $T_i(G_1)$ et $T_j(G_2)$ a été fait au préalable

Fonction Iso(G_1, G_2) :
    $\forall i \in V_{G_1}$,
        Déterminer $T_i(G_1)$
        Si i est un point d'entrée de G_1 :
            $\forall j \in V_{G_2}$,
                Déterminer $T_j(G_2)$
                Si j est un point d'entrée de G_2 :
                    Si $T_i(G_1) = T_j(G_2)$, *return true*

    return false

FIGURE 20 – Algorithme de détection d'isomorphisme de graphes à l'aide du parcours en profondeur

à l'aide de l'algorithme 18. Il y a donc 2*n transformations à faire. L'algorithme utilisé pour la transformation ne nécessite qu'un parcours du graphe à chaque fois. À chaque sommet atteint, il y a au plus 2 arcs à ajouter au graphe en sortie en vertu de la définition 7 d'un CFG simple, c'est à dire 2.n opérations. Le calcul préalable des transformations se fait donc en $O(n^2)$. Pour chaque sommet i de G_1, il a n sommets j de G_2 pour lesquels il faut tester l'égalité. On en conclut qu'il y a donc n^2 comparaisons matricielles à faire. Dans le pire des cas ces comparaisons se font en $O(n^2)$ opérations. La complexité globale de l'algorithme de test de l'isomorphisme est donc en $O(n^2) + O(n^4)$, c'est à dire en $O(n^4)$.

On a donc démontré qu'avec les hypothèses prises, c'est à dire qu'on est en présence de CFG simples (graphes ayant un point d'entrée, dont chaque sommet est au plus associé à deux autres sommets), le problème de l'isomorphisme de graphe est polynomial. Bien que cet algorithme ne soit pas généralisable tel quel aux isomorphismes de sous-graphes, il montre à quel point il est important de prendre en compte la spécificité des graphes que l'on manipule afin d'obtenir une solution optimale.

5.8 À l'aide d'automates d'arbre

Afin de construire une base de signatures par graphe de flot de contrôle et de pouvoir y reconnaître un malware analysé, les auteurs de [11] proposent d'utiliser des automates d'arbres. Les CFG seront donc au préalable tranformés en arbres, et un automate d'arbres sera utilisé pour l'ensemble de la base de signatures. Cet automate d'arbres permettra la reconnaissance des malware. On considère les graphes étudiés comme étant des CFG simples (définition 7).

5.8.1 Des CFG aux arbres

Un chemin est un mot sur $\{1,2\}^*$, ε est le chemin vide. Soit l'ordre $<$ défini sur $\{1,2\}^*$ comme suit pour $\rho, \tau \in \{1,2\}^*$ et $i \in \{1,2\}$:

- $\rho 1 < \rho 2$
- $\rho < \rho i$
- $\rho < \tau \Rightarrow \rho\rho' < \tau\tau'$.

Un domaine d'arbre est un ensemble $d \subset \{1,2\}^*$ tel que, pour tout $\rho \in \{1,2\}^*$ et $i \in \{1,2\}$, on ait $\rho i \in d \Rightarrow \rho \in d$.

Soit l'ensemble de symboles $\mathbb{F} = \{inst, jmp, call, jcc, ret\} \cup \{1, 2\}^*$. On considère un arbre sur cet ensemble de symboles comme étant une paire $t = (d(t), \hat{t})$ avec d(t) un domaine d'arbre et $\hat{t}$ une fonction de $d(t)$ dans $\mathbb{F}$. Il est à noter qu'avec cette définition, un arbre a des nœuds de deux types. Un nœud peut être une instruction prise en compte par le CFG ou un chemin, ou *pointeur*, vers un autre nœud de l'arbre.

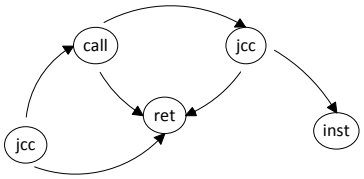
Soit $\dot{d}(t)$ l'ensemble des nœuds internes de l'arbre t. $\dot{d}(t) = \{\rho \in d(t), \hat{t}(\rho) \in \{inst, jmp, call, jcc, ret\}\}$. Il s'agit des nœuds dont le label est une instruction.

Définition 8. *Un arbre t est* bien formé *si quels que soient les chemins* $\rho, \tau \in d(t)$, $(\hat{t}(\tau) = \rho) \Rightarrow (\rho \in \dot{d}(t)$ *et* $\rho \leq \tau)$.

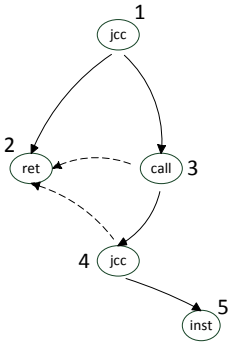
La définition 8 impose, pour un arbre bien formé, que tout chemin pointe vers un nœud interne, et interdit que le nœud vers lequel il pointe soit un précédesseur, assurant que l'arbre ainsi défini représente bien un CFG.

L'algorithme utilisé pour la transformation d'un CFG simple G en arbre bien formé est le suivant. On construit la tranformation $T_i(G)$ définie à l'algorithme 18 à partir d'un point d'entrée i du graphe G. La structure que l'on va conserver est celle d'arbre recouvrant du premier type d'arcs. On considère, conformément à la définition de l'ordre sur les sommets de l'arbre, que la racine de l'arbre a pour chemin ε, que le fils gauche d'un sommet d'arité 2 dont le chemin est ρ a pour chemin $\rho 1$ tandis que le fils droit du sommet ρ a pour chemin $\rho 2$. Si un sommet de chemin ρ n'a qu'un seul fils, celui-ci a pour chemin $\rho 1$. Le second type d'arcs sera transformé en feuilles de type chemin selon la règle suivante. Si $(u, v) \in E_{G,\ 2}$, alors on construit le sommet w de type chemin et dont la valeur est le chemin de v dans G. w est placé dans l'arbre G comme une feuille, fils du sommet u. Par exemple, figure 21, un graphe G est donné avec sa transformation par l'algorithme de parcours en profondeur à partir du sommet jcc de gauche, ainsi que sa représentation sous forme d'un arbre bien formé.

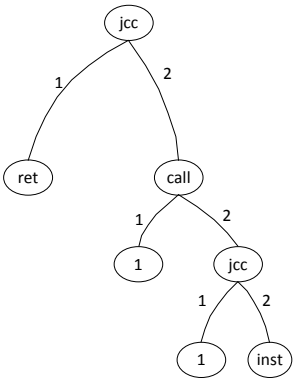
Une fois cette transformation d'un CFG à un arbre effectué, il va être possible de définir un automate d'arbres reconnaissant le langage constitué des malware connus, et donc de mener à leur détection.



(a) CFG G



(b) $T_i(G)$ où i est une instruction jcc



(c) Arbre bien formé

FIGURE 21 – CFG G, sa transformation par parcours en profondeur, et sa représentation sous forme d'arbre bien formé

5.8.2 Automates d'arbres

La théorie des automates d'arbres est largement abordée dans [22] et les résultats suivants en sont extraits.

Définition 9. *Un automate d'arbres fini est un tuple* $\mathcal{A} = (\mathbb{Q}, \mathbb{F}, \mathbb{Q}_f, \Delta)$*, où* $\mathbb{Q}$ *est un ensemble fini d'états,* $\mathbb{F}$ *un ensemble de symboles,* $\mathbb{Q}_f \subset \mathbb{Q}$ *un ensemble d'états finaux et* Δ *un ensemble fini de règles de transition de la forme* $a(q_1...q_i) \to q$ *avec* $a \in \mathbb{F}$ *ayant i successeurs, et* $q, q_1, ..., q_i \in \mathbb{Q}$*.*

Une transformation de l'arbre est faite selon l'automate. Dans un premier temps chaque feuille est transformée en un état, selon les transitions définies, puis ces transitions sont utilisées pour remonter dans l'arbre au fur et à mesure des transformations effectuées. Si à l'issue de ces transformations l'arbre est dans un état final de l'automate, on dit que l'automate reconnaît l'arbre. Pour un arbre t ayant $|t|$ nœuds, la transformation peut être effectuée avec une complexité temporelle linéaire, soit $O(|t|)$. La figure 22 présente un exemple d'automate d'arbres en définissant le tuple $(\mathbb{Q}, \mathbb{F}, \mathbb{Q}_f, \Delta)$. Cet automate ne reconnaît qu'un arbre bien fondé particulier. Celui-ci est présenté dans la figure 23 où il est dans son état initial puis subit les transformations définies par les transitions de l'automate afin de le réduire en un seul état, état final de l'automate.

Les automates d'abres ont des propriétés utiles pour notre analyse. Il est possible de construire un automate qui reconnaît exactement tous les arbres d'un ensemble fini d'arbres. Cette construction ayant une complexité temporelle linéaire, en $O(n)$ où n est la somme du nombre de nœuds de chaque arbre de l'ensemble. Il est également possible de fusionner deux automates $\mathcal{A}$ et $\mathcal{A}'$ pour obtenir un automate reconnaissant exactement l'union des arbres reconnus par chacun des deux automates originaux. Ceci peut être effectué avec une complexité de $O(|\mathcal{A}'|)$ où $|\mathcal{A}'|$ est le nombre de règles de transition de $\mathcal{A}'$.

Le résultat le plus intéressant est le théorème 4 sur les automates d'arbres minimaux. Pour chaque langage reconnaissable d'arbres, il existe un unique automate minimal en nombre d'états reconnaissant ce langage. Cet automate est donc la meilleure représentation de la base des malware, donnés sous forme d'arbres bien formés. La possiblité de déterminer cet automate minimal avec une complexité raisonnable (temps quadratique) démontre la faisabilité théorique de cette approche.

La base de données, sous forme d'automate d'arbres, est créée à partir des transformations sous forme d'arbres bien formés des malware analysés. L'algorithme de détection est alors le suivant. Pour un CFG simple que l'on souhaite scanner, on construit toutes les transformations sous forme d'arbre bien formé possibles (il y a une par point d'entrée du CFG). On tente de reconnaître chacun de ces arbres à l'aide de l'automate d'arbres représentant la base. Si l'un de ces arbres est reconnu, le programme analysé est considéré comme un malware, et l'on peut savoir avec quel(s) malware de la base il a des correspondances (selon l'état final atteint par l'automate). Si aucun de ces arbres n'est reconnu par l'automate, le programme sera considéré comme sain.

Cette approche permet de détecter les isomorphismes de graphes, ainsi que la détection d'isomorphismes de sous-graphes dans le cas où le sous-graphe est sous un graphe, c'est à dire qu'un des points d'entrée du sous-graphe est l'unique fils du sous-graphe dont le père est strictement dans le graphe. Formellement, les sous-graphes H de G détectés sont ceux vérifiant $T^{in} = \{i\}$ avec i point

$\mathbb{Q} = \{q_1, q_2, q_3, q_4, q_5, q_6\}$
$\mathbb{Q}_f = \{q_6\}$
$\mathbb{F} = \{\text{inst, jmp, ret}\} \cup \{1, 2\}*$

(a) États et symboles de l'automate

(1) $ret \rightarrow q_1$
(2) $1 \rightarrow q_2$
(3) $inst \rightarrow q_3$
(4) $jcc(q_2, q_3) \rightarrow q_4$
(5) $call(q_2, q_4) \rightarrow q_5$
(6) $jcc(q_1, q_5) \rightarrow q_6$

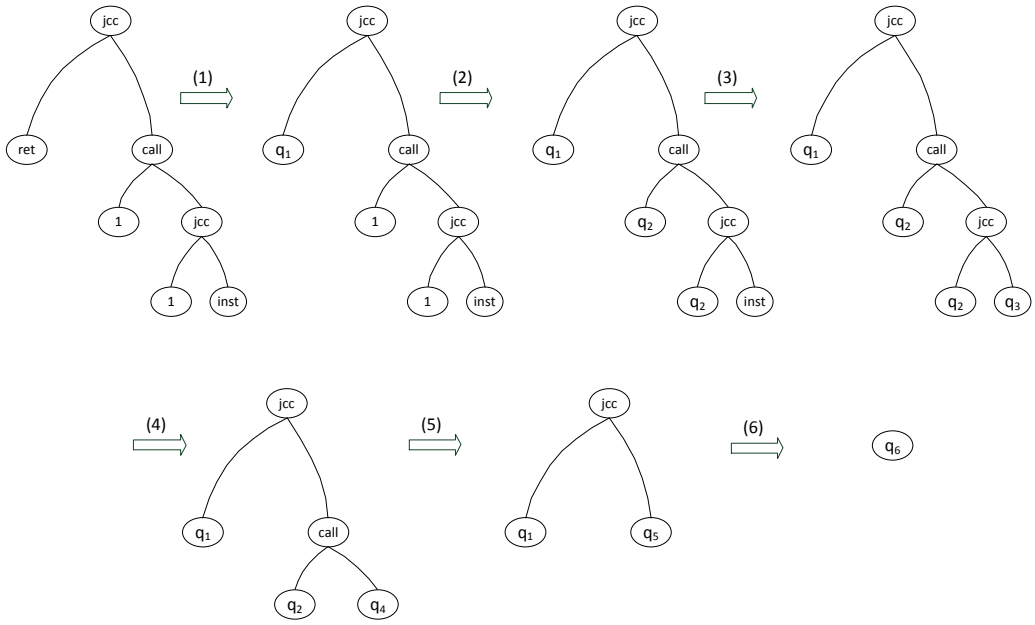
(b) Règles de transition

FIGURE 22 – Automate d'arbre reconnaissant un unique arbre

d'entrée de H et $T^{out} = \emptyset$. Il est intéressant de noter que cet ensemble de graphes détectés, bien que très inférieur à l'ensemble des sous-graphes, permet d'avoir déjà des résultats intéressant. De plus cette approche est assez efficace vis-à-vis de la complexité puisque la détermination de la reconnaissance ou non d'un arbre par un automate d'arbres se fait en $O(k * n)$ où k est le nombre de transitions de l'automate et n le nombre de sommets de l'arbre à reconnaître [22]. En effectuant cette reconnaissance à partir de tous les points d'entrée, la complexité est au pire $O(k * n^2)$. Il faudrait maintenant évaluer le nombre de transitions de l'automate en fonction du nombre de graphes qui ont servi à le construire.

Théorème 4. *Pour tout automate d'arbres $\mathcal{A}$ qui reconnaît un langage d'arbres $\mathbb{L}$, on peut déterminer en temps quadratique, soit $O(|\mathcal{A}|^2)$, un automate d'arbres $\mathcal{A}_{min}$ qui est minimal et reconnaît $\mathbb{L}$.*

FIGURE 23 – Arbre bien formé et transformations pour sa reconnaissance par l'automate



6 Conclusion

Dans ce rapport, nous avons défini ce qu'était un malware, ou infection informatique. Il a ensuite été question de la détection de ces programmes malveillants. Un état de l'art a été effectué et a exposé les faiblesses de l'approche classique par signatures simples. Entre cette approche classique et une approche purement comportementale, la méthode de détection par analyse morphologique semble prometteuse.

Afin d'une part d'avoir un socle théorique plus étayé, et d'autre part de meilleures performances, il est nécessaire de définir clairement la forme des graphes de flot de contrôle que nous souhaitons détecter. En effet, selon ce qu'on veut détecter, certains algorithmes seront adaptés tandis que d'autres ne permettront pas d'atteindre théoriquement l'objectif, ou de réaliser une implémentation s'exécutant dans un temps raisonnable. Dans un second temps, il faudra donc sélectionner plusieurs algorithmes permettant de réaliser la tâche, et les analyser sur le plan de la complexité dans le cas de la détection de nos graphes. Enfin, et pour permettre le choix de l'un ou l'autre des algorithmes, des implémentations seront réalisées afin de mesurer les perfomances (temporelles et spatiales) des algorithmes considérés, et de vérifier la pertinence des résultats obtenus. Cette étape d'implémentation pour des tests sur des malware réels est nécessaire afin de s'assurer que les algorithmes retenus sont applicables sur des cas pratiques de programmes malveillants rencontrés par les systèmes d'information.

Références

[1] F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1986.

[2] L. Adleman. An abstract theory of computer viruses. In *Advances in Cryptology*, volume 403 of *Lecture Notes in Computer Science*, pages 354–374. Springer, 1988.

[3] G. Bonfante, M. Kaczmarek, and J.Y. Marion. On abstract computer virology from a recursion theoretic perspective. *Journal in computer virology*, 1(3) :45–54, 2006.

[4] É. Filiol. *Les virus informatiques : théorie, pratique et applications*. Springer, 2004.

[5] P. Szor. *The Art of Computer Virus Defense and Research*. Symantec Press, 2005.

[6] Mihai Christodorescu and Somesh Jha. Testing malware detectors. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2004.

[7] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware : from a survey towards an established taxonomy. *Journal in Computer Virology*, 4 :251–266, 2008.

[8] P. Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007.

[9] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Usenix Security Symposium*, 2006.

[10] *www.pintool.org*.

[11] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, 5 :263–270, 2009.

[12] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 129–143, 2006.

[13] Sébastien Bardin and Philippe Herrmann and Franck Védrine. Refinement-based CFG Reconstruction from Unstructured Programs. Technical report, CEA, 2010.

[14] J.L. Gross and J. Yellen. *Graph Theory and its applications*. Chapman and Hall, CRC Press, 2006.

[15] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The graph isomorphism problem : its structural complexity*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993.

[16] J.R. Ullman. An algorithm for Subgraph Isomorphism. *Journal of the Association for Computing Machinery, Vol 23, No 1, pages 31-42*, 1976.

[17] B.T. Messmer. *Efficient Graph Matching Algorithms for Preprocessed Model Graph*. PhD thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, Switzerland, 1995.

[18] L.P. Cordella, P. Foggia, C.Sansone, and M. Vento. Performance Evaluation of the VF Graph Matching Algorithm. *http ://amalfi.dis.unina.it*, 1999.

[19] P. Foggia, C.Sansone, and M. Vento. A Performance Comparison of Five Algorithms for Graph Isomorphisms. 2002.

[20] B. T. Messmer and H. Bunke. Subgraph isomorphism in polynomial time. Technical report, 1995.

[21] Robert Tarjan. Depth-first search and linear graph algorithms. *Foundations of Computer Science, Annual IEEE Symposium on*, 0 :114–121, 1971.

[22] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. *tata.gforge.inria.fr*, 2008.