HAL
open science

Désassemblage et détection de logiciels malveillants auto-modifiants

Aurélien Thierry

To cite this version:

Aurélien Thierry. Désassemblage et détection de logiciels malveillants auto-modifiants. Informatique et langage [cs.CL]. Université de Lorraine, 2015. Français. NNT : 2015LORR0011 . tel-01751411v2

HAL Id: tel-01751411

https://theses.hal.science/tel-01751411v2

Submitted on 23 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LORRAINE

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Désassemblage et détection de logiciels malveillants auto-modifiants

THÈSE

présentée et soutenue publiquement le 11 mars 2015

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Aurélien Thierry

Composition du jury

Rapporteurs :	Frédéric Cuppens	Professeur, Télécom Bretagne
	Hervé Debar	Professeur, Télécom SudParis
Examinateurs :	Sébastien Bardin	Chargé de recherche, CEA
	Guillaume Bonfante	Maître de conférences, Université de Lorraine
	Pierrick Gaudry	Directeur de recherche, CNRS
	Valérie Viêt Triêm Tông	Maître de conférences, Supélec
Directeur de thèse :	Jean-Yves Marion	Professeur, Université de Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Remerciements

Je tiens en premier lieu à remercier les membre du jury qui ont accepté d'évaluer mon travail de thèse et en particulier Frédéric Cuppens et Hervé Debar d'avoir pris le temps de relire ce document en me faisant de précieux retours.

Ma reconnaissance va principalement à mon directeur de thèse, Jean-Yves Marion. Il a su me guider scientifiquement pendant ces années et a pu accorder beaucoup de temps à la relecture du manuscrit malgré ses nombreuses contraintes. Merci d'avoir su me proposer les orientations pertinentes pour ma recherche tout en me laissant libre du choix. Je remercie également Pierrick Gaudry de m'avoir apporté un point de vue extérieur très pertinent sur mes travaux.

J'ai eu un grand plaisir à travailler au sein de l'équipe CARTE et en particulier avec mes collègues du LHS. Merci à Fabrice Sabatier pour son expertise, sa bonne humeur et sa disponibilité, même lorsque le temps nous manquait, à Guillaume Bonfante d'avoir su proposer et porter des sujets originaux, à Thanh Dinh et Joan de m'avoir fait découvrir des sujets connexes au mien dans le domaine de l'analyse de code. Je suis plus généralement reconnaissant envers tous les membres de CARTE pour leur accueil et leur énergie alimentant des discussions passionnées. Merci aux membres du bureau B235, particulièrement à Hugo et Hubert pour nos discussions quotidiennes, toujours intéressantes, souvent scientifiques et parfois productives, ainsi que pour la bonne humeur qui régnait dans notre bureau.

Je tiens également à remercier Guillaume de m'avoir donné l'opportunité d'enseigner et pour ses conseils dans ce domaine, ainsi que Karen pour ses conseils pédagogiques avisés. Je remercie mes étudiants de m'avoir permis, à leurs dépends, de diversifier mon emploi du temps et d'avoir (parfois) été réceptifs à mes enseignements.

Je n'oublierai pas non plus les doctorants du Loria qui m'ont fait découvrir le laboratoire et qui ont, entre autres, animé le picnic hebdomadaire. Merci aux anciens, à Nicolas, Jean-Christophe, Henri, à Frédéric, Hamza, aux plus jeunes, à Laurent, Ludovic, Éric et tous les autres. Merci aux amis avec qui j'ai découvert et pratiqué l'escalade, à Aurélia, Răzvan, Svyatoslav et Jordi. Merci à Will, Bianca, Christophe, Dounia et Aurélie avec qui j'ai découvert Nancy et qui ont égayé mon séjour en Lorraine.

Je suis également reconnaissant à Ludovic Mé ainsi qu'à l'ensemble des enseignants de l'équipe CIDRE à Rennes de m'avoir fait découvrir la recherche, ainsi qu'à Mounir et Thomas grâce à qui j'ai décidé de continuer dans cette voie.

Merci à mes parents et à mon frère pour leur soutien sans faille durant toutes mes études et leur implication dans le difficile exercice de relecture du manuscrit.

Enfin je remercie Marine qui a su m'encourager et me supporter au quotidien et que j'espère avoir malgré tout encouragée à se lancer dans l'aventure d'une thèse.

Sommaire

Introduction **1**

Partie I Désassemblage et analyse de binaires **5**

1 Assembleur **7**

1.1 Compilation et fichiers exécutables 7

1.2 Assembleur x86 et x86_64 9

1.3 Désassemblage 13

1.3.1 Reconstruction du graphe de flot de contrôle 14

1.3.2 Parcours linéaire 15

1.3.3 Parcours récursif 16

1.3.4 Désassemblage parfait et graphe de flot de contrôle parfait . . . 17

1.3.5 Difficulté théorique du désassemblage 19

1.3.6 Analyse statique et dynamique 19

1.4 Conclusion 20

2 Techniques d'obscurcissement de code **21**

2.1 Théorie de l'obscurcissement 21

2.2 Exemples d'obscurcissement 22

2.3 Chevauchement de code 27

2.4 Auto-modification 30

2.5 Logiciels malveillants et obscurcissement 32

2.6 Conclusion 32

3 Sémantique de l'assembleur et langage intermédiaire **35**

3.1 Sémantique concrète pour un langage assembleur 36

3.2 Assembleur et langage intermédiaire 39
3.3 Revue de littérature des langages intermédiaires 40
3.4 Langage intermédiaire et analyse de binaires 41
3.5 Implémentations 43
3.5.1 Preuve de concept en C 43
3.5.2 Émulation de code auto-modifiant avec BAP 45
3.6 Conclusion 46

4 Analyse dynamique de programmes auto-modifiants 47
4.1 Auto-modification et vagues 47
4.2 Trace, niveaux d'exécution et vagues 49
4.3 Revue de littérature 54
4.4 Implémentations 54
4.4.1 Émulation avec BAP 55
4.4.2 Instrumentation avec Pin 56
4.5 Conclusion 56

5 Traitement du chevauchement de code par analyse statique 57
5.1 Revue de littérature 57
5.2 Analyse statique du chevauchement de code 60
5.2.1 Définitions 60
5.2.2 Couches linéaires 61
5.2.3 Couches désassemblées par parcours récursif 63
5.2.4 Parcours récursif 64
5.3 Conclusion 65

6 Désassemblage d'un programme auto-modifiant 67
6.1 Désassemblage parfait 67
6.1.1 Graphe de flot de contrôle simple 68
6.1.2 Graphe de flot de contrôle parfait 71
6.1.3 Revue de littérature 72
6.2 Graphe de flot de contrôle paramétré par une exécution 73
6.3 Approche hybride 75
6.3.1 Architecture générale 75
6.3.2 Analyse statique de chaque instantané 76
6.4 Revue de littérature 77
6.4.1 Analyse dynamique 77

6.4.2 Interprétation abstraite et analyse hybride 78
6.5 Implémentation 78
6.6 Expérience sur `hostname.exe` empaqueté 79
6.7 Conclusion 81

Partie II Détection et analyse morphologique 83

7 Détection des logiciels malveillants 85
7.1 Contexte de détection 85
7.2 État de l'art 87
7.3 Analyse morphologique 88
7.3.1 Comparaison des graphes de flot de contrôle 88
7.3.2 Algorithme de comparaison 90
7.3.3 Revue de littérature 91
7.4 Conclusion 92

8 Algorithmes de détection de sous-graphes 93
8.1 Détection par analyse morphologique 93
8.1.1 Graphes de flot 93
8.1.2 Isomorphisme de graphes et de sous-graphes de flot 95
8.1.3 Sites 96
8.1.4 Fonctionnement d'un détecteur morphologique 98
8.1.5 Problème de la détection des sites isomorphes 100
8.2 Approches générales à l'isomorphisme de sous-graphes 101
8.2.1 Représentation sous forme matricielle et recherche exhaustive . 102
8.2.2 Algorithme d'Ullmann 104
8.2.3 Revue de littérature 113
8.3 Approches spécifiques aux graphes de flot 113
8.3.1 Algorithme par parcours de graphes de flot 113
8.3.2 SIDT : Site Isomorphism Decision Tree 131
8.4 Comparaison des performances 135
8.4.1 Complexité 135
8.4.2 Détail des implémentations 137
8.4.3 Performances des implémentations 138
8.5 Conclusion 141

9 Application à la détection de similarités logicielles 143
9.1 Contexte d'analyse de code 143
9.1.1 Waledac et OpenSSL 144
9.2 Analyse morphologique 144
9.2.1 Correspondance fine entre instructions 145
9.2.2 Implémentation et résultats 145
9.3 Limites 149
9.4 Application à Duqu et Stuxnet 149
9.5 Perspectives 150
9.6 Conclusion 150

10 Cas d'étude : détection de Duqu connaissant Stuxnet 153
10.1 Détection d'une infection par Duqu 153
10.1.1 Déroulement d'une infection 153
10.1.2 Reconstruction du code du pilote 154
10.2 Analyse fonctionnelle du pilote de Duqu à partir du code source 157
10.2.1 Initialisation du pilote lors du démarrage du système 157
10.2.2 Injection de code 160
10.3 Réalisation d'une version défensive 162
10.3.1 Phase d'initialisation 162
10.3.2 Phase de mémorisation 162
10.3.3 Phase de détection 163
10.3.4 Démonstration 163
10.4 Perspectives 165
10.5 Conclusion 165

Conclusion 167

Bibliographie 169

Annexe A Désassemblage et détection de `hostname.exe` empaqueté 177

Introduction

Les risques liés à la sécurité informatique sont multiples. L'utilisateur d'un ordinateur souhaite protéger les données relevant de sa vie privée. Les entreprises courent le risque de pertes liées à la divulgation de leur propriété intellectuelle ou de leurs secrets commerciaux.

Les logiciels sont au centre des systèmes d'information. Ils peuvent présenter des bogues ou des vulnérabilités exposant leurs utilisateurs à ces risques. La présence et l'exploitation d'une vulnérabilité dans un programme légitime ne traduit que des erreurs de conception et non une intention malveillante de la part du développeur du logiciel. Par ailleurs un programme peut envoyer des données personnelles à son éditeur dans le but de servir de la publicité ciblée. Si cette fonctionnalité est clairement indiquée à l'utilisateur, le programme peut tout de même être considéré comme légitime. À l'opposé un logiciel dont le but est d'exposer la vie privée de l'utilisateur sans le prévenir est malveillant. Nous proposons donc la définition suivante pour un logiciel malveillant.

Un logiciel est dit malveillant s'il réalise volontairement et à l'insu de l'utilisateur des opérations allant à l'encontre de l'intérêt de celui-ci.

Dans la pratique la différence entre un logiciel légitime et un logiciel malveillant est que ce dernier cherche à dissimuler son existence et son action sur le système. Il déploie à cette fin des techniques de protection logicielle rendant son analyse plus difficile que celle d'un programme légitime. Le programme malveillant cherche à exécuter une action finale, appelée *charge utile*, sur sa cible tout en la dissimulant à des analystes éventuels.

Analyse et détection de logiciels malveillants

Les mécanismes de protection employés, ou techniques d'obscurcissement, sont tels qu'aucune analyse basée sur une observation passive du programme ne peut permettre de prédire sa charge utile et donc de décider s'il est malveillant. La principale difficulté rencontrée lors de l'analyse d'un logiciel malveillant est qu'il est auto-modifiant : il se décompresse en réécrivant sur lui-même en mémoire. L'auto-modification complexifie grandement la sémantique du programme puisqu'une instruction peut-être amenée à être modifiée au cours de l'exécution de celui-ci. Afin de comprendre cette technique il est nécessaire d'utiliser une sémantique compatible avec l'auto-modification. Nous avons repris les analyses développées par Reynaud [Rey10] et Calvet [Cal13] permettant de séparer l'exécution d'un

programme auto-modifiant en plusieurs parties non auto-modifiantes.

Le désassemblage consiste en la récupération du code assembleur du programme analysé. En plus de la liste des instructions du programme, nous cherchons à déterminer le graphe de flot de contrôle qui représente les enchaînements possibles entre les différentes instructions désassemblées. Le premier objectif de cette thèse est donc de résoudre le problème suivant.

Problème 1. *Désassembler et reconstruire le graphe de flot de contrôle d'un programme obscurci.*

Une fois ces éléments extraits du programme analysé, même partiellement, nous nous interrogeons sur la détection d'un comportement malveillant au sein du programme. Les premiers travaux traitant de virologie informatique datent de 1986 avec Cohen [Coh86] puis Adleman [Adl88] en 1988. Ils s'intéressent particulièrement au comportement auto-reproducteur de certains programmes. Déterminer si un programme a un comportement auto-reproducteur est un problème indécidable. De même, il n'existe pas de programme capable de décider, sans jamais se tromper, si un programme analysé a un comportement malveillant.

Les décennies qui ont suivi ont vu apparaître différentes techniques de détection partielles dont la plus répandue est l'analyse de signatures syntaxiques. Ces approches consistent dans un premier temps à extraire d'un corpus de logiciels connus pour être malveillants certaines caractéristiques : la présence de certaines chaînes de caractères dans le fichier contenant le programme ou l'utilisation de certaines instructions dans un ordre précis, par exemple. Dans un second temps, pour classifier un logiciel inconnu, il suffit de regarder s'il possède une des caractéristiques extraites de ce corpus. Cette technique possède le double avantage d'être rapide et de générer peu de fausses alarmes.

Cependant chaque souche originale d'un logiciel malveillant est généralement déclinée en de nombreuses versions dont les caractéristiques varient. Ces souches forment une famille de logiciels malveillants. Pour éviter la détection par signatures syntaxiques, il suffit souvent d'insérer du code inutile ou de réorganiser le code : le nouveau logiciel malveillant est identique fonctionnellement au code initial mais les signatures précédemment extraites n'y sont plus présentes.

Nous nous sommes intéressés à la technique de détection initiée par Kaczmarek [BKM09]. Il s'agit d'une technique de détection par signatures basée sur la comparaison des graphes de flot de contrôle. La comparaison des graphes de flot de contrôle, ou analyse morphologique, consiste en la détection d'isomorphismes de graphes entre des parties des graphes de flot considérés. L'algorithme utilisé à l'origine fonctionnait par reconnaissance d'arbres à l'aide d'un automate d'arbres. Il n'existait pas de cadre formel définissant les objets détectés par cette analyse. Sans cadre formel il était difficile de mesurer les performances des implémentations utilisées en termes de précision des résultats. La seconde partie de notre travail consiste donc à apporter des solutions au problème suivant.

Problème 2. *Formaliser et optimiser la reconnaissance de graphes dans l'optique de la détection de programmes malveillants.*

Contributions

Notre objectif principal est donc le désassemblage et la reconstruction du graphe de flot de contrôle d'un programme malveillant obscurci. Nous cherchons à contrer particulièrement deux méthodes d'obscurcissement. La première est l'auto-modification, technique permettant aux programmes de cacher leur charge utile et de ne la révéler que juste avant son exécution. La seconde est le chevauchement de code, permettant à plusieurs instructions d'être codées sur des adresses communes.

Nous proposons une méthode d'analyse hybride basée sur une trace d'exécution qui permet de guider l'analyse statique. La trace d'exécution est récupérée, comme dans les travaux de Reynaud [Rey10], par une analyse dynamique qui découpe l'exécution en parties successives et non auto-modifiantes du programme, appelées vagues. La trace restreinte à une vague ne présente pas d'auto-modification, ce qui permet l'emploi de méthodes d'analyse statique sur chacune des vagues. Cette analyse statique fait une utilisation intensive des informations contenues dans la vague afin de guider son analyse. Elle reprend certaines techniques détaillées par Krügel [Krü+06] mais fonctionne sur des binaires utilisant le chevauchement de code et permet d'en mesurer l'utilisation. Nous proposons une sémantique pour le chevauchement de code en rangeant les instructions dans des couches de code au sein desquelles il n'y a pas de chevauchement.

Nous détaillons une implémentation de cette technique d'analyse hybride et validons des expériences précédentes, comme celles menées par Calvet [Cal13], montrant l'utilisation presque systématique de l'auto-modification par les programmes malveillants. Nous mettons en lumière l'utilisation occasionnelle du chevauchement de code par certains logiciels de protection de binaires.

La seconde partie de notre travail est centrée sur la détection de programmes malveillants et la technique d'analyse morphologique [BKM09], consistant à comparer les graphes de flot de contrôle de programmes connus pour être malveillants à celui du programme que l'on cherche à analyser. L'objectif est de comprendre les mécanismes permettant d'améliorer les performances du détecteur sachant qu'il cherche à résoudre un cas simplifié du problème NP-complet de l'isomorphisme de sous-graphes.

Nous proposons une formalisation du problème exactement résolu par l'analyse morphologique : il s'agit d'un problème plus simple que celui de l'isomorphisme de sous-graphes et pour lequel il existe des solutions en temps polynomial. Nous détaillons alors un algorithme de détection dont le temps d'exécution croît linéairement avec le nombre de programmes malveillants connus. Cet algorithme est complet : il résout exactement le problème posé par l'analyse morphologique. Nous décrivons également la conception et l'implémentation d'un algorithme incomplet mais dont le temps d'exécution ne dépend pas du nombre de programmes malveillants connus.

Enfin nous cherchons à appliquer cette approche à des cas concrets d'analyse de programmes. Une possibilité consiste à utiliser l'analyse morphologique pour la détection de similarités logicielles et en particulier l'utilisation de bibliothèques logicielles. Nous illustrons cette idée avec un logiciel malveillant, Waledac, et son emploi d'OpenSSL. Ces

travaux ont été publiés à REcon [Bon+12a] et Malware [Bon+12b].

Nous utilisons également l'analyse morphologique sur les programmes malveillants Duqu et Stuxnet dont nous avons montré qu'ils ont du code en commun. Dans une optique de détection nous nous interrogeons sur la possibilité de détecter Duqu connaissant Stuxnet et montrons qu'il est nécessaire de surveiller l'exécution de Duqu pour réagir à l'injection d'un code en mémoire, code que le détecteur morphologique est capable de relier à Stuxnet. Ces travaux ont fait l'objet de publications à Malware [Bon+13a] et SSTIC [Bon+13b].

Organisation du document

Nous commencerons, dans une première partie, par définir les notions d'assembleur et de désassemblage qui seront utilisées dans la suite du manuscrit, puis détaillerons une technique d'analyse dynamique, puis statique, de programmes obscurcis. Cette partie se terminera par la combinaison des deux approches et la présentation de notre outil de désassemblage.

La seconde partie portera plus particulièrement sur l'analyse morphologique et les algorithmes de comparaison de graphes. Nous reviendrons sur les applications d'une telle méthode à la détection de programmes malveillants, à la détection de similarités logicielles et à l'analyse pratique de programmes malveillants avec l'exemple de Duqu et Stuxnet.

Première partie

Désassemblage et analyse de binaires

Chapitre 1

Assembleur

Cette partie est consacrée à l'analyse d'un programme utilisant des méthodes de protection. Dans ce chapitre nous expliquons dans un premier temps ce qu'est un programme binaire et les spécificités du langage assembleur dans lequel ces programmes sont écrits. Dans un second temps nous définissons les objectifs de l'analyse d'un programme binaire ainsi que les difficultés rencontrées.

1.1 Compilation et fichiers exécutables

Un exécutable est en général d'abord écrit dans un langage de haut niveau. Chacun de ses modules est ensuite compilé en un fichier encodant le langage assembleur spécifique à la machine. La dernière étape est l'édition de liens qui consiste à regrouper tous ces fichiers compilés en un exécutable unique.

Prenons l'exemple d'un simple `Hello World` en C (Figure 1.1). Il est uniquement composé d'un appel à la fonction `printf` permettant l'affichage, à l'exécution, de la chaîne de caractère "Hello, world.". Une implémentation possible en assembleur NASM x86 (32 bits) pouvant tourner sous une distribution GNU/Linux est donnée en figure 1.2. Il est alors composé de deux appels systèmes vers le noyau Linux : un premier (sys_write) permettant l'affichage de la chaîne et un second (sys_exit) permettant de fermer le processus. On peut déjà remarquer que le programme est séparé en une section de données (`.data`) contenant la chaîne de caractère à afficher et une section de code (`.text`) contenant le code assembleur à exécuter.

Le fichier binaire exécutable résultant de la compilation est un exécutable binaire pour Linux, sous format ELF. Comme indiqué sur la figure 1.3(a), il contient des entêtes dans lesquels sont indiquées des informations générales sur le binaire telles que le point

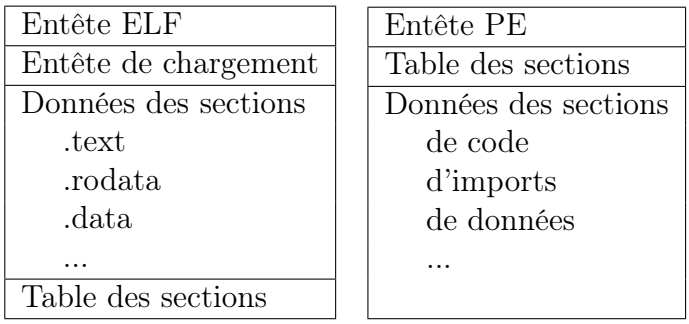
```c
int main(int argc, char* argv[]){
  printf("Hello, world.");
}
```

FIGURE 1.1 – Code C de `Hello World`

```
section .data
msg     db      "Hello, world", 0xa     ; La chaîne à afficher
len     equ     13                      ; La taille de la chaîne

section .text
global _start

_start:
; Afficher la chaîne de caractères
mov     eax, 4          ; Numéro d'appel système (sys_write)
mov     ebx, 1          ; Premier argument : le fichier de sortie (ici stdout)
mov     ecx, msg        ; Second argument : un pointeur vers la chaîne à afficher
mov     edx, len        ; Troisième argument : la taille de la chaîne
int     0x80            ; Appel au noyau

; Fermer proprement le programme
mov     eax, 1          ; Numéro d'appel système (sys_exit)
mov     ebx, 0          ; Premier argument : le code de retour (0 : normal)
int     0x80            ; Appel au noyau
```

FIGURE 1.2 – Code assembleur x86 (32 bits) de `Hello World`

d'entrée du programme, des informations sur les différentes sections du programme (leur taille, leurs adresses) et les sections : ici une section `.data` (Figure 1.4) contient les données du programme (dont la chaîne de caractères "Hello World") et une section `.text` (Figure 1.5) contenant le code assembleur à exécuter. L'entête de chargement (ou *program header table*) contient des informations supplémentaires pour un binaire amené à être chargé en mémoire à une adresse spécifique. À l'instar de la figure 1.3(a) donnant une structure simplifiée du format ELF pour Linux, la figure 1.3(b) donne un aperçu du format des fichiers exécutables pour un binaire x86 sous Windows (format PE).



(a) Fichier ELF (b) Fichier PE

FIGURE 1.3 – Format des exécutables ELF (Linux) et PE (Windows)

Emplacement dans le fichier	Adresses de chargement	Octets	Caractères ascii
94	80490a4	48 65 6c 6c 6f 2c 20	H e l l o ,
9b	80490ab	77 6f 72 6x 64	W o r l d
a0	80490b0	0a	Fin de chaîne

FIGURE 1.4 – Section `.data` de `Hello World`

Emplacement dans le fichier	Adresses de chargement	Octets	Instruction
80	8048080	b8 04 00 00 00	mov eax,0x4
85	8048085	bb 01 00 00 00	mov ebx,0x1
8a	804808a	b9 a4 90 04 08	mov ecx,0x80490a4
8f	804808f	ba 11 00 00 00	mov edx,0x11
94	8048094	cd 80	int 0x80
96	8048096	bb 00 00 00 00	mov ebx,0x0
9b	804809b	b8 01 00 00 00	mov eax,0x1
a0	80480a0	cd 80	int 0x80

FIGURE 1.5 – Section `.text` de `Hello World`

Analyse de binaires. La principale difficulté lors de l'analyse d'un programme malveillant est que le code source n'est pas disponible à l'analyste qui doit se contenter du fichier binaire compilé. Un programme compilé se présente donc sous la forme d'un fichier binaire contenant le code machine devant être lancé à l'exécution du programme ainsi que des informations de chargement du binaire : la distinction de différentes sections, les adresses mémoires auxquelles le système devra les charger en mémoire, ainsi que les bibliothèques logicielles du système dont il a besoin et qui devront être chargées en mémoire à l'exécution.

La principale tâche de l'analyste est alors d'extraire du fichier binaire les informations utiles et surtout d'analyser les parties de code assembleur de l'exécutable. Nous détaillerons, dans la suite de ce chapitre, quelques spécificités du langage assembleur considéré et les difficultés rencontrées par l'analyste.

1.2 Assembleur x86 et x86_64

L'architecture la plus fréquente pour les processeurs des ordinateurs personnels est celle des processeurs Intel CISC avec le jeu d'instructions x86 pour les machines adressant la mémoire sur 32 bits, et le jeu d'instructions x86_64 pour celles adressant la mémoire sur 64 bits.

Nous allons présenter deux approches historiques pour l'architecture d'une machine et expliquerons quelles notions de ces deux architectures sont présentes dans les processeurs actuels.

Architecture de Harvard et de von Neumann. L'architecture de Harvard [IBM], sépare le code exécutable des données en deux mémoires distinctes. La première implémentation de l'architecture de Harvard était L'ASCC (*Automatic Sequence Controlled Calculator*) d'IBM, également appelé le Mark I, et considéré comme le premier calculateur universel, en 1944. Il lisait les instructions sur des cartes perforées et les données étaient entrées manuellement à l'aide d'interrupteurs. Ainsi le code exécutable était physiquement non modifiable et séparé des données.

L'architecture de von Neumann, nommée en référence aux travaux de John von Neumann, John William Mauchly et John Eckert en 1945, acceptait la modification de la logique des programmes [Tim]. Elle était cependant limitée à l'utilisation d'un seul bus de données entre le processeur et la mémoire. Cette restriction limitait grandement les capacités de lecture et d'écriture mémoire d'une machine utilisant le modèle de von Neumann.

L'architecture utilisée actuellement pour les ordinateurs personnels est un mélange des deux approches. Elle utilise plusieurs bus mémoire mais les instructions et les données sont stockées dans la même mémoire et donc accessibles autant en lecture qu'en écriture [Tim]. Dans ce modèle la machine est articulée autour du processeur et de son unité de contrôle chargée de synchroniser les autres composants, d'exécuter les instructions du binaire chargé en mémoire, de gérer les entrées et les sorties, de lire et d'écrire dans la mémoire et les registres. Les registres ne peuvent contenir que des entiers dans les intervalles $[-2^{31},\ 2^{31}-1]$ ou $[-2^{63},\ 2^{63}-1]$ en assembleur x86 et x86_64 respectivement. L'unité arithmétique et logique opère toute l'arithmétique du processeur et modifie les drapeaux du registre d'état en conséquence selon les résultats des opérations effectuées. Une organisation simplifiée d'une machine utilisant ce modèle est donnée Figure 1.6.

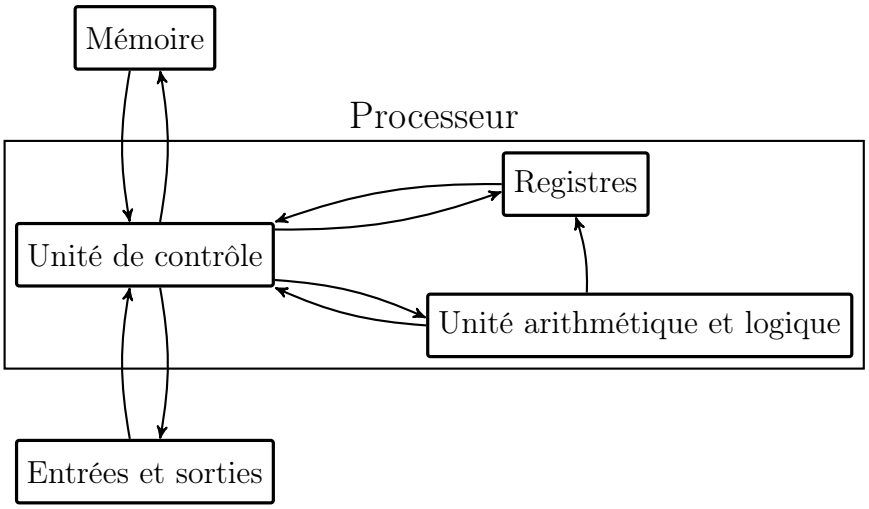


FIGURE 1.6 – Architecture simplifiée

Structure de la mémoire. La mémoire physique de la machine peut-être faite, entre autres, de mémoire volatile (mémoire vive) et de supports amovibles (disques durs). Chacun de ces supports peut être vu comme une liste d'adresses mémoire où l'on peut stocker

des données et le système d'exploitation gère ces supports comme il l'entend. Plus précisément un programme n'est pas nécessairement chargé sur une plage contiguë d'adresses mémoire. Pour éviter au programmeur de devoir gérer des adresses mémoires qui ne concernent pas son programme, un mécanisme de mémoire virtuelle est mis en place. De son point de vue, le programme courant voit sa mémoire comme un intervalle d'adresses mémoires contiguës. C'est le système d'exploitation qui traduit les adresses virtuelles en adresses physiques lors de l'exécution du programme. Cette technique permet au système d'exploitation de choisir, de manière transparente pour le programme, un support différent pour certaines parties du programme. Elle lui permet aussi de gérer plus finement l'accès à la mémoire, tant pour interdire l'accès à certaines zones que pour partager des zones entre plusieurs processus.

En pratique, lors de l'exécution d'un programme, des informations peuvent être stockées en plusieurs lieux. Tout d'abord les registres du processeur permettent un accès rapide à quelques variables. Certains registres sont réservés, souvent par convention. Par exemple, lors d'un appel de fonction, la convention par défaut (CDECL) stipule que la valeur de retour est passée dans le registre `eax`. La seconde structure de mémoire est la pile. Il s'agit d'une structure de type LIFO (*Last In First Out*) dans laquelle les mots (de 32 bits en x86, 64 en x86_64) sont empilés à l'aide de l'instruction `push` et dépilés avec l'instruction `pull` de sorte que le mot dépilé est celui qui a été empilé en dernier. Les variables locales sont en général enregistrées sur la pile et, lors d'un appel de fonction, les arguments sont passés sur la pile. La dernière structure est le tas qui est géré par l'appel de fonctions d'allocations dynamiques (telles `malloc` en C) et est généralement utilisé pour entreposer les structures mémoire plus encombrantes telles que des tableaux ou des structures complexes. En pratique la mémoire d'un processus contient d'abord les sections de code, puis les sections de données, puis le tas qui est susceptible de s'étendre ainsi que la pile qui peut également s'étendre, dans le sens inverse (Figure 1.7).

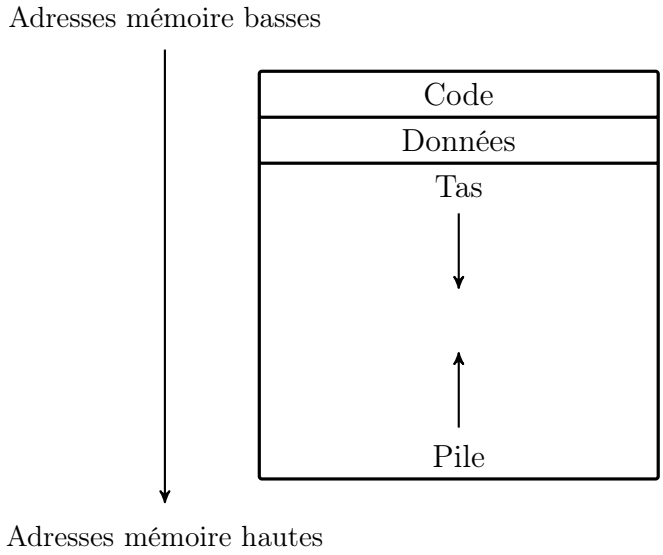


FIGURE 1.7 – Organisation de la mémoire d'un processus

Jeu d'instructions. Les processeurs Intel x86 utilisent un jeu d'instruction complexe (CISC). La représentation d'une instruction en une suite d'octets inclut principalement un code d'opération (ou *opcode*) et ses arguments. Des informations supplémentaire peuvent être codées dans l'instruction, comme des préfixes. Le format détaillé des instructions est donné à la figure 1.8. L'instruction `mov ecx, 0x080490a4` du programme précédent, codée sur les octets `b9 a4 90 04 08` est composée du code d'opération `b8+r` indiquant une opération de copie d'une valeur immédiate vers un registre. Ici r est égal à 1, ce qui indique que le registre concerné est `ecx`. La valeur à copier vers le registre est `0x080490a4` soit les octets `a4 90 04 08` codés en petit-boutistes (*little endianness*), c'est à dire que les mots de poids faible sont écrits en premier, à l'inverse de l'écriture usuelle. Le décalage permet de modifier une instruction d'adressage indirect comme `mov ecx, [eax]`, écrivant le contenu en mémoire à l'adresse `eax` dans le registre `ecx` en `mov ebx, [eax+d]` où `d` est le décalage. Enfin les préfixes permettent par exemple de répéter plusieurs fois l'instruction jusqu'à ce qu'une condition sur un registre soit remplie.

	Préfixe 1 à 4	Opcode 1 à 3	Modificateur 0 à 2	Décalage 0 à 4	Valeur immédiate 0 à 4
Exemples :					
mov ecx, 0x080490a4		b9			a4 90 04 08
add eax, 2		83	c0		02

FIGURE 1.8 – Format d'une instruction x86 et taille des opérandes en octets

Intuition de sémantique. Bien que la documentation exhaustive d'Intel pour les processeurs x86 et x86_64 [Inta] détaille plusieurs centaines d'instructions, elles peuvent être regroupées en 3 classes informelles. Prenons l'instruction `mov` : elle sert à copier des informations depuis une zone (mémoire ou registre) vers une autre zone. Ainsi `mov eax, [ebx+10]` copie la valeur à l'adresse `ebx+10` dans le registre `eax`. Cette instruction est déclinée en plusieurs dizaines de variantes selon la taille et le type des opérandes copiés.

Une seconde catégorie d'instructions permet de gérer les opérations arithmétiques sur les octets en mémoire et dans les registres. Une instructions comme `add eax, ebx` ajoute la valeur de `ebx` à celle de `eax` et stocke le résultat de l'opération dans `eax`. Ce type d'opération combine en fait une opération arithmétique et une assignation. On peut y inclure l'instruction `cmp` qui compare deux valeurs et met à jour les drapeaux de tests. Ces drapeaux indiquent entre autres le signe du résultat d'une opération et si elle a provoqué un débordement d'entier.

Les instructions précédemment décrites sont séquentielles et ne modifient pas l'ordre d'exécution des instructions. Après l'exécution d'une instruction de type `mov` à l'adresse a codée sur n octets, l'instruction située à l'adresse $a + n$ sera exécutée. Le dernier type d'instructions modifie le flot de contrôle du programme : elles effectuent des sauts à des adresses spécifiées dans le programme. Ces sauts peuvent être inconditionnels comme un `jmp 8048085` qui provoque un saut à l'adresse `0x8048085` ou le saut dynamique `jmp eax` dont l'adresse cible du saut est la valeur de `eax`. Les sauts peuvent aussi être conditionnels :

l'instruction `je +10`, à l'adresse a et de taille n, sera suivie de l'instruction à l'adresse $a + 10$ si le registre ZF vaut 1 et de l'instruction à l'adresse $a + n$ dans le cas contraire.

1.3 Désassemblage

Analyser un programme sous forme binaire est en général plus difficile que d'analyser le code source du programme écrit dans un langage de haut niveau. La compilation provoque une perte d'informations de haut niveau. Par exemple en assembleur il n'y a plus de noms ni de types pour les variables et les fonctions.

Le désassemblage est l'opération inverse de l'assemblage : il consiste à récupérer le code assembleur source du binaire. Cette tâche semble particulièrement simple puisque l'assemblage consiste à trouver les octets correspondants à chaque instruction à l'aide de la documentation officielle du processeur puis à mettre en forme le fichier binaire en remplissant correctement ses entêtes et ses sections. Pourtant on a vu que, dans le modèle de von Neumann, les données et le code peuvent être mêlés. En particulier les parties de données (comme la section `.data`) peuvent être exécutées. De même les sections de code peuvent contenir des informations destinées à être simplement lues ou modifiées mais jamais exécutées. La difficulté du désassemblage consiste donc à séparer les parties potentiellement exécutables des données.

Ainsi un désassemblage a priori cohérent de `Hello World` consisterait à considérer les deux sections comme contenant du code potentiellement exécutable. Ce désassemblage sera composé de la section `.text` d'origine (figure 1.5) et de la section `.data` (figure 1.9).

Emplacement dans le fichier	Adresses de chargement	Octets	Instruction
94	80490a4	48	dec eax
95	80490a5	65	gs
96	80490a6	6c	ins BYTE PTR es :[edi],dx
97	80490a7	6c	ins BYTE PTR es :[edi],dx
98	80490a8	6f	outs dx,DWORD PTR ds :[esi]
99	80490a9	2c 20	sub al,0x20
9b	80490ab	77 6f	ja 0x804911c
9d	80490ad	72 6c	jb 0x804911b
9f	80490af	64	fs
a0	80490b0	0a	.byte 0xa

FIGURE 1.9 – Section `.data` de `Hello World` désassemblée comme du code

Ce désassemblage n'est pas correct et on peut le prédire. On connaît le point d'entrée du binaire, qui est l'adresse `8048080` dans la section `.text`. Les instructions exécutées dans la section `.text` à la suite du point d'entrée sont séquentielles et ne peuvent détourner l'exécution vers la section `.data`. Le code désassemblé dans cette section n'est donc pas atteignable. Dans ce cas on sait alors que la section `.data` ne contient pas de code et ne doit pas être désassemblée.

En pratique un programme est constitué d'un grand nombre d'instructions de saut et notamment de sauts dynamiques de type `jmp eax`. Ce type d'instructions pose un souci lors de l'analyse : sans connaître précisément la ou les valeurs potentielles d'`eax`, il est impossible de prédire la cible du saut et donc de savoir le code potentiellement exécuté à la suite de cette instruction. Dans l'exemple du `Hello World`, si une telle instruction est présente dans la section `.text`, il devient difficile d'exclure l'exécution possible de code dans la section `.data`. Une des principales difficultés lors du désassemblage réside dans la présence de ces sauts dynamiques dont la cible ne peut être déterminée qu'en ayant une connaissance fine de la ou des valeurs possibles des différents registres.

Dans cette partie nous nous intéressons au problème du désassemblage, tant d'un point de vue théorique en définissant ses objectifs et les obstacles rencontrés, que d'un point de vue pratique en expliquant les différentes techniques mises en œuvre pour l'effectuer.

Nous utilisons les notions expliquées précédemment et l'architecture simplifiée que nous avons décrite, dans laquelle les instructions et les données ne sont pas séparées dans la mémoire. Certains programmes, dits auto-modifiants, utilisent ce fait pour modifier leur code au fur et à mesure de leur exécution, rendant leur analyse plus difficile. Un des objets de cette thèse est l'étude de ces programmes : nous détaillerons dans les chapitres suivants leur fonctionnement et certaines techniques d'analyse.

Cependant, dans la suite de ce chapitre introductif, nous considérons que les programmes étudiés ne sont pas auto-modifiants. Cela nous permet de définir les notions de désassemblage et de graphe de flot de contrôle dans un cas plus simple tandis que nous étendrons ces définitions dans les chapitres suivants. En conséquence, dans ce chapitre, l'intégralité du code des programmes est observable dès leur chargement en mémoire et ce code n'est pas amené à être modifié au cours de l'exécution.

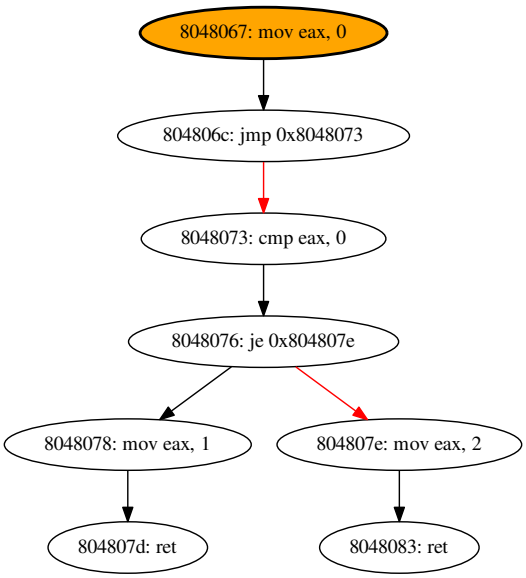
1.3.1 Reconstruction du graphe de flot de contrôle

Nous avons utilisé jusqu'ici une représentation des programmes assembleur et binaires sous forme de code linéaire mais il est plus aisé de visualiser les programmes à l'aide d'un graphe de flot de contrôle (GFC). Ce graphe représente les instructions sous forme de sommets et les sauts du flot de contrôle d'une instruction à une autre sous forme d'arcs. Prenons le programme assembleur donné en figure 1.10(a). Il ne fait que modifier le registre `eax`, effectue des sauts selon la valeur d'`eax` et se termine avec une valeur pour `eax` à 1 ou 2. L'instruction `mov eax, 3` n'est pas atteignable puisqu'elle suit un saut inconditionnel et qu'aucune autre instruction ne provoque un saut vers elle. Ainsi le graphe de flot de contrôle de ce programme est celui donné en figure 1.10(b).

Le sommet coloré en orange est le point d'entrée du programme. Dans le cas où le flot de contrôle peut passer de l'instruction a à l'instruction b, l'arc entre a et b est en noir si b suit a en mémoire et en rouge dans le cas contraire, c'est à dire s'il s'agit d'un saut.

Adresse	Octets	Instruction
8048067	b8 00 00 00 00	mov eax,0x0
804806c	eb 05	jmp 0x8048073
804806e	b8 03 00 00 00	mov eax,0x3
8048073	83 f8 00	cmp eax,0x0
8048076	74 06	je 0x804807e
8048078	b8 01 00 00 00	mov eax,0x1
804807d	c3	ret
804807e	b8 02 00 00 00	mov eax,0x2
8048083	c3	ret

(a) Code assembleur



(b) Graphe de flot de contrôle

FIGURE 1.10 – Représentation d'une fonction assembleur modifiant `eax`

1.3.2 Parcours linéaire

Un désassemblage suivant un parcours linéaire désassemble, pour chaque section, la première instruction à l'adresse a puis l'instruction à l'adresse $a+k$ où k est le nombre d'octets sur lesquels l'instruction est codée. Ainsi pour une suite d'instructions séquentielles cette méthode suit le flot de contrôle mais si une instruction de type `jmp` est rencontrée, le désassemblage s'intéressera à l'instruction qui suit en mémoire et non celle qui sera logiquement exécutée ensuite. La figure 1.10(a), donnée précédemment, est un exemple de désassemblage linéaire d'une fonction assembleur.

L'algorithme 1.1 explicite le parcours linéaire d'un programme composé de plusieurs sections étant chacune un bloc d'octets présent à une adresse mémoire. Un désassemblage consiste en la donnée d'un ensemble de couples dont chaque couple représente une instruction assembleur : le premier élément du couple est l'adresse où l'instruction est présente et le second élément est l'instruction x86 ou x86_64 étant à cette adresse. Nous désassemblons entre la première et la dernière adresse mémoire de chaque section et supposons que l'on dispose de deux opérateurs supplémentaires *decode* et *taille*. Le premier fournit, à partir d'une adresse et d'un programme, l'instruction présente à cette adresse dans le programme. Le second indique la taille d'une instruction, c'est à dire le nombre d'octets

sur lesquels elle est codée.

Algorithme 1.1 : Désassemblage linéaire d'un programme P composé de plusieurs sections

Entrées : Un programme P composé de plusieurs sections
Résultat : Le désassemblage linéaire de P

```
desassemblageLineaire(P)
    D ← ∅
    pour toute section S de P faire
        // On désassemble section par section
        D ← D ∪ desassemblageSection(S, P)
    fin
    retourner D
desassemblageSection(S, P)
    // On va désassembler entre la première et la dernière adresse
    mémoire où S est mappée
    debut ← premiereAdresse(S)
    fin ← dernieresAdresse(S)
    a ← debut
    D ← ∅
    tant que a ≤ fin faire
        // On récupère l'instruction de P présente à l'adresse a
        I ← decode(a, P)
        D ← D ∪ {(a, I)}
        a ← a + taille(I)
    fin
    retourner D
```

L'avantage de cette technique réside dans sa simplicité et le fait qu'elle couvre l'ensemble de la mémoire. Pourtant, ne cherchant pas à suivre le flot de contrôle, elle ne permet pas de séparer le code exécutable des données et désassemble des instructions qui ne sont pas atteignables comme l'instruction `mov` à l'adresse 804806e de la figure 1.10(a). C'est l'approche utilisée par le désassembleur standard du projet GNU, objdump [Obj]. La commande de désassemblage par défaut du désassembleur interactif libre Radare [rad] fonctionne également par parcours linéaire.

1.3.3 Parcours récursif

À l'inverse du parcours linéaire, le parcours récursif suit le flot de contrôle et désassemble les instructions qui se suivent logiquement en partant du point d'entrée du programme. Le graphe de flot de contrôle peut être déduit du parcours récursif : la figure 1.10(b) donne celui de l'exemple précédent.

L'algorithme 1.2 détaille le parcours récursif d'un programme ayant un point d'entrée. Nous avons cette fois besoin de connaître les enchaînements possibles à la suite de l'exécution d'une instruction : nous ajoutons alors l'opérateur $fils$ qui, à partir d'une instruction I, renvoie l'ensemble des adresses des instructions pouvant être exécutées à la suite de I.

Une instruction de saut conditionnel aura typiquement deux fils : le premier est atteint si la condition est remplie et on atteint le second dans le cas contraire.

Algorithme 1.2 : Désassemblage récursif d'un programme P

Entrées : Un programme P de point d'entrée *ep*
Résultat : Le désassemblage récursif de P

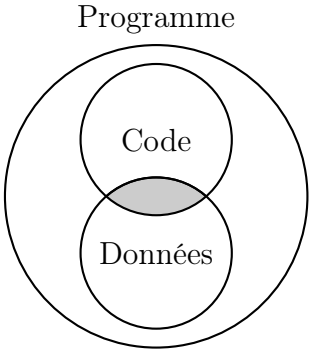
```
desassemblageRecursif(P)
    // On désassemble à partir du point d'entrée de P
    retourner desassemblage(ep, ∅, P)

desassemblage(a, D, P)
    I ← decode(a, P)
    D ← D ∪ {(a, I)}
    // On désassemble récursivement tous les fils de I qui n'ont pas
    déjà été parcourus
    pour a' ∈ fils(I) faire
        si a' n'est pas présent dans D alors
            D ← D ∪ desassemblage(a', D, P)
        fin
    fin
    retourner D
```

L'avantage de cette technique est qu'elle ne désassemble que des instructions dont on peut raisonnablement penser qu'elles sont atteignables. Cependant certains chemins ne sont pas atteignables en raison de variables définies. L'instruction à l'adresse `8048076` provoque un saut vers l'adresse `804807e` si la comparaison effectuée à l'instruction précédente est vraie, c'est à dire si `eax` $= 0$, et saute vers l'adresse `8048078` dans le cas contraire. Or la première instruction de la fonction donne à `eax` la valeur 0. Dans tous les cas le saut se produira donc vers l'adresse `804807e` et l'autre branche n'est pas atteignable.

D'autre part si des sauts dynamiques sont utilisés comme avec l'instruction `jmp eax`, le parcours récursif s'arrête alors qu'il est clair que d'autres instructions seront exécutées. La difficulté théorique vient du fait que déterminer l'ensemble des valeurs possibles pour `eax` est dans certains cas un problème indécidable. Le désassembleur commercial IDA [Hexb] fonctionne par parcours récursif.

1.3.4 Désassemblage parfait et graphe de flot de contrôle parfait

Il est aisé de définir ce que l'on appelle code lors d'un désassemblage : il s'agit de toute partie du programme sur laquelle est écrite une instruction atteignable lors d'une exécution du programme. Les données sont les parties du programme qui peuvent être lues lors d'une exécution du programme. Une partie du programme peut être à la fois lue et exécutée et donc être à la fois de la donnée et du code. Une adresse d'un binaire chargé en mémoire peut donc être soit inutilisée, soit atteignable lors de l'exécution, soit potentiellement lue, soit potentiellement lue et exécutée.



Une définition formelle du désassemblage parfait d'un programme P (définition 1.1) est alors la détermination de l'ensemble des instructions atteignables de P, munies de l'adresse à laquelle elles sont présentes dans P. Nous utilisons, comme précédemment, l'opérateur *decode* qui, à partir d'une adresse et d'un programme, renvoie l'instruction présente à cette adresse dans le programme.

Définition 1.1. *Étant donné un programme P non auto-modifiant prenant une entrée I, le désassemblage parfait de P est la donnée de* $\bigcup_I \{(a, decode(a, P)),\ P(I) \text{ atteint } a\}$.

L'analyste est intéressé par davantage d'informations que la seule liste des instructions atteignables. L'enchaînement des instructions lors des différentes exécutions du programme est une donnée primordiale. Reprenons l'exemple de code assembleur de la figure 1.10(a). Une exécution possible consiste à prendre les instructions `0x8048067`, `0x804806c`, `0x8048073`, `0x8048076`, `0x804807e`, `0x8048083` dans cet ordre. En fait, quelles que soient les valeurs initiales de `eax`, des autres registres ou de la mémoire, le chemin précédent est toujours emprunté : il s'agit du seul chemin d'exécution possible. La meilleure source pour l'analyse d'un programme serait cette connaissance de tous les chemins d'exécution possibles, quelle que soit l'entrée donnée au programme (définition 1.2).

Définition 1.2. *Étant donné un programme P prenant une entrée I, l'ensemble des chemins d'exécution possibles de P est l'union sur les entrées I des chemins d'exécution pris par P(I).*

À partir de cette donnée il est possible de parcourir l'ensemble des comportements possibles du programme, que ce soit la valeur des différents registres au cours de l'exécution, les interactions entre le programme et le système d'exploitation ou les enchaînements possibles entre différentes instructions. Nous nous intéressons en particulier au graphe de flot de contrôle parfait (définition 1.3) dans lequel les sommets sont exactement les adresses de toutes les instructions atteignables. Dans ce graphe de flot de contrôle parfait, un arc relie deux instructions si et seulement si il existe un chemin d'exécution dans lequel les deux instructions se suivent immédiatement. Nous ne nous intéressons ici qu'aux programmes non auto-modifiants mais nous verrons par la suite comment étendre la notion de graphe de flot de contrôle parfait à un programme qui se modifie lui-même.

Définition 1.3. *Étant donné un programme non auto-modifiant P prenant une entrée I, le graphe de flot de contrôle parfait de P est le graphe orienté $T = (V, E)$ tel que :*

- *L'ensemble V des sommets de T est l'ensemble de couples (adresse, instruction) présents dans le désassemblage parfait de T*
- *$E \subset V \times V$ est l'ensemble des arcs de T*
- *$(a, b) \in E$ si et seulement si il existe un chemin d'exécution de P au sein duquel l'adresse b suit immédiatement l'adresse a dans l'ordre d'exécution.*

En pratique nous ne disposons pas de la liste exhaustive de tous les chemins d'exécution possibles du programme à analyser. Notre objectif est alors d'obtenir une approximation cohérente du graphe de flot de contrôle parfait.

1.3.5 Difficulté théorique du désassemblage

Si on cherche à séparer le code des données alors le but du désassemblage d'un programme P est de résoudre le problème suivant. On se limite dans un premier temps à un programme P non auto-modifiant.

Problème. *Soit a une adresse, existe-t-il une entrée I de P telle que P(I) atteint a ?*

Nous reprenons ici un argument tiré de la thèse de Joan Calvet [Cal13] montrant le caractère indécidable de ce problème en réduisant le problème de l'arrêt à celui-ci.

Notons CODE le programme vérifiant, pour tout programme P et adresse a, CODE(P, a) = 1 si et seulement si il existe une entrée I de P telle que P(I) atteint a. Dans le cas contraire CODE (P, a) = 0.

Soit P un programme ne prenant pas d'entrée et ayant un unique point d'arrêt indiqué par l'instruction `halt` à l'adresse α. CODE(P, α) = 1 si et seulement si P atteint l'adresse α, c'est à dire si et seulement si P termine.

On en déduit donc que le désassemblage permet de résoudre le problème de l'arrêt qui est pourtant connu pour être indécidable ; c'est absurde. Ainsi le problème du désassemblage d'un programme non auto-modifiant est indécidable. Le désassemblage d'un programme auto-modifiant est un cas plus difficile que celui traité ici : il s'agit également un problème indécidable.

1.3.6 Analyse statique et dynamique

Dans le domaine de l'analyse de code, on parle d'analyse dynamique lorsque le fichier binaire à analyser est exécuté au moins partiellement et que l'analyse consiste à observer une ou des exécutions du programme. Au contraire lors d'une analyse statique on cherche à inférer les propriétés du programme sans l'exécuter.

Les deux techniques permettent de récupérer du code assembleur à partir du programme et donc de réaliser un désassemblage et de reconstruire un graphe de flot de contrôle. L'avantage de l'analyseur dynamique est que, puisqu'il travaille sur une exécution spécifique du programme, il est précis : les instructions qui sont exécutées doivent être incluses dans le désassemblage. Les traces d'exécution fournissent ainsi une sous-approximation de l'ensemble des chemins d'exécution possibles. Par contre l'analyseur

dynamique n'est pas complet vu qu'il ne suivra pas certaines branches du programme qui pourraient être utilisées si leurs conditions étaient vérifiées. À l'inverse un analyseur statique n'est pas précis et ne peut en général pas être complet à cause de l'impossibilité de prédire certaines cibles de sauts dynamiques. En considérant que les sauts dynamiques dont les cibles sont inconnues peuvent aboutir à n'importe quelle adresse mémoire, l'analyse statique fournit une sur-approximation de l'ensemble des chemins d'exécution possibles.

En pratique le désassemblage est classiquement du domaine de l'analyse statique. Nous expliquerons comment l'analyse dynamique peut aider à reconstruire les graphes de flot et le code des programmes obscurcis.

1.4 Conclusion

Nous avons décrit les machines sur lesquelles les programmes que nous étudions s'exécutent. Un programme assembleur est généralement initialement écrit dans un langage de haut niveau puis compilé en un binaire encodant les instructions dans le langage assembleur cible.

Le principal problème du désassemblage réside dans la difficulté de séparer le code exécutable des données dans un fichier binaire : cette séparation est un problème indécidable. Le chapitre suivant présente plusieurs méthodes d'obscurcissement de code utilisées par les auteurs de logiciels malveillants pour compliquer davantage la tâche d'un analyste.

Chapitre 2

Techniques d'obscurcissement de code

L'analyse d'un logiciel malveillant a pour but de comprendre ses mécanismes internes : selon les logiciels il peut, entre autres, s'agir des techniques d'attaque, de communication avec d'autres instances du programme, ou de clés de chiffrement utilisées. Le programmeur a donc intérêt à protéger son logiciel contre l'analyse. Son but est de la rendre plus compliquée, nécessitant plus de ressources en temps ou en argent de la part de l'analyste.

De nombreuses techniques de protection sont applicables à un programme binaire pour rendre son analyse plus compliquée. Certaines rendent le code plus difficile à comprendre en ajoutant par exemple du code inutile. Une autre technique consiste à modifier le programme au cours de son exécution afin que le code réellement utile du binaire ne soit pas lisible à première vue : on parle alors d'auto-modification.

Un auteur de programmes malveillants peut produire dans un premier temps son programme sans protection puis utiliser un logiciel de protection qui produit un binaire sémantiquement équivalent mais qui est rendu plus difficile à analyser. En pratique l'exécutable final, protégé, combine plusieurs techniques d'obscurcissement dont des techniques d'auto-modification.

Dans ce chapitre nous chercherons rapidement à comprendre les difficultés rencontrées pour quiconque cherche à protéger son programme contre l'analyse. Dans un second temps nous nous intéresserons aux protections rencontrées lors de l'analyse de logiciels malveillants et en particulier aux problèmes liés au chevauchement de code assembleur et à l'auto-modification.

2.1 Théorie de l'obscurcissement

Collberg et Nagra [NC09] définissent plusieurs propriétés souhaitables pour une protection logicielle, nous reprenons ici quelques uns de leurs arguments. La première propriété est que le programme protégé soit équivalent en terme de sorties au programme d'origine. Une protection, ou obscurcissement, d'un programme P est une transformation $\mathcal{T}$ telle que le programme $\mathcal{T}(P)$ a le même comportement que P : quelle que soit l'entrée I de P, $\mathcal{T}(P)(I) = P(I)$. On souhaite également ne pas affecter de manière importante les performances du programme, ni en temps d'exécution ni en taille des binaires.

Deux formes d'obscurcissement formelles sont décrites dans la littérature [BR13] :

— L'obscurcissement de type boîte noire : $\mathcal{T}$ est un opérateur d'obscurcissement de type boîte noire si, quel que soit le programme P, l'analyse de $\mathcal{T}(P)$ ne fournit pas plus d'informations que ce à quoi aurait accès un oracle pouvant simplement utiliser P sur des entrées et en observer les sorties.

— L'obscurcissement indistinguable : $\mathcal{T}$ est un opérateur d'obscurcissement indistinguable si, pour toute classe de programmes $P_1, P_2, ...$ équivalents fonctionnellement, les programmes obscurcis $\mathcal{T}(P_1)$, $\mathcal{T}(P_2)$, ... sont indistinguables.

L'obscurcissement indistinguable est plus faible que l'obscurcissement de type boîte noire. L'obscurcissement de type boîte noire a été prouvé possible pour les programmes implémentant une fonction point [LPS04], c'est à dire renvoyant toujours la même valeur sauf en un point particulier : c'est le cas des fonctions vérifiant des mots de passe. Inversement il a été montré que certaines fonctions ne peuvent pas subir d'obscurcissement de type boîte noire [Bar+12] : il n'existe donc pas d'opérateur d'obscurcissement de type boîte noire universel. Garg, Gentry, Halevi, Raykova, Sahai et Waters [Gar+13] ont montré la possibilité de construire un opérateur d'obscurcissement indistinguable pour toute fonction représentée par un circuit de taille polynomiale. Ils utilisent pour cela des opérateurs de chiffrement homomorphe.

2.2 Exemples d'obscurcissement

De nombreuses techniques d'obscurcissement sont utilisées en pratique, nous en donnons quelques exemples ici.

Insertion de code mort. Insérer du code non atteignable (ou code mort) peut forcer un désassembleur par parcours linéaire à se désaligner avec le code réellement exécuté et à favoriser le code mort au détriment du code légitime. L'exemple donné en figure 2.1 montre du code assembleur avec deux octets de données placés à la suite d'une instruction `jmp`. Ces deux octets aux adresses $0x08048062$ et $0x08048063$ ne sont pas atteignables. Pourtant un désassembleur linéaire (Figure 2.2) chercherait à les désassembler et serait alors incapable de voir une partie des instructions réellement exécutées.

```
jmp suite
db      0a 05        ; Octets non atteignables
suite:
cmp ecx, 0x2
je 0x8048069
mov ebx, 0x2  ;  0x00000002
```

FIGURE 2.1 – Insertion de code mort dans du code légitime

```
08048060    eb 02                 jmp 0x8048064
08048062    0a 05 83 f9 02 74     or al, [0x7402f983]
08048068    00 bb 02 00 00 00     add [ebx+0x2], bh
```

FIGURE 2.2 – Sortie de `objdump` : L'insertion de code mort dupe facilement un désassemblage par parcours linéaire

Appels de fonctions sans retour. L'utilisation d'un contrôle de flot non standard peut forcer un désassembleur par parcours récursif à explorer du code non atteignable. Le comportement par défaut de l'instruction `call` à une adresse a et de taille n est d'empiler l'adresse de retour $a + n$ puis de sauter vers la première adresse de la fonction appelée. L'instruction `ret` placée à la fin de la fonction appelée dépile la première valeur de la pile et provoque un saut vers celle-ci.

Normalement la valeur dépilée lors du `ret` est $a+n$ afin que le flot d'exécution revienne à la fonction appelant. Ainsi un désassembleur récursif désassemble à partir de la cible du `call` ainsi que de l'adresse de retour.

Une technique classique d'obscurcissement [LD03 ; SH12] consiste à combiner l'empilement d'une adresse (`push` b) et l'instruction `ret`. Ces deux instructions provoquent un saut vers l'adresse b sans que le `ret` ne marque la fin de la fonction et le retour à l'instruction suivant le `call` : cette instruction, à l'adresse `0xa+n`, ne sera pas forcément atteinte. La transformation consiste alors à remplacer des instructions `jmp` par la séquence `push` puis `ret` puisque les deux suites d'instructions suivantes sont équivalentes.

`push 0xb` `ret`	`jmp 0xb`

Prédicats opaques. Lorsqu'un désassembleur récursif rencontre une instruction de saut conditionnel comme `je`, qui provoque un saut si les deux valeurs comparées sont égales, il cherche à désassembler à la fois la cible potentielle du saut comme l'instruction suivante, qui sera exécutée si la condition n'est pas remplie. Une autre technique courante d'obscurcissement [MKK08] consiste à utiliser comme condition du saut une relation que le programmeur sait toujours vraie ou fausse. De cette manière il prédit qu'une seule des deux branches est atteignable alors qu'un désassembleur va parcourir également la branche inutile. Une telle condition est appelée un prédicat opaque et peut être implémentée par des relations d'arithmétique. Par exemple en appliquant le petit théorème de Fermat : quel que soit l'entier e, $e^3 = e \ mod \ 3$. Le programmeur sait que l'égalité est toujours vérifiée mais un analyseur statique ne pourra pas le déterminer aisément.

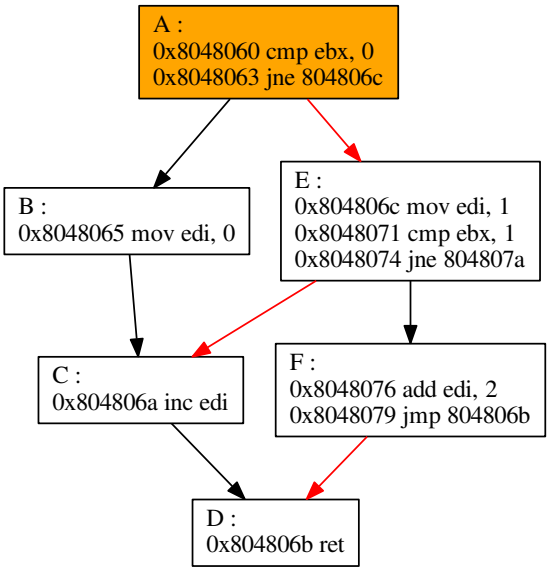
Aplatissement de graphe de flot de contrôle. Nous avons vu que le but principal de l'analyse est la construction d'un graphe de flot de contrôle cohérent du binaire. Celui-ci peut être à la base d'une analyse automatique et, s'il est suffisamment lisible, permet à un analyste humain de gagner du temps dans la compréhension du programme. Le développeur de programmes malveillants peut transformer le GFC afin de le rendre plus difficile à interprêter.

Une de ces techniques est l'aplatissement du GFC [Wan+00]. Elle consiste à transformer les changements de flot de contrôle, par exemple les appels à différentes fonctions, en un seul bloc déterminant l'adresse du saut et y effectuant un saut dynamique à l'instar d'un bloc d'aiguillage (ou *switch*).

Prenons le programme x86 donné en figure 2.3(a), dont le graphe de flot de contrôle est donné à la figure 2.3(b). Ce programme modifie la valeur de `edi` selon la valeur d'origine de `ebx`. Son GFC est simple et l'on peut par exemple déterminer facilement les valeurs possibles de `edi` en sortie (lorsque que l'instruction `ret` est atteinte) : 1, 2 ou 3.

Adresse	Octets	Instruction
8048060 <A>	83 fb 00	cmp ebx,0x0
8048063	75 07	jne 804806c <E>
8048065 <B>	bf 00 00 00 00	mov edi,0x0
804806a <C>	47	inc edi
804806b <D>	c3	ret
804806c <E>	bf 01 00 00 00	mov edi,0x1
8048071	83 fb 01	cmp ebx,0x1
8048074	74 00	jne 804806a <C>
8048076 <F>	83 c7 02	add edi,0x2
8048079	eb f0	jmp 804806b <D>

(a) Code assembleur



(b) Graphe de flot de contrôle

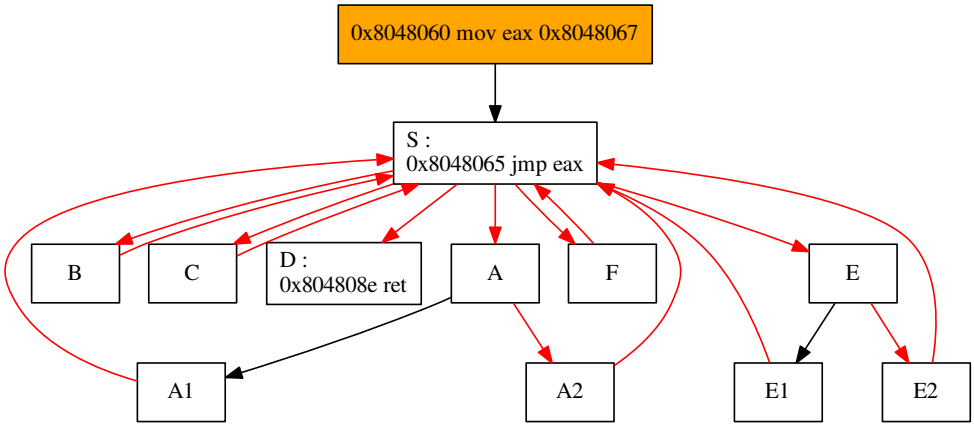
FIGURE 2.3 – Programme d'origine

Il est équivalent au programme aplati donné en figure 2.4(a) dont le GFC est donné en figure 2.4(b). Ce programme modifié est articulé autour de l'instruction de saut dynamique `jmp eax`. On a remplacé chaque saut par l'assignation de l'adresse du saut à `eax` suivi d'un appel à l'instruction de saut dynamique.

Le graphe de flot de contrôle du programme obscurci ne peut pas être exploité en l'état pour connaître les valeurs possibles de `edi` en sortie : l'enchaînement des blocs C, puis S, puis C n'est pas possible dans une exécution du programme mais l'observation du graphe de flot de contrôle de la figure 2.4(b) ne permet pas de le conclure sans regarder en détail les instructions du programme.

Adresse	Octets	Instruction
8048060	b8 67 80 04 08	mov eax,0x8048067
8048065 <S>	ff e0	jmp eax
8048067 <A>	83 fb 00	cmp ebx,0x0
804806a	75 07	jne 8048073 <A2>
804806c <A1>	b8 7a 80 04 08	mov eax,0x804807a
8048071	eb f2	jmp 8048065 <S>
8048073 <A2>	b8 8f 80 04 08	mov eax,0x804808f
8048078	eb eb	jmp 8048065 <S>
804807a <B>	bf 00 00 00 00	mov edi,0x0
804807f	b8 86 80 04 08	mov eax,0x8048086
8048084	eb df	jmp 8048065 <S>
8048086 <C>	47	inc edi
8048087	b8 8e 80 04 08	mov eax,0x804808e
804808c	eb d7	jmp 8048065 <S>
804808e <D>	c3	ret
804808f <E>	bf 01 00 00 00	mov edi,0x1
8048094	83 fb 01	cmp ebx,0x1
8048097	75 07	jne 80480a0 <E2>
8048099 <E1>	b8 a7 80 04 08	mov eax,0x80480a7
804809e	eb c5	jmp 8048065 <S>
80480a0 <E2>	b8 86 80 04 08	mov eax,0x8048086
80480a5	eb be	jmp 8048065 <S>
80480a7 <F>	83 c7 02	add edi,0x2
80480aa	b8 8e 80 04 08	mov eax,0x804808e
80480af	eb b4	jmp 8048065 <S>

(a) Code assembleur



(b) Graphe de flot de contrôle

FIGURE 2.4 – Programme aplati

2.3 Chevauchement de code

La taille d'une instruction assembleur x86 varie de 1 à 15 octets [Inta]. Il est parfaitement possible que la cible d'un saut soit une adresse se trouvant au milieu d'une autre instruction. Ainsi on parle de chevauchement de code lorsque deux instructions (ou plus) à des adresses différentes sont codées sur des adresses communes. Si une instruction à l'adresse `0xa` de taille $k \geq 2$ est atteignable, il peut y avoir une autre instruction valide et atteignable à l'adresse `0xa+1` et ces deux instructions se chevauchent.

Il est à noter que, comme indiqué par Sikorski et Honig [SH12], il n'y a dans ce cas aucun désassemblage sous forme de liste d'instructions assembleur qui soit correct et puisse être assemblé en la séquence d'octets originale. En effet une telle liste devra choisir entre l'instruction à l'adresse `0xa` et celle à l'adresse `0xa+1` alors qu'elles sont toutes les deux valides et atteignables. Une solution pour écrire un tel code assembleur est de mettre la première instruction en temps qu'instruction classique tandis que la deuxième sera présente sous forme d'octets codés en dur dans le fichier assembleur.

Exemples de chevauchement

Dans tElock. Le code de la figure 2.5 est extrait d'un programme protégé par tElock et désassemblé à l'aide d'un parcours récursif à partir de l'adresse `0x01006e7a`. Il y a une instruction `jmp +1` à l'adresse `0x01006e7d` et codée sur les deux octets `eb ff`, qui saute vers l'adresse `0x01006e7d+1` où est présente l'instruction `dec ecx`, codée sur `ff c9` et qui partage donc l'octet `ff` à l'adresse `0x01006e7d+1` avec l'instruction `jmp`.

Octets à désassembler : `fe 04 0b eb ff c9 7f e6 8b c1`

```
01006e7a    fe 04 0b        inc byte [ebx+ecx]
01006e7d    eb ff           jmp +1
01006e7e       ff c9        dec ecx
01006e80    7f e6           jg 01006e68
01006e82    8b c1           mov eax, ecx
```

FIGURE 2.5 – Désassemblage récursif de tElock

Le code assembleur permettant d'être assemblé en ces octets est donné figure 2.6 : la première instruction `jmp` peut être présente dans le code tandis que l'instruction `dec` la chevauchant est codée en dur grâce à l'octet `c9`.

```
inc byte [ebx+ecx]
jmp +1
db c9              ; l'ajout de l'octet c9 complète l'instruction dec ecx
jg 01006e68
mov eax, ecx
```

FIGURE 2.6 – Code assembleur du chevauchement de tElock

Le graphe de flot de contrôle correct pour ce code est donné sur la figure 2.7. Le sommet orange est la première instruction et les lignes en pointillés reliant deux sommets marquent un chevauchement entre les instructions de ces sommets.

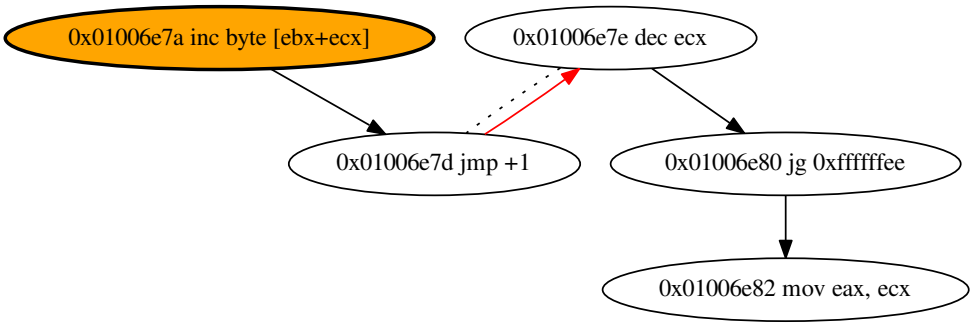


FIGURE 2.7 – Graphe de flot de contrôle de tElock

Dans UPX. UPX [UPX] cherche à optimiser la taille du binaire protégé. La procédure d'exécution du binaire protégé utilise un saut conditionnel pour séparer le contrôle de flot en deux blocs se chevauchant et finissant sur un bloc où ils se réalignent.

Octets à désassembler : `89 f9 79 07 0f b7 07 47 50 47 b9 57 48 f2 ae 55`

```
010059f0                89 f9              mov ecx, edi
010059f2                79 07              jns +9 <suite>
010059f4                0f b7 07           movzx eax, word [edi]
010059f7                47                 inc edi
010059f8                50                 push eax
010059f9                47                 inc edi
010059fa                b9 57 48 f2 ae     mov ecx, aef24857
  010059fb <suite>         57              push edi
  010059fc                    48           dec eax
  010059fd                       f2 ae     repne scasb
010059ff                55                 push ebp
```

FIGURE 2.8 – Chevauchement de code dans UPX

Le graphe de flot de contrôle pour ce chevauchement est donné en figure 2.9.

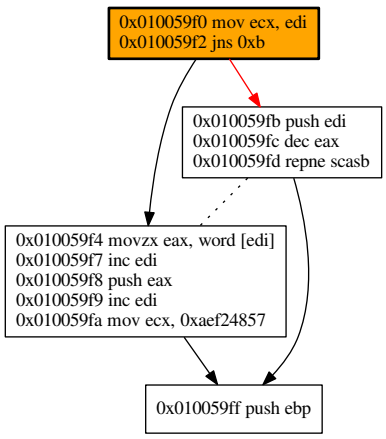


FIGURE 2.9 – Graphe de flot de contrôle de l'échantillon d'UPX

Cacher une séquence de code de taille arbitraire. Jämthagen, Lantz et Hell [JLH13] ont proposé un moyen d'inclure une longue séquence de code cachée au sein d'une séquence d'assembleur valide qui s'exécute sans provoquer d'erreur.

Une des possibilités est d'utiliser une instruction `nop` avec préfixe qui se code sur neuf octets. La partie fixe (préfixe et opérande) est codée sur quatre octets (`66 0f 1f 84`) tandis que les cinq autres octets peuvent être choisis arbitrairement. Ainsi chaque instruction `nop` peut véhiculer 5 octets de code caché. Afin d'encoder une séquence de code dans une séquence de `nop` il faut pouvoir utiliser les 4 octets fixes de l'instruction `nop` suivante de manière utile ou non perturbante pour la séquence de code caché. Une possibilité, si le registre `edx` n'est pas utilisé dans la séquence à cacher, est de prendre `ba` pour dernier octet de chaque instruction `nop` afin de former l'instruction `mov edx, 0x841f0f66` codée sur `ba 66 0f 1f 84`. On peut alors encoder n'importe quelles instructions de taille inférieure ou égale à quatre octets dans cette séquence de `nop`. Prenons la séquence d'octets suivante.

$$\texttt{66 0f 1f 84}\ \texttt{x}_1\ \texttt{x}_2\ \texttt{x}_3\ \texttt{x}_4\ \texttt{ba 66 0f 1f 84}\ \texttt{y}_1\ \texttt{y}_2\ \texttt{y}_3\ \texttt{y}_4\ \texttt{ba 66 0f 1f 84}\ \texttt{z}_1\ \texttt{z}_2\ \texttt{z}_3\ \texttt{z}_4.$$

Si on la désassemble linéairement à partir du premier octet il s'agit d'une séquence de trois `nop` tandis que si on la désassemble à partir de $\texttt{x}_1$, les instructions encodées dans les octets $\texttt{x}_\texttt{i}$, $\texttt{y}_\texttt{i}$ et $\texttt{z}_\texttt{i}$ rentrent en considération :

À partir du début		À partir de x_1	
66 0f 1f 84 $\texttt{x}_1$ $\texttt{x}_2$ $\texttt{x}_3$ $\texttt{x}_4$ ba	nop	$\texttt{x}_1$ $\texttt{x}_2$ $\texttt{x}_3$ $\texttt{x}_4$	?
66 0f 1f 84 $\texttt{y}_1$ $\texttt{y}_2$ $\texttt{y}_3$ $\texttt{y}_4$ ba	nop	ba 66 0f 1f 84	mov edx, 0x841f0f66
66 0f 1f 84 $\texttt{z}_1$ $\texttt{z}_2$ $\texttt{z}_3$ $\texttt{z}_4$ ba	nop	$\texttt{y}_1$ $\texttt{y}_2$ $\texttt{y}_3$ $\texttt{y}_4$	?
		ba 66 0f 1f 84	mov edx, 0x841f0f66
		$\texttt{z}_1$ $\texttt{z}_2$ $\texttt{z}_3$ $\texttt{z}_4$	?

Les octets forment en fait deux chemins. L'un est exclusivement composé d'instructions `nop` et est inoffensif : il s'agit du chemin d'exécution principal. Le second, le chemin d'exécution caché, débute à x_1 et exécute le code que l'on souhaite dissimuler. En pratique le premier octet du chemin principal sera une cible valide et évidente du flot de contrôle tandis qu'il sera possible de sauter indirectement sur x_1, par exemple avec un `jmp eax`. De cette manière le désassembleur est poussé à considérer le chemin principal sans examiner le chemin caché.

La limite est que les instructions cachées ne peuvent excéder une taille de 4 octets. Les auteurs expliquent cependant que beaucoup d'instructions x86 peuvent être séparées en plusieurs instructions plus petites, en utilisant par exemple des registres restreints comme `ax` ou `al` à la place d'`eax`.

2.4 Auto-modification

Il a été expliqué dans la section 1.1 que, avec l'architecture de Von Neumann, le code n'est pas physiquement séparé des données lors de l'exécution sur une machine réelle. Un programme auto-modifiant est simplement un programme utilisant cette propriété pour modifier le code assembleur le définissant au cours même de son exécution. Ainsi on parle de comportement auto-modifiant lorsqu'une instruction du programme est codée sur au moins un octet qui a au préalable été modifié par ce programme.

En pratique les processeurs récents implémentent une protection, appelée bit NX ou W^X (prononcé "W xor X"), permettant d'empêcher qu'une page mémoire puisse être à la fois écrite et exécutée lors de l'exécution du programme. Cette protection a été ajoutée pour éviter des attaques par injection de code. Ce type d'attaques résulte en l'exécution de code au sein de données entrées par l'utilisateur du programme. Le choix d'activer ou non la protection revient au système d'exploitation mais en général ceux-ci l'autorisent par défaut parce que l'auto-modification a des cas d'utilisation légitimes. De ce fait l'activation ou non de la protection est spécifiée lors de la compilation et si un programme n'est pas protégé il lui suffit d'utiliser un appel système (`mprotect` sous Linux) pour autoriser l'exécution de code dans les sections de données ou l'écriture dans les sections de code.

Prenons le programme de la figure 2.10. Ce programme commence par autoriser l'accès en écriture à la section de code `.text` : les droits d'accès de cette section deviennent RWX (lecture, écriture et exécution sont autorisées). Puis il place une adresse mémoire, `0x8048076` dans `eax`, puis l'instruction 2 écrit à l'adresse $eax + 1$, provoquant la modification de l'instruction 4 en `mov edi, 2`, codée sur `bf 02 00 00 00`. Puis l'instruction 3 provoque une seconde auto-modification en transformant l'instruction 5 en `mov ebx, 2`, codée sur `bb 02 00 00 00`. Si on ne prend pas en compte les instructions auto-modifiantes la valeur finale, affichée, de `edi` et `ebx` est 1. Pourtant les instructions 2 et 3 modifient le code de telle sorte que la valeur finale de ces deux registres soit 2 au moment de leur affichage.

On constate ici que le programme se modifie au cours de son exécution et donc on ne peut pas se contenter de la représentation d'origine du programme pour l'analyser. De plus les instructions modifiées pourraient modifier le flot de contrôle du programme en rajoutant des instructions de saut conditionnel par exemple, donc le graphe de flot de

i	Adresse	Octets	Instruction
0	(...)	(...)	`.text` -> RWX
1	8048060	b8 6b 80 04 08	mov eax, 0x8048076
2	8048065	66 c7 40 01 02 00	mov [eax+1], 2
3	804806b	66 c7 40 06 02 00	mov [eax+6], 2
4	8048076	bf 01 00 00 00	mov edi, 1
5	804807b	bb 01 00 00 00	mov ebx, 1
6	(...)	(...)	Affiche edi et ebx
7	(...)	(...)	Quitte

FIGURE 2.10 – Programme auto-modifiant

contrôle initial peut être amené à évoluer au cours de l'exécution du programme.

2.5 Logiciels malveillants et obscurcissement

Un programmeur cherchant à protéger son binaire va se tourner vers les techniques d'obscurcissement dont nous avons donné quelques exemples ci-dessus. Puisque chacune des transformations a pour effet de rendre le programme plus difficile à comprendre, il n'y a souvent pas de limite au nombre de fois qu'il est possible d'itérer une technique de protection particulière. On peut alors chercher un compromis entre niveau de protection et complexité ajoutée au programme (en temps, en taille et en mémoire utilisée).

En pratique les auteurs des logiciels malveillants utilisent des logiciels existants pour protéger leurs binaires. Ces logiciels de protection s'appliquent à l'ensemble du programme malveillant sans avoir à définir des atouts particuliers à protéger. On appelle ces programmes des empaqueteurs (*packers*) : un empaqueteur prend un binaire en entrée et produit en sortie un binaire empaqueté, protégé. Une technique couramment utilisée est de cacher le binaire d'origine dans les données du binaire empaqueté, par exemple chiffré ou compressé (voir figure 2.11). Lors de l'exécution du binaire protégé une première phase consiste à restaurer le binaire d'origine en mémoire puis à effectuer un saut vers le point d'entrée du code restauré, provoquant ainsi l'exécution du binaire d'origine.

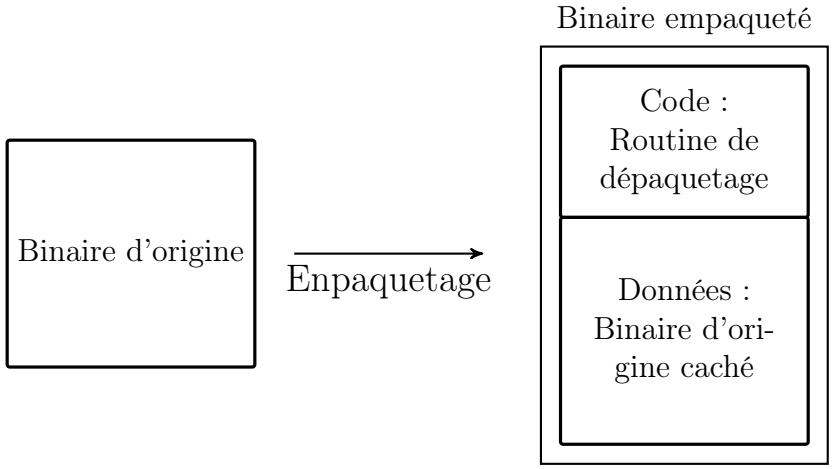


FIGURE 2.11 – Empaquetage d'un binaire

L'empaqueteur libre UPX [UPX] est un cas typique du schéma précédent. Il compresse le binaire d'origine et le restaure à l'exécution. Ainsi il est plutôt conçu pour optimiser la taille du binaire paqueté et non pour l'obscurcir. La transformation utilisée étant réversible, l'empaqueteur permet également de récupérer le programme binaire d'origine sans l'exécuter. Il est cela dit parfois utilisé par des auteurs de logiciels malveillants sous une forme modifiée ne permettant plus de le dépaqueter automatiquement.

2.6 Conclusion

Cette thèse s'intéresse particulièrement à l'analyse des programmes écrits en assembleur x86 et x86_64. Ces programmes ont en général été compilés à partir d'un langage de

haut niveau puis ont été modifiés à l'aide d'un logiciel de protection. Les binaires que l'on étudie sont donc protégés avec des techniques statiques et l'emploi de l'auto-modification. Notre travail consiste alors à chercher à désassembler correctement ces programmes dans le but de faciliter leur analyse.

Les chapitres suivants détailleront plusieurs techniques d'analyse que nous avons appliquées. Dans un premier temps nous nous intéresserons à l'aspect auto-modifiant des programmes et verrons comment l'analyse dynamique peut être utilisée pour reconstruire un modèle pour le programme auto-modifiant. Nous introduirons ensuite des techniques d'analyse statiques pour chercher à construire un graphe de flot de contrôle approximant le graphe de flot de contrôle parfait et contourner certaines méthodes de protection comme le chevauchement de code.

Chapitre 3

Sémantique de l'assembleur et langage intermédiaire

Dans ce chapitre nous proposons un modèle de langage simplifié pour l'assembleur et une sémantique concrète pour ce langage.

Lors de l'analyse d'un binaire, il est crucial de pouvoir analyser chaque instruction. La question centrale est la suivante : quelle est l'opération réalisée par cette instruction ? En particulier quels sont ses opérandes d'entrée, de sortie, et comment sont-ils lus et modifiés ? Prenons l'instruction `push ecx`. Il est clair que le registre `ecx` est un opérande mais cette information est insuffisante : cette instruction consiste à empiler la valeur de `ecx` sur la pile, elle modifie donc le pointeur de pile `esp` et écrit à une adresse mémoire.

Nous reprenons la classification informelle en trois niveaux d'information proposée par Calvet [Cal13] à la figure 3.1, par exemple pour l'instruction `push ecx` :

— Niveau 1 - Opérandes explicites : `ecx`
— Niveau 2 - Opérandes implicites précis : `ecx` et `esp` sont des entrées, l'adresse mémoire à l'adresse $esp - 4$ est une sortie
— Niveau 3 - Opération : l'instruction décrémente `esp` de 4 puis écrit à l'adresse mémoire pointée par `esp` la valeur qui est dans `ecx`

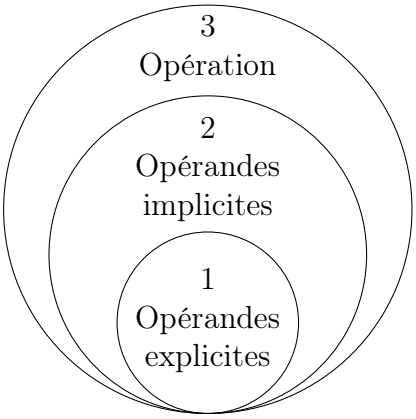


FIGURE 3.1 – Niveaux de précision des informations sur une instruction

Une sémantique concrète se place au niveau 3 : elle donne une information exhaustive sur l'opération réalisée par chaque instruction. Ce chapitre sera donc consacré à la définition d'une sémantique concrète pour un langage assembleur simplifié. Nous verrons par la suite que pour certaines analyses une sémantique de niveau 2 peut-être suffisante.

3.1 Sémantique concrète pour un langage assembleur

Représentation de la mémoire et des registres. On peut voir la mémoire comme un tableau indexé sur les entiers contenant la pile et le tas. Les registres sont des variables distinctes de la mémoire et sont en nombre limité. L'assembleur distingue plusieurs modes d'adressage dont les adressage direct et indirect : à l'exécution `eax` prend la valeur du registre `eax` (adressage direct) tandis que `[eax]` fait référence à la valeur en mémoire à l'adresse contenue dans `eax` (adressage indirect). Ces éléments sont définis formellement à la définition 3.1 : $\mathbb{A}$ est l'ensemble des registres et $\mathbb{T}$ représente la mémoire comme un tableau.

Définition 3.1. *On définit un ensemble fini de registres $a \in \mathbb{A}$ et un tableau de taille finie $\mathbb{T} = \{T_n,\ n \in [0,\ N]\}$. Les variables $v \in \mathbb{V} = \mathbb{A} \cup \mathbb{T}$ sont soit un registre soit dans le tableau :*
$var := a \in \mathbb{A} \mid T_n$ avec $n \in \mathbb{N}$.
Les expressions $\mathbb{E}$ sont de l'un des types suivants :
$expr := v \in \mathbb{V} \mid [v] \in \mathbb{V} \mid \perp \mid n \in \mathbb{N}$.

Langage assembleur simplifié. Nous allons définir un langage intermédiaire dans lequel on peut transcrire chaque instruction x86 en une liste d'instructions atomiques de notre langage simplifié. Les instructions x86 sont disponibles à différentes adresses entières. Les instructions atomiques sont de plusieurs types explicités en définition 3.2 : le premier consiste en l'assignation. Il est possible d'assigner la valeur d'une expression à une variable représentée par une expression. Le second type d'instruction regroupe les sauts inconditionnels et conditionnels. On distingue également l'instruction `end` forçant l'arrêt du programme.

Définition 3.2. *Les instructions atomiques d'un programme sont de ce type, si l'on dispose d'un ensemble d'opérateurs g de $\mathbb{N}^m$ dans $\mathbb{N}$ avec $x_1, ..., x_m$, x et x' étant des expressions.*
$inst := x \leftarrow g(x_1, ..., x_m) \mid$ goto $x \mid$ if x then goto $x' \mid$ end

La figure 3.2 donne des exemples de transcription d'instructions x86 dans le langage défini précédemment. Ils sont naïfs au sens que les instructions x86 sont plus complexes et un simple `sub` provoque des effets de bord modifiant des registres. Ce point sera développé par la suite lorsque nous discuterons des implémentations possibles pour cette sémantique concrète.

Comme nous l'avons vu dans les chapitres précédents, un programme est simplement composé d'un ou plusieurs blocs d'octets à charger en mémoire. Une fois ces segments

Instruction x86	Instructions atomiques équivalentes
mov eax, 3	$eax \leftarrow 3$
mov [eax], 4	$[eax] \leftarrow 4$
mov [eax+1], 5	$tmp \leftarrow addition(eax, 1)$ $[tmp] \leftarrow 5$
jmp eax	$goto\ eax$
sub eax, 3	$eax \leftarrow soustraction(eax, 3)$

FIGURE 3.2 – Exemple de transcription d'instructions x86 dans le langage assembleur simplifié

chargés dans le tableau $\mathbb{T}$ représentant la mémoire, le point d'entrée du programme est placé dans le registre `ep`.

Pour faire le lien entre la machine et les instructions exécutées, il est nécessaire de pouvoir désassembler des instructions en mémoire. C'est le rôle de l'opérateur de désassemblage (définition 3.3).

Définition 3.3. *On appelle D l'opérateur de désassemblage qui, à une adresse de la mémoire $\mathbb{T}$ associe une liste d'instructions atomiques et la taille de cette instruction dans $\mathbb{N}$. Pour toute adresse $t \in \mathbb{T}$, on note $D(t) = d_1..d_n$ et $D_S(t)$ respectivement la suite d'instructions atomiques à l'adresse t et la taille de cette instruction. Dans le cas où il n'y a pas d'instruction valide à l'adresse t, $D(t) = \perp$ et $D_S(t) = 0$.*

Une sémantique concrète cherche à définir les opérations de chaque instruction de la manière la plus précise qui soit afin de pouvoir simuler une exécution réelle du programme. Pour cela on va utiliser un store qui conserve l'état des variables lors de l'exécution du programme. Toute variable non initialisée a la valeur spéciale $\perp$ tandis que les variables définies ont des valeurs entières (définition 3.4).

Définition 3.4. *Un store dynamique Θ associe à chaque variable une valeur, $\Theta : \mathbb{V} \rightarrow \mathbb{N} \cup \{\perp\}$. Si v est une variable et n une valeur dans $\mathbb{N} \cup \{\perp\}$, on note $\Theta[v \leftarrow n]$ l'assignation de v à n dans Θ.*

Pour permettre l'adressage indirect on doit également définir le store sur l'ensemble des expressions de la forme $[v]$ et de toutes les expressions dans le cas général. La définition 3.5 étend la notion de store à l'adressage indirect afin que chaque symbole ait une valeur.

Définition 3.5. *Définissons une extension Θ_X d'un store aux expressions, c'est à dire $\Theta_X : \mathbb{E} \rightarrow \mathbb{N} \cup \{\perp\}$. Soit $x \in \mathbb{E}$.*

- *Si $x \in \mathbb{V}$: $\Theta_X(x) = \Theta(x)$*
- *Si $x = \perp$, $\Theta_X(x) = \perp$*
- *Si $x \in \mathbb{N}$, $\Theta_X(x) = x$*
- *Sinon, $x = [v]$ avec $v \in \mathbb{V}$.*
  - *Si $\Theta(v) = \perp$ alors $\Theta_X(x) = \perp$*
  - *Si $\Theta(v) \in \mathbb{N}$ et $\Theta(v) \notin [0,\ N]$ alors $\Theta_X(x) = \perp$*
  - *Sinon $\Theta(v) \in \mathbb{N}$ et $\Theta(v) \in [0,\ N]$, alors $\Theta_X(x) = \Theta(\Theta(v))$.*

De même on étend l'opération d'assignation d'une valeur à une expression de la manière suivante. Soit $x \in \mathbb{E}$ et $n \in \mathbb{N} \cup \{\bot\}$.

- *Si $x = \bot$, $\Theta_X[x \leftarrow n]$ n'a aucun effet*
- *Si $x \in \mathbb{N}$, $\Theta_X[x \leftarrow n]$ n'a aucun effet*
- *Si $x \in \mathbb{V}$: $\Theta_X[x \leftarrow n]$ est équivalent à $\Theta[x \leftarrow n]$.*
- *Sinon, $x = [v]$ avec $v \in \mathbb{V}$.*
  - *Si $\Theta(v) = \bot$ alors l'assignation n'a aucun effet*
  - *Si $\Theta(v) \in \mathbb{N}$ et $\Theta(v) \notin [0,\ N]$ alors l'assignation n'a aucun effet*
  - *Sinon $\Theta(v) \in \mathbb{N}$ et $\Theta(v) \in [0,\ N]$, alors $\Theta_X[x \leftarrow n]$ se résout par $\Theta[T_{\Theta(v)} \leftarrow n]$.*

Règles de transition. Les états d'exécution sont de la forme $\langle t, \Theta_X \rangle$ où t est une adresse ou l'adresse invalide (ou finale) $\bot$. À chaque instruction exécutée, il y a une transition $\langle t, \Theta_X \rangle \rightarrow \langle t', \Theta'_X \rangle$.
S'il y a une suite de transitions amenant d'un état $\langle t, \Theta_X \rangle$ à l'état $\langle t', \Theta'_X \rangle$, on note $\langle t, \Theta \rangle \rightarrow^* \langle t', \Theta'_X \rangle$.
Un programme s'arrête en partant de l'état initial $\langle ep, \Theta_X \rangle$ où `ep` est son point d'entrée si et seulement si $\langle ep, \Theta_X \rangle \rightarrow^* \langle \bot, \Theta'_X \rangle$. Le programme dans ce cas s'arrête à la première occurrence d'une adresse finale ou invalide $\bot$.

On définit par la suite la sémantique des instructions atomiques et on étend donc les états d'exécution à celles-ci : on note $\langle t : d_1..d_n, \Theta_X \rangle$ l'état $\langle t, \Theta_X \rangle$ sur lequel il y a les instructions atomiques $d_1..d_n$ à évaluer. À partir d'un état $\langle t, \Theta_X \rangle$, on commence par déterminer les instructions atomiques à évaluer à l'aide de l'opérateur de désassemblage : on arrive à un état $\langle t : D(t), \Theta_X \rangle = \langle t : d_1..d_n, \Theta_X \rangle$ dans lequel on va pouvoir évaluer les instructions atomiques l'une après l'autre. Une fois que toutes ces instructions ont été évaluées et si aucune d'elle n'a provoqué de saut vers une adresse différente, on passe à l'instruction qui suit séquentiellement à l'adresse $t + D_S(t)\rangle$.

L'état initial du store est : $\forall v \in \mathbb{V}, \Theta_X(v) = \bot$ et l'on part du point d'entrée `ep`. Les règles de transition suivantes permettent d'aboutir à une adresse finale ou invalide $\bot$ si le programme termine.

$$
\begin{array}{lcl}
\langle t \in \mathbb{T};\ \Theta_X\rangle & \longrightarrow & \langle t : D(t);\ \Theta_X\rangle \\
\langle t : x \leftarrow g(x_1, ..., x_m), d_2..d_n;\ \Theta_X\rangle & \longrightarrow & \langle t : \bot;\ \Theta_X\rangle \\
 & & \quad \text{si } \exists i, \Theta_X(x_i) = \bot \\
 & \longrightarrow & \langle t : d_2..d_n;\ \Theta_X[x \leftarrow [\![g]\!](\Theta_X(x_1), ...,\ \Theta_X(x_m))]\rangle \\
 & & \quad \text{sinon} \\
\langle t : \text{goto } x, d_2..d_n;\ \Theta_X\rangle & \longrightarrow & \langle \Theta_X(x);\ \Theta_X\rangle \\
\langle t : \text{if } x \text{ then goto } x', d_2..d_n;\ \Theta_X\rangle & \longrightarrow & \langle \Theta_X(x'),\ \Theta_X\rangle \\
 & & \quad \text{si } \Theta_X(x) = 1 \\
 & \longrightarrow & \langle t + D_S(t),\ \Theta_X\rangle \\
 & & \quad \text{sinon} \\
\langle t : \text{end}, d_2..d_n;\ \Theta_X\rangle & \longrightarrow & \langle \bot;\ \Theta_X\rangle \\
\langle t : \bot;\ \Theta_X\rangle & \longrightarrow & \langle \bot;\ \Theta_X\rangle \\
\langle t \in \mathbb{T} : \emptyset;\ \Theta_X\rangle & \longrightarrow & \langle t + D_S(t);\ \Theta_X\rangle \\
\langle t \notin \mathbb{T};\ \Theta_X\rangle & \longrightarrow & \langle \bot;\ \Theta_X\rangle
\end{array}
$$

Dans la suite nous appellerons évaluation sémantique ou `sem_eval` la fonction qui, à une adresse t et un store Θ_X, associe l'état suivant : si t' et Θ'_X sont les premières valeurs vérifiant $\langle t, \Theta\rangle \rightarrow^* \langle t', \Theta'_X\rangle$ avec $t \neq t'$ alors $\texttt{sem_eval}(t, \Theta_X) = (t', \Theta'_X)$.

3.2 Assembleur et langage intermédiaire

En pratique pour analyser un programme assembleur on veut en avoir une représentation dont on connaît la sémantique concrète. On va pour cela réécrire le programme dans un langage intermédiaire et effectuer les analyses sur le programme en langage intermédiaire. Le langage défini dans la section précédente est un langage intermédiaire possible.

Une des difficultés rencontrées dans la transformation d'un langage assembleur en langage intermédiaire est la richesse sémantique du langage x86 qui contient des centaines d'instructions et utilise de nombreux registres et drapeaux du processeur. L'instruction `sub eax, b` par exemple soustrait l'opérande `b` à `eax` en stockant le résultat de l'opérant dans `eax`. Le code d'opération `sub` désigne une vingtaine de variantes de la soustraction selon la taille des opérandes pris en compte. Elle effectue l'opération `eax-b` et met à jour les drapeaux CF et OF indiquant un dépassement de valeur entière, le drapeau de signe SF, le drapeau ZF (à 1 si le résultat est nul) et celui de parité PF.

Une instruction assembleur va donc être traduite en une ou plusieurs instructions atomiques dans le langage intermédiaire, que l'on sait équivalentes sémantiquement à l'instruction x86 et pour lesquelles on dispose d'une sémantique concrète permettant de

les exécuter. Une difficulté pratique vient de la richesse du langage x86 : écrire l'équivalence de chacune de ses instructions dans le langage intermédiaire choisi est une tâche longue et prône aux erreurs, notamment lors de l'analyse de la documentation. Pour cette raison nous avons rapidement choisi de nous orienter vers des langages intermédiaires pour lesquels cette étape a déjà été réalisée.

Une seconde difficulté provient de l'implémentation des appels systèmes. Ces instructions (`int 80` en assembleur x86 et `syscall` en assembleur x86_64) appellent des fonctions spécifiques du système d'exploitation sur lequel le programme est exécuté. La sémantique pour l'entrée comme la sortie de ces instructions n'est donc pas spécifiée dans le langage assembleur mais par le système d'exploitation cible. Ces appels sont omniprésents dans un programme réel puisque la plupart des interactions avec le système (lecture et écriture dans un fichier, connexion internet, allocation de mémoire via la fonction *malloc* en C par exemple, etc.) passent par un appel système.

3.3 Revue de littérature des langages intermédiaires

BAP. Parmi les projets de langages intermédiaires existants, fournissant d'une part une transcription depuis l'assembleur et d'autre part une sémantique concrète du langage considéré, nous nous sommes intéressés à BAP [Bru+11]. BAP est le successeur du projet BitBlaze qui a défini et développé une plateforme d'analyse statique basée sur le langage intermédiaire Vine IL [Son+08]. Sémantiquement le langage intermédiaire défini par BAP ne diffère pas fondamentalement de celui défini dans ce chapitre. Ils définissent également un opérateur permettant de gérer l'adressage indirect : un appel à `load(`e_1`, `e_2`, `e_3`, `τ_{reg}`)` place la valeur de l'expression e_1 à l'adresse e_2, ainsi `load(`x`, `eax`, `e_3`, `τ_{reg}`)` effectue l'opération $[eax] \leftarrow x$. Le paramètre e_3 a une valeur booléenne indiquant si la machine fonctionne en petit ou grand boutiste et τ_{reg} indique le nombre d'octets qui doivent être copiés. L'avantage de BAP par rapport au langage présenté précédemment est qu'il est bien plus proche du langage assembleur : les valeurs des variables ne sont pas dans $\mathbb{N}$ mais dans $[0,\ 2^{32}-1]$, $[-2^{31},\ 2^{31}-1]$, $[0,\ 2^{64}-1]$ ou $[-2^{63},\ 2^{63}-1]$ selon la machine que l'on cherche à modéliser et le type des variables (signé ou non). De même les fonctions entières sont explicitées et contiennent, entre autres, des opérations binaires comme l'addition, le "ou" logique, la comparaison de deux valeurs.

Jakstab. Le projet Jakstab [KV08] ajoute un modèle de la mémoire pouvant gérer les allocations dynamiques de mémoire effectuées par l'appel système *malloc* en incluant dans langage intermédiaire les instructions *alloc* et *free* [KV10]. La mémoire est alors scindée en trois régions : la région globale contient le code et les variables globales du programme, une seconde région contient la pile, et la dernière région inclut le tas, c'est à dire la mémoire allouée dynamiquement par des *malloc*. Cette dernière région contient autant de blocs distincts qu'il y a eu d'allocations dynamiques.

DBA. Le langage intermédiaire DBA (*Dynamic Bitvector Automata*) [Bar+11], faisant partie du projet BinSec [ANR], intègre des permissions au sein de son modèle. À l'instar du

comportement de Jakstab la mémoire est scindée en régions disjointes et chaque allocation dynamique crée une nouvelle région. Une région de mémoire peut être marquée comme accessible en lecture, en écriture, en exécution ou une combinaison de ces trois permissions. L'exécution d'une instruction provoquant un accès mémoire interdit est incluse dans la sémantique et provoque une erreur. Cette spécificité permet à DBA de mieux représenter la mémoire d'une machine réelle où le système d'exploitation gère ces droits d'accès.

REIL. Dullien et Porst [TS09] ont développé un langage intermédiaire adapté à l'analyse de la sécurité logicielle et en particulier à la recherche de vulnérabilités, REIL (*Reverse Engineering Intermediate Language*). Il est composé de 17 instructions permettant les opérations de calculs arithmétiques, de lecture et d'écriture des registres et de la mémoire, et de sauts conditionnels. Le langage intermédiaire REIL est un des composants inclus dans les logiciels d'analyse de binaires commercialisés par Zynamics [Zynb ; Zyna] : ils indiquent disposer d'une sémantique concrète utilisée pour l'analyse et d'implémentations de transformations depuis l'assembleur vers REIL mais n'ont pas publié sur ces résultats.

3.4 Langage intermédiaire et analyse de binaires

Un programme sous forme d'instructions dans un langage intermédiaire est plus facile à exploiter qu'un programme assembleur puisqu'on a une sémantique concrète clairement définie et assez compacte. Il est donc, comme nous le verrons par la suite, fréquent de construire des méthodes de désassemblage et d'analyse de binaires autour d'une représentation intermédiaire.

Approche globale. Une première approche, suivie par exemple par BAP, consiste, à l'instar de la compilation, à transformer l'intégralité du programme assembleur en un programme en langage intermédiaire et à effectuer les analyses sur cette représentation intermédiaire. Cette approche est illustrée par le graphique donné à la figure 3.3. La partie d'analyse consiste en l'exécution ou l'émulation du programme dans son langage intermédiaire en partant du point d'entrée. Une première instruction est décodée, analysée et ajoutée dans le graphe de flot de contrôle du programme. Puis cette instruction est évaluée selon la méthode d'évaluation sémantique précédemment définie. Cette évaluation a des conséquences sur l'état des registres du processeur et de la mémoire de la machine. L'instruction qui doit être exécutée ensuite est celle présente à l'adresse donnée par le registre de compteur ordinal (`eip` en assembleur x86). On récupère alors cette instruction pour l'analyser à son tour.

Le souci de cette approche est qu'elle induit une perte d'informations de bas niveau : la représentation d'origine du binaire sous forme d'octets n'est plus accessible dans la représentation intermédiaire. Pourtant on ne peut pas séparer les instructions de leur représentation en mémoire : si une instruction est codée à l'adresse `0xa` sur deux octets et que l'octet à l'adresse `0xa+1` est modifié, cette modification peut être transcrite dans le langage intermédiaire mais il est toujours nécessaire de connaître l'octet à l'adresse `0xa` pour pouvoir décoder la nouvelle instruction.

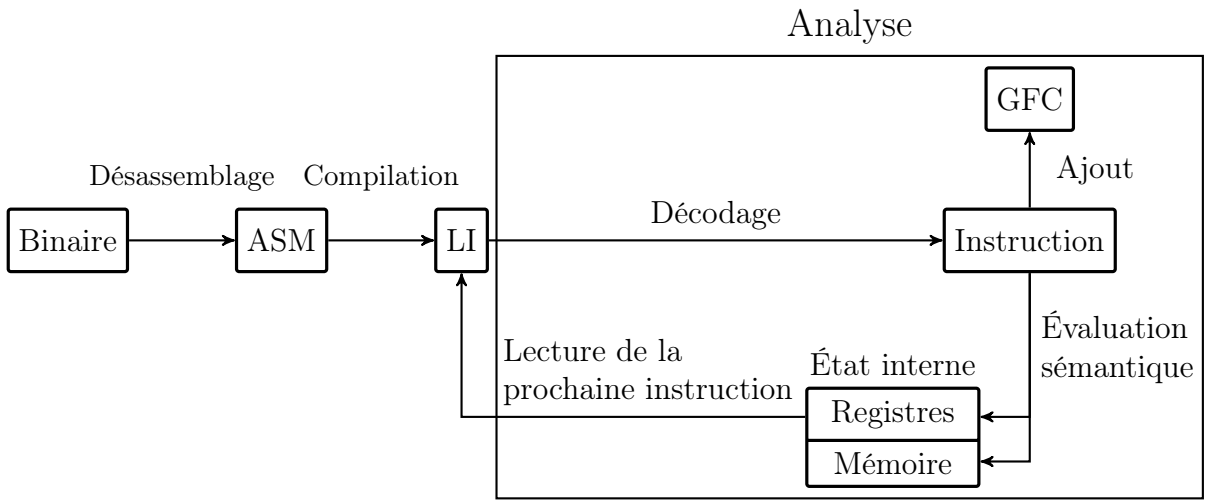


FIGURE 3.3 – Approche globale de l'analyse de binaires à l'aide d'un langage intermédiaire (LI)

La figure 3.4 donne un exemple de ce cas. Le segment de code donné est constitué de trois instructions dont l'une modifie la valeur du registre `edi`. Supposons que le but de l'analyse est de connaître la valeur de `edi` à l'issue de l'exécution.

Adresse	Octets	Instruction ASM	Transcription en LI
8048060	b8 6b 80 04 08	mov eax,0x804806b	eax ← 0x804806b
8048065	66 c7 40 01 02 00	mov [eax+1], 2	tmp ← addition(eax, 1) [tmp] ← 2
804806b	bf 01 00 00 00	mov edi, 1	edi ← 1

FIGURE 3.4 – Programme assembleur et sa transcription dans le langage intermédiaire

Les deux premières instructions servent à modifier le second octet de l'instruction modifiant `edi` de `01` à `02`, transformant l'instruction `mov edi, 1` en `mov edi, 2`. L'approche globale de la transcription du programme dans le langage intermédiaire, bien que chaque instruction assembleur soit sémantiquement équivalente aux instructions transcrites, résulte en un programme qui n'est pas auto-modifiant. Il est composé des quatre instructions de la dernière colonne du tableau. Les trois premières écrivent la valeur 2 en mémoire à l'adresse `0x804806c` et la dernière, indépendante, attribue la valeur 1 au registre `edi`. Ainsi l'analyse conclurait, à tort, que l'exécution du segment de code assembleur d'origine attribue 1 à `edi`, au lieu de 2.

Ainsi, pour pouvoir utiliser ce langage intermédiaire dans le cas d'un programme automodifiant, il est nécessaire d'avoir accès à une représentation en mémoire du binaire exécuté lors de son exécution sous forme intermédiaire.

Approche atomique. En pratique nous souhaitons donc avoir accès à la représentation sous forme binaire du programme intial. Nous allons donc commencer par le charger

en mémoire comme une partie intégrante de l'état interne de la machine. C'est le comportement attendu d'une machine de Von Neumann et le comportement du programme peut donc être auto-modifiant. Le décodage, le désassemblage puis la transformation en langage intermédiaire et l'évaluation sémantique des instructions du programme se feront donc au fur et à mesure de l'analyse, une instruction à la fois, depuis l'état de la mémoire. Ce modèle est illustré par le diagramme de la figure 3.5.

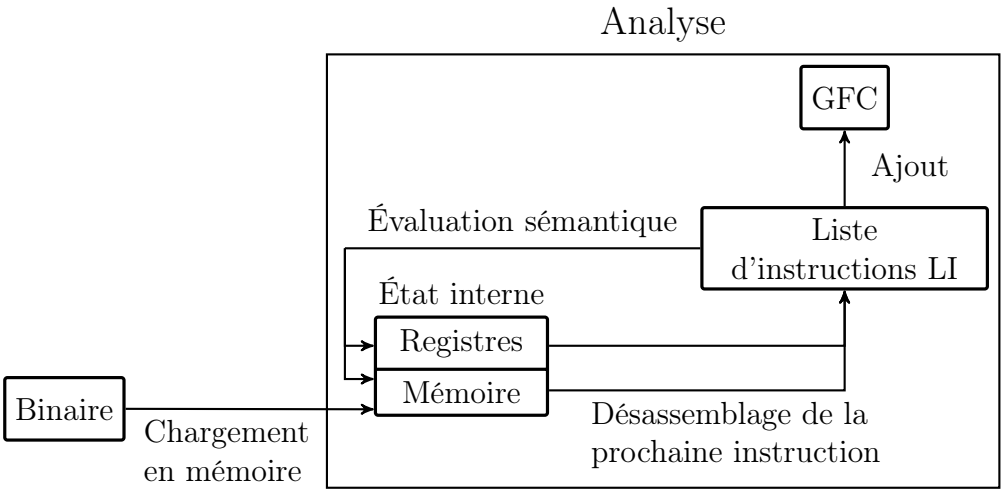


FIGURE 3.5 – Approche atomique de l'analyse de binaires à l'aide d'un langage intermédiaire

Au cœur de cette architecture se trouvent deux opérateurs. Le premier est l'opérateur de désassemblage capable de transformer des octets présents à une certaine adresse mémoire en une liste d'instructions dans le langage intermédiaire, équivalentes sémantiquement à l'instruction assembleur codée sur ces octets. Le second est l'opérateur d'évaluation sémantique qui modifie l'état interne de la machine selon l'instruction dans le langage intermédiaire donnée en entrée.

3.5 Implémentations

L'objectif principal de l'implémentation est d'obtenir un couple d'opérateurs de désassemblage et d'évaluation sémantique permettant l'analyse de programmes auto-modifiants. Ces implémentations fonctionneront donc selon l'approche atomique définie à la partie précédente. Les détails de la sémantique du langage intermédiaire utilisé n'ont pas d'importance vu qu'il n'est jamais utilisé en dehors de ces deux opérateurs et que seule la sortie de l'évaluation sémantique est observée.

3.5.1 Preuve de concept en C

Nous avons réalisé une implémentation en C de la sémantique concrète de notre langage. Le langage intermédiaire est calqué sur celui défini dans ce chapitre : une instruction atomique est d'un des types suivants.

— ASSIGN : Assignation
— GOTO : Saut inconditionnel
— IFTHENGOTO : Saut conditionnel
— NOP : cette instruction n'a pas d'effet
— END : indique la fin du programme
— UNKNOWN : l'instruction n'a pas été reconnue

Une variable est d'un des types suivants :

— CONST : une constante entière
— MEM : une adresse de la mémoire
— REG : un registre
— FLAG : un drapeau du processeur

Un store associe à une variable une valeur qui est soit un entier (de type *int*, à valeurs dans $[-2^{31},\ 2^{31}-1]$), soit $\perp$ si la variable n'a pas été initialisée.

Pour les instructions d'assignation de la forme $x \leftarrow g(x_1, ..., x_m)$, nous avons implémenté quelques fonctions totales comme la somme de deux variables, l'opposé d'une variable, la fonction renvoyant 1 si la variable est positive ou nulle, 1 sinon. Cette dernière fonction, avec celles permettant de déterminer si une variable est nulle ou si une variable est paire, est cruciale pour transcoder l'instruction assembleur `cmp` permettant de comparer deux variables avant d'effectuer un saut selon le résultat de la comparaison.

Désassemblage. Nous avons utilisé la librairie XED d'Intel [Intb] qui permet d'encoder et de décoder des instructions assembleur sur demande. À partir d'une séquence d'octets elle fournit une structure contenant, entre autres, les informations correspondant au niveau 2 de sémantique précédemment défini :

— Une chaîne de caractères décrivant l'instruction
— Le type de l'instruction (`CMP`, `MOV`, `JMP` par exemple)
— Les arguments et leurs types (adresse mémoire, registre ou drapeau)

Ces informations ne suffisent pas en l'état pour construire une suite d'instructions dans le langage intermédiaire équivalentes sémantiquement à l'instruction assembleur décodée. Les informations fournies permettent par contre de classer les instructions en différentes catégories, selon la documentation officielle, et d'écrire des règles de transcription adaptées à chacune afin d'en déduire un désassemblage. Cette étape de transcription des instructions assembleur est la plus pénible à implémenter puisqu'elle consiste principalement à lire la documentation des processeurs ciblés (ici les processeurs Intel [Inta]). Prenons les instructions de type `CMP` prenant comme arguments deux variables à comparer, par exemple `cmp eax, ebx`. Cette instruction met à jour les drapeaux du processeur selon des caractéristiques de $eax - ebx$. Elle est sémantiquement équivalente à la suite d'instructions suivantes.

tmp ← opposé(ebx)
tmp ← somme(tmp, eax)
PF ← parité(tmp)
ZF ← nul(tmp)
SF ← signe(tmp)
OF ← débordement(tmp)
CF ← retenue(tmp)

Évaluation sémantique. La seule instruction atomique provoquant la modification de la valeur d'une variable, hors compteur ordinal, est l'assignation, de la forme $x \leftarrow g(x_1, ..., x_m)$. L'évaluation concrète d'une assignation est donnée dans la sémantique du langage intermédiaire et consiste simplement à évaluer les arguments, puis à calculer la valeur de $g(x_1, ..., x_m)$ et à l'assigner à la variable x.

Le second aspect de l'évaluation sémantique est la détermination de l'adresse de la prochaine instruction à évaluer, c'est à dire de la valeur de `eip`. Cette valeur est également donnée par la sémantique du langage intermédiaire.

Limites de l'approche. Nous avons implémenté la transcription de quelques instructions assembleur en langage intermédiaire mais le temps requis pour implémenter toutes les variantes de chaque instruction est suffisamment long pour qu'on se tourne vers des plateformes d'analyse de binaires où cette étape a déjà été réalisée.

3.5.2 Émulation de code auto-modifiant avec BAP

BAP permet de transcrire un programme en assembleur en un programme écrit dans le langage intermédiaire utilisé par BAP, en utilisant le modèle global décrit précédemment. Il fournit également un émulateur permettant d'exécuter un programme en langage intermédiaire. Seul l'assembleur x86 est supporté par la version 0.7 de BAP que nous avons utilisée et l'émulateur ne permet pas l'auto-modification.

Nous souhaitions utiliser BAP pour qu'il nous fournisse l'opérateur de désassemblage et d'évaluation sémantique et ainsi permettre l'analyse de code auto-modifiant. Ces deux opérateurs sont déjà implémentés dans BAP mais ne sont pas normalement accessibles depuis l'interface de programmation. Nous avons donc légèrement modifié le code de BAP, codé en OCaml, pour les rendre accessibles et permettre le décodage d'une instruction puis son exécution dans un contexte donné.

Limites de l'approche. Nous sommes capables d'émuler l'exécution de programmes auto-modifiants simples mais les appels systèmes ne sont pas gérés par BAP. La sémantique d'un appel système dépendant largement du système d'exploitation sur lequel le programme est exécution, il s'agit d'une limite naturelle de toute méthode basée sur l'émulation. Notre implémentation n'est donc pas fonctionnelle pour des programmes réels qui utilisent de nombreux appels systèmes.

3.6 Conclusion

Nous avons présenté les différents niveaux d'information qu'une sémantique du langage assembleur utilisé peut nous apporter et avons détaillé un langage intermédiaire fournissant une sémantique de niveau 3 et des exemples de code équivalent entre l'assembleur et ce langage intermédiaire. Ce langage intermédiaire permet de transcrire le comportement auto-modifiant de l'assembleur. Nous avons implémenté un émulateur partiel utilisant ce langage intermédiaire et avons utilisé une plateforme existante, BAP, que nous avons entendue à l'émulation de programmes auto-modifiants. Le chapitre suivant utilise ces notions afin de définir un méthode d'analyse adaptée aux programmes auto-modifiants.

Chapitre 4

Analyse dynamique de programmes auto-modifiants

Nous avons jusqu'à présent fait l'hypothèse que les programmes analysés ne sont pas auto-modifiants. Dans ce chapitre nous détaillons une approche d'analyse dynamique adaptée aux programmes auto-modifiants et ayant pour but de réduire l'analyse d'un programme auto-modifiant en l'analyse de plusieurs sous-ensembles non auto-modifiants du programme. Cette approche est donnée par la littérature existante, en particulier par les travaux de thèse de Reynaud [Rey10] et Calvet [Cal13].

L'analyse dynamique consiste à se baser sur une ou des exécutions particulières d'un programme pour inférer des propriétés sur son fonctionnement. Elle demande donc de se munir d'un modèle pour le fonctionnement de la machine durant l'exécution du programme, d'une sémantique concrète de l'assembleur utilisé et de lancer l'évaluation sémantique du programme sur une ou plusieurs entrées : nous reprendrons à cet effet le langage assembleur et la sémantique concrète définis au chapitre précédent comme référence.

4.1 Auto-modification et vagues

Prenons le programme auto-modifiant donné à la figure 4.1 et analysons une exécution de ses instructions 1 à 5. Deux instructions provoquent une auto-modification. L'instruction 2 à l'adresse `0x8048065` va provoquer une écriture à l'adresse pointée par la valeur de `eax` incrémentée de 1, soit à l'adresse `0x8048077`. Cette écriture modifie l'instruction 4 en `mov edi, 2`, codée sur les octets `bf 02 00 00 00`. De même l'instruction 3 modifie l'instruction 5 en `mov ebx, 2`, codée sur `bb 02 00 00 00`.

Au vu de l'enchaînement des instructions, on peut construire trois représentations en mémoire des parties exécutables du programme. La première correspond à la vision du programme lors de son chargement : la section `.text` est dans son état initial, donné en figure 4.1. La seconde représentation est celle après la première modification du programme réalisée par l'instruction 2 et la troisième après la seconde modification effectuée par l'instruction 3. En fait vu qu'aucune des adresses modifiées par la première instruction auto-modifiante n'est exécutée avant que la seconde modification ne soit faite, on

i	Adresse	Octets	Instruction
1	8048060	b8 6b 80 04 08	mov eax, 0x8048076
2	8048065	66 c7 40 01 02 00	mov [eax+1], 2
3	804806b	66 c7 40 06 02 00	mov [eax+6], 2
4	8048076	bf 01 00 00 00	mov edi, 1
5	804807b	bb 01 00 00 00	mov ebx, 1

FIGURE 4.1 – Programme auto-modifiant

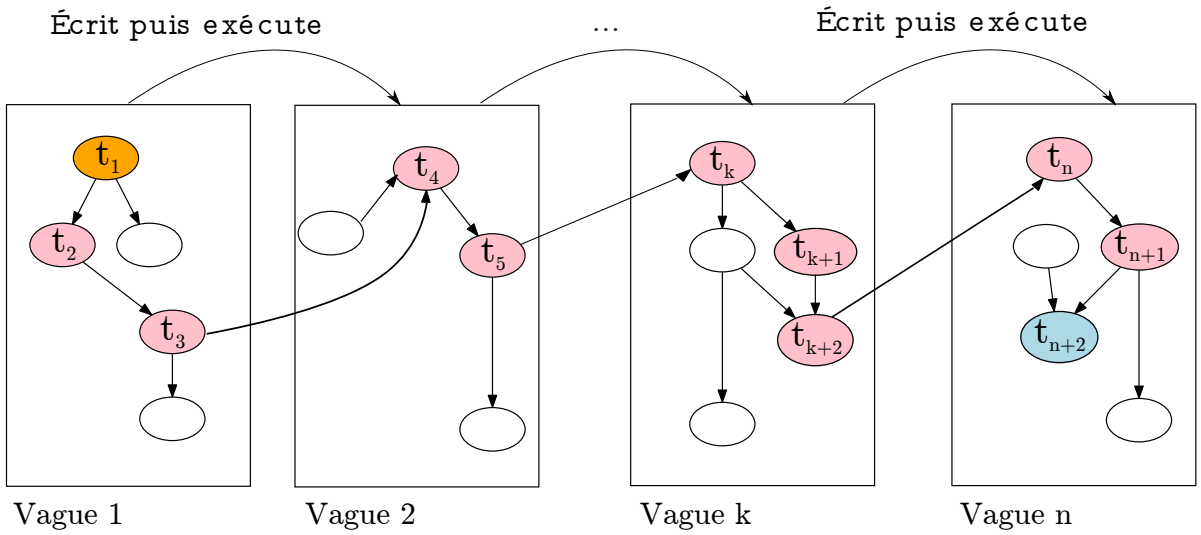


FIGURE 4.2 – Vision informelle des vagues

peut regrouper les deux instructions auto-modifiantes et considérer que le programme n'a que deux représentations en mémoire : la représentation initiale et la représentation après que l'instruction 3 a été exécutée. Beaucoup de logiciels protégés restaurent de grandes parties de code au démarrage : un grand nombre d'instructions sont modifiées sans que les instructions modifiées ne soient exécutées. Il nous paraît plus cohérent que toutes ces modifications soient réunies puisqu'elles correspondent à une étape lors de l'exécution du programme, l'étape suivante étant l'exécution d'une de ces instructions modifiées.

Dans ce découpage informel on appelle vague le désassemblage d'un instantané de la mémoire, pris à un instant donné. L'exécution d'un programme est alors caractérisée par une suite d'exécutions sur des vagues successives comme représenté en figure 4.2. Les instructions qui sont présentes dans la trace sont colorées en rose tandis que le point d'entrée et la dernière instruction sont en orange et bleu clair, respectivement. On passe d'une vague k à la vague suivante $k+1$ lorsqu'une adresse mémoire écrite dans la vague k est exécutée. Ainsi dans une vague k, toutes les instructions exécutées ont été écrites au moins à la vague $k-1$. En ce sens chacune des vagues, prise indépendamment des autres, ne présente pas d'auto-modification.

Nous détaillerons par la suite, formellement, ce qu'est une trace d'exécution pour l'analyse dynamique ainsi que la sémantique d'enchaînement des vagues.

4.2 Trace, niveaux d'exécution et vagues

Le programme exécuté a pour sources principales de données les registres et la mémoire constituée de la pile et du tas qui sont tous les deux adressables par des entiers. Une variable d'un programme est donc soit un registre du processeur soit une adresse mémoire, de même que défini dans la sémantique du langage intermédiaire défini au chapitre précédent (définition 3.1).

En utilisant la sémantique concrète précédemment définie on est capable, à partir d'un ensemble de valeurs initiales pour les registres et la mémoire, d'exécuter un programme sur cette entrée. Schématiquement, l'exécution d'un programme consiste en la définition d'un contexte d'exécution initial E_0, en l'évaluation concrète de la première instruction D_1 dans ce contexte, puis en l'évaluation de l'instruction suivante dans le contexte mis à jour, et ainsi de suite. Dans cette partie nous définirons formellement ces éléments mais leur enchaînement est schématiquement le suivant.

$$E_0 \xrightarrow{D_1} E_1 \xrightarrow{D_2} E_2 \xrightarrow{\dots} \dots \xrightarrow{D_n} E_n$$

Nous rappelons que nous notons, à tout instant durant l'exécution, $\mathbb{T}$ le tableau contenant les adresses mémoires et Θ le store représentant l'état des variables (mémoire et registres), comme indiqué aux définitions 3.1 et 3.4. Nous définissons une instruction dynamique (définition 4.1) par son adresse, les adresses mémoires sur lesquelles elle est codée et l'instruction machine correspondant. Ces informations sont données par le décodage d'une instruction à l'adresse mémoire spécifiée dans un contexte donné.

Définition 4.1. *On note D une instruction dynamique constituée des éléments suivants.*
- *$\mathcal{A}[D]$ l'adresse mémoire de l'instruction dynamique*
- *$\mathcal{C}[D]$ l'intervalle des adresses mémoire sur lequel D est codée*
- *$\mathcal{I}[D]$ l'instruction machine à l'adresse $\mathcal{A}[D]$*

On définit l'opérateur `decode` *qui associe à une adresse mémoire a et un store Θ l'instruction dynamique $D =$* `decode`(a, Θ) *présente à l'adresse a.*

Si l'on dispose d'un langage intermédiaire comme défini au chapitre précédent, son opérateur de désassemblage atomique nous donne ces informations. Pour l'assembleur x86, une instruction est codée au maximum sur 15 octets. On cherche à désassembler l'exemple précédent à l'adresse `0x8048060`. À l'adresse `0x8048060` sont présents les octets suivants : Θ[`0x8048060..0x804806e`] = `b8 6b 80 04 08 b8 6b 80 04 08 66 c7 40 06 02`, l'opérateur `decode` renvoie la première instruction dynamique D à cette adresse : il s'agit de l'instruction assembleur $\mathcal{I}[D]$=`mov eax, 0x8048076` (d'une taille de cinq octets : `b8 6b 80 04 08`) à l'adresse $\mathcal{A}[D]$=`0x8048060`, codée sur l'intervalle d'adresses $\mathcal{C}[D]$=[`0x8048060..0x8048064`].

Afin de pouvoir séparer la trace d'exécution selon le moment où chaque instruction a été écrite, nous définissons aussi, pour une instruction dynamique et un store Θ, l'ensemble des adresses mémoires sur lesquelles l'instruction provoque une écriture (définition 4.2).

Définition 4.2. *Soit D une instruction dynamique et Θ un store représentant l'état des variables (mémoire et registres). On note $\mathcal{W}^{\Theta}[D]$ l'ensemble des adresses mémoires sur lesquelles l'exécution de D provoque une écriture.*

Cette information est donnée par la sémantique concrète choisie : si l'instruction assembleur $\mathcal{I}[D]$ est sémantiquement équivalente, avec le store Θ, à la suite d'instructions atomiques $d_1, ..., d_n$ dans un langage intermédiaire alors l'ensemble des adresses écrites par D est l'union des adresses écrites par chacune des instructions atomiques d_i dans la sémantique concrète choisie pour ce langage intermédiaire (propriété 4.1).

Propriété 4.1. *Soit D une instruction dynamique, Θ un store représentant l'état des variables (mémoire et registres) et $d_1, ..., d_n$ une suite d'instructions atomiques dans un langage intermédiaire telle que la suite d'instructions $d_1, ..., d_n$ dans le langage intermédiaire est sémantiquement équivalente à l'instruction assembleur $\mathcal{I}[D]$. On a :*

$$\mathcal{W}^{\Theta}[D] = \mathcal{W}^{\Theta}[d_1, ..., d_n] = \mathcal{W}^{\Theta}[d_1] \ \cup \ ... \ \cup \ \mathcal{W}^{\Theta}[d_n].$$

Avec la sémantique définie au chapitre précédent les seules instructions qui provoquent des écritures sont celles dont la liste d'instructions atomiques donnée par le désassemblage contiennent des assignations écrivant, par adressage direct ou indirect, à une adresse mémoire m. Si $\mathcal{I}[D]$ est une instruction assembleur sémantiquement équivalente, avec le store Θ, à l'enchaînement des instructions atomiques $d_1, ..., d_n$ dans le langage intermédiaire précédent, alors $\mathcal{W}^{\Theta}[D]$=$\mathcal{W}^{\Theta}[d_1] \cup ... \cup \mathcal{W}^{\Theta}[d_n]$ (propriété 4.1) avec

$$\mathcal{W}^{\Theta}[d_i] = \begin{cases} m & \text{si } d_i = m \leftarrow g(x_1, ..., x_k) \ et \ m \in \mathbb{T} \\ \Theta(v) & \text{si } d_i = [v] \leftarrow g(x_1, ..., x_k) \ et \ \Theta(v) \in \mathbb{T} \\ \emptyset & \text{sinon.} \end{cases}$$

Nous définissons plus formellement la notion de contexte d'exécution (définition 4.3) comme la donnée d'un niveau d'exécution, d'un store contenant les valeurs de la mémoire et des registres et d'un store contenant les niveaux d'écriture courants de chaque adresse mémoire.

Définition 4.3. *Un contexte d'exécution est la donnée d'un triplet $E = (X, \Theta, W)$ où*
- *$X \in \mathbb{N}$ est le niveau d'exécution du contexte*
- *Θ est le store des valeurs du contexte, associant une valeur à chaque registre et chaque adresse mémoire*
- *W est le store des niveaux d'écriture du contexte, associant à chaque adresse mémoire un niveau d'écriture dans $\mathbb{N}$*

Une exécution (définition 4.4) consiste en une série de contextes d'exécution dont la transition de l'un au suivant est provoquée par l'évaluation sémantique de l'instruction pointée par le registre de compteur ordinal (noté en général `pc` ou `eip` en assembleur x86).

Définition 4.4. *Une exécution d'un programme, dont le point d'entrée est ep, est la donnée d'une suite finie de contextes d'exécution* $E_0, E_1, ..., E_n$ *tel que :*

- $X_0 = 1$, $\Theta_0[eip] = ep$ *et* $\forall m \in \mathbb{T}, W_0[m \leftarrow 0]$
- *En notant* $D_{i+1} = \texttt{decode}(\Theta_i[eip], \Theta_i)$ *l'instruction dynamique exécutée lors de la transition entre le contexte* E_i *et* E_{i+1}*, on a :*
  - *Le niveau d'écriture de* D_{i+1} *est le niveau d'écriture maximum des octets qui la composent :* $W_D = max(\{W[m],\ m \in \mathcal{C}[D_{i+1}]\})$
  - $X_{i+1} = max(X_i, W_D + 1)$
  - Θ_{i+1} *est* Θ_i *mis à jour par l'évaluation sémantique de* $\mathcal{I}[D]$
  - $W_{i+1} = W_i$ *sauf pour les adresses mémoires écrites par* D_{i+1} *:* $\forall m \in \mathcal{W}^{\Theta}[D_{i+1}],\ W_{i+1}[m \leftarrow X_i]$

Une trace d'exécution est une signature d'une exécution. Elle contient les instructions successives avec leur niveau d'exécution respectif (définition 4.5).

Définition 4.5. *Étant donnée une exécution d'un programme composée des contextes d'exécution* $E_0, E_1, ..., E_n$ *avec* $E_i = (X_i, \Theta_i, W_i)$*, on appelle trace d'exécution la suite* $T = (t_1, t_2, ..., t_n)$ *où* $t_i = (i, X_{i-1}, D_i)$ *avec* $D_i = \texttt{decode}(\Theta_{i-1}[eip], \Theta_{i-1})$*.*

Nous avons donc, pour une exécution donnée, des niveaux d'exécution successifs $1, 2, ..., n$. À chaque contexte d'exécution toute adresse en mémoire m a un niveau d'écriture $W[m]$ correspondant au dernier niveau d'exécution durant lequel une instruction a modifié la valeur à l'adresse m. Une instruction D a un niveau d'écriture W_D qui est le niveau d'écriture le plus élevé parmi les adresses sur lesquelles elle est codée. Lors d'une exécution le niveau d'exécution ainsi que le niveau d'écriture de chaque adresse mémoire sont croissants.

En pratique une instruction D écrite par une instruction ayant pour niveau d'exécution k puis directement exécutée aura pour niveau d'écriture $W_D = k$ et pour niveau d'exécution $X = k + 1$. On définit alors un instantané du niveau d'exécution k selon la définition 4.6 et une vague comme étant le graphe de flot de contrôle parfait de cet instantané (définition 4.7).

Définition 4.6. *Étant donnée une exécution d'un programme composée des contextes d'exécution* $E_0, E_1, ..., E_n$ *avec* $E_i = (X_i, \Theta_i, W_i)$*, on appelle instantané du niveau d'exécution k l'état de la mémoire contenu dans* Θ_j *où* E_j *est le premier contexte d'exécution dont le niveau d'exécution est* $X_j = k$*. On appelle point d'entrée et point de sortie de cet instantané respectivement* $D_{in} = \texttt{decode}(\Theta_j[eip], \Theta_j)$ *et* $D_{out} = \texttt{decode}(\Theta_l[eip], \Theta_l)$ *où* E_l *est le dernier contexte d'exécution dont le niveau d'exécution est* $X_l = k$*.*

Définition 4.7. *Étant donnée une exécution d'un programme, on appelle vague k le graphe de flot de contrôle parfait du programme représenté par l'instantané du niveau d'exécution k muni de son point d'entrée et considéré comme non auto-modifiant.*

Une première vague est définie dès l'exécution de la première instruction du programme puis à chaque changement de niveau d'exécution une nouvelle vague est construite. L'algorithme 4.3 (utilisant les algorithmes 4.1 et 4.2) permet d'exécuter un programme dynamiquement avec la sémantique concrète choisie tout en déterminant les niveaux d'exécution

Algorithme 4.1 : Mise à jour du niveau d'exécution d'une instruction

Entrées : La mémoire, l'opérateur de niveau d'écriture, une instruction dynamique et le niveau d'exécution courant

Résultat : Le niveau d'exécution courant mis à jour

`MAJExecution(`M, W, D, X`)`

$W_D \leftarrow max(W[a],\ a \in \mathcal{C}[D])$

$X \leftarrow max(X,\ W_D + 1)$

retourner X

Algorithme 4.2 : Mise à jour des niveaux d'écriture lors de l'exécution d'une instruction

Entrées : La mémoire, l'opérateur de niveau d'écriture, une instruction dynamique et le niveau d'exécution courant

Résultat : L'opérateur de niveau d'écriture mis à jour

`MAJEcriture(`M, W, D, X`)`

pour $m \in \mathcal{W}[D]$ **faire**

  $W[m] \leftarrow X$

fin

retourner W

et d'écriture au fur et à mesure de l'exécution. Il fournit en sortie la trace d'exécution et la liste des instantanés d'exécution reconstruits.

Reprenons l'exemple du programme auto-modifiant précédent. La figure 4.3 donne une trace d'exécution de ce programme en détaillant les informations sur chaque instruction dynamique ainsi que les niveaux d'écriture et d'exécution de chaque instruction. Au départ toute la mémoire est dans son état d'origine et a pour niveau d'exécution 0. Lorsque l'instruction D_1 est exécutée, il n'y a pas eu d'auto-modification donc le niveau d'écriture est 0 et le niveau d'exécution est 1. Les instructions D_2 et D_3 provoquent une auto-modification : les octets aux adresses `0x8048077` et `0x804807c` sont modifiés et leurs niveaux d'écriture deviennent donc le niveau d'exécution courant, soit 1. Lorsque l'exécution atteint D_4, qui a été modifié, le niveau d'écriture est 1 donc le niveau d'exécution devient 2. L'instruction suivante D_5 a également un niveau d'écriture de 1 donc le niveau d'exécution est inchangé.

i	$\mathcal{A}[D_i]$	$\mathcal{C}[D_i]$	$\mathcal{I}[D_i]$	$\mathcal{W}[D_i]$	W_i^{Θ}	X_i
1	8048060	[8048060, 8048064]	mov eax, 0x8048076		0	1
2	8048065	[8048065, 804806a]	mov [eax+1], 2	0x8048077	0	1
3	804806b	[804806b, 8048075]	mov [eax+6], 2	0x804807c	0	1
4	8048076	[8048076, 804807a]	mov edi, 1		1	2
5	804807b	[804807b, 804807f]	mov ebx, 1		1	2

FIGURE 4.3 – Trace d'exécution du programme auto-modifiant de la figure 4.1

Cette exécution est donc séparée en deux vagues : la vague initiale, v_0 dont l'instan-

Algorithme 4.3 : Analyse dynamique avec calcul des vagues

Entrées : Les registres R et une mémoire M dans laquelle un programme de point d'entrée `ep` a été chargé

Résultat : La trace des instructions dynamiques chacune associée à leur niveau d'exécution et les différentes vagues de la trace

```
analyseDynamique(R, M, ep)
    pour m ∈ M faire
        W[m] ← 0
    fin
    (X, X_{-1}, i, T, instantanes, eip) ← (1, 0, 1, ∅, ∅, ep)
    tant que la fin du programme n'est pas atteinte faire
        /* Le programme est exécuté en prenant l'instruction suivante à
           l'adresse eip                                                 */
        D ← decode(eip, M)
        X ← MAJExecution(M, W, D, X)
        si X ≠ X_{-1} alors
            /* Quand le niveau d'exécution change, on prend un
               instantané de la mémoire                                  */
            instantanes ← instantanes ∪ {(X_{-1}, M)}
        fin
        X_{-1} ← X

        /* On met à jour le contexte à partir de l'instruction courante
           en l'évaluant sémantiquement                                  */
        (eip, R, M) ← sem_eval(eip, R, M)
        W ← MAJEcriture(M, W, D, X)
        T ← T ∪ {(i, X, D_i)}
        i ← i + 1
    fin
    retourner T, instantanes
```

tané est l'état de la mémoire avant l'exécution de la première instruction et la vague v_1 contenant l'état de la mémoire juste après l'exécution de D_3 et avant l'exécution de la première instruction modifiée D_4.

Non unicité des instructions dynamiques dans la trace. Étant donné la définition croissante des niveaux d'exécution, une instruction dynamique peut-être exécutée non seulement plusieurs fois au même niveau d'exécution mais également être présente à des niveaux d'exécution différents.

Exécution d'un programme avec une entrée. Supposons que l'on dispose d'un programme P dont le point d'entrée est *ep* et prenant une entrée. Nous appelons exécution de ce programme sur une entrée I composée d'états initiaux pour les registres I_R et pour la mémoire I_M, sous réserve que le programme P est correctement chargé dans I_M, la donnée

de la trace et des instantanés d'exécution résultant de l'application de l'algorithme 4.3, soit `analyseDynamique(`I_R`, `I_M`, ep)` que l'on notera par la suite `exécution(P, I, ep)`.

4.3 Revue de littérature

La notion de vague présentée dans ce chapitre a été développée dans les thèses de Reynaud [Rey10] et Calvet [Cal13]. Elle est similaire à la notion de *phase* présentée par Debray et Patel [DP10] et utilisée pour automatiser la suppression de la protection d'un binaire. Le découpage d'une trace en phases est, chez eux, identique au découpage en vagues que l'on présente dans ce chapitre. En particulier le programme auto-modifiant précédent (figure 4.1), exécuté, forme deux vagues : la vague initiale est le programme initial et la seconde, modifiée, est identique à l'exception des instructions 4 et 5 qui ont été modifiées en `mov edi, 2` et `mov ebx, 2` respectivement. En général la suppression des protections se fait à l'aide d'une analyse dynamique et d'une image de la mémoire à un instant donné au cours de l'exécution. C'est cette image mémoire qui sera considérée comme étant le programme d'origine. La difficulté réside alors dans le choix de l'instant où prendre l'image mémoire : il s'agit souvent de la dernière phase.

Preda, Giacobazzi, Debray, Coogan et Townsend [Pre+10] effectuent un découpage en phases mais chaque exécution d'une instruction écrite lors d'une vague précédente (pas uniquement lors de la phase en cours) provoque la création d'une nouvelle phase. Avec l'exemple précédent, leur approche crée trois phases : la phase initiale, celle où uniquement l'instruction 4 a été modifiée, puis celle où les instructions 4 et 5 ont été modifiées.

4.4 Implémentations

Plusieurs choix s'offrent à qui cherche à implémenter un système d'analyse dynamique de binaire tels l'émulation, l'instrumentation et le débogage.

L'émulation consiste à lancer l'exécution dans un environnement d'exécution simulé, qui peut être un système d'exploitation complet comme c'est le cas avec BAP ou TEMU, le module d'analyse dynamique du projet BitBlaze [Son+08], basé sur l'émulateur QEMU [Bel07].

On instrumente un binaire exécuté en y insérant, généralement au cours de son exécution, du code assembleur servant à son analyse. Intel développe PinTools [Luk+06] pour l'analyse de programmes tournant sur leurs processeurs.

Enfin le débogage suit pas à pas l'exécution d'un programme en utilisant le drapeau de trappe (*Trap Flag*), permettant de reprendre la main après chaque instruction du programme débogué afin d'examiner son environnement d'exécution.

Le débogage comme l'instrumentation n'utilisent pas de langage intermédiaire tandis qu'un émulateur tel que BAP transcrit d'abord les instructions dans son langage intermédiaire pour les exécuter avec la sémantique concrète du langage intermédiaire. L'émulation est donc intéressante parce qu'à aucun moment le programme analysé n'a un accès libre au système sur lequel il s'exécute. La limitation est donc que les interactions

du programme émulé avec le système d'exploitation visé sont restreintes. En particulier les appels systèmes, qui ne sont pas transcrits par BAP, ne peuvent pas être émulés directement, rendant l'analyse très partielle. Les approches nécessitant une exécution non restreinte sur le système sont alors réalisées au sein d'une machine virtuelle.

Une caractéristique cruciale d'un analyseur dynamique est qu'il doit être transparent : le programme analysé ne doit pas être capable de différencier son exécution dans l'analyseur de son exécution sur un système réel. Cette transparence est en général partielle, que ce soit avec un émulateur, un débogueur ou une technique d'instrumentation [Fer07].

L'instrumentation, par rapport au débogage, offre des performances temporelles d'exécution supérieures. Pour ces raisons, nous nous sommes intéressés à l'émulation comme à l'instrumentation.

L'émulation permet une analyse plus abstraite et nous avons développé un analyseur partiel de programmes auto-modifiants basé sur BAP. L'instrumentation permet d'exécuter plus fidèlement le programme à analyser, nous avons donc principalement favorisé cette approche pour l'analyse de programmes malveillants. Nous avons choisi Pin qui, sans fournir de sémantique concrète pour l'assembleur, permet d'obtenir d'une instruction dynamique l'ensemble des adresses sur lesquelles elle écrit, comme souhaité à la définition 4.2.

4.4.1 Émulation avec BAP

Nous avons repris l'implémentation d'un émulateur pour programmes auto-modifiants basée sur BAP présentée au chapitre 3. Nous avions modifié BAP afin de pouvoir exécuter des instructions une à une et avons ainsi réalisé un émulateur de programmes auto-modifiants, fonctionnalité qui n'est pas supportée telle quelle par BAP.

Nous y avons ajouté la séparation de la trace en plusieurs vagues d'exécution en implémentant l'algorithme 4.3. Cette implémentation donne le résultat attendu avec des programmes simples et qui n'utilisent aucun appel système. Sur l'exemple de programme auto-modifiant donné en début de chapitre, la sortie est la suivante. L'exécution, de la première (*pc : 1*) à la dernière instruction, est correctement découpée en vagues et les instructions correspondent à celles modifiées lors de l'exécution.

```
Vague 1, pc: 1: addr 0x8048060 @asm "mov    eax, 0x8048071"
Vague 1, pc: 2: addr 0x8048065 @asm "movw   [eax+1], 2"
Vague 1, pc: 3: addr 0x804806b @asm "movw   [eax+6], 2"
Vague 2, pc: 4: addr 0x8048071 @asm "mov    edi, 2"
Vague 2, pc: 5: addr 0x8048076 @asm "mov    ebx, 2"
```

Ne pouvant pas émuler des programmes utilisant des appels systèmes, nous avons laissé de coté cette approche pour une approche par instrumentation permettant l'analyse de programmes réels.

4.4.2 Instrumentation avec Pin

Pin est l'outil d'instrumentation développé par Intel pour ses processeurs x86 et x86_64. Il fournit une information sémantique de niveau 2, suffisante pour suivre chaque instruction et connaître l'ensemble des adresses mémoires que l'instruction modifie. Nous avons implémenté l'algorithme 4.3 sous la forme d'un PinTool : codé en C, il s'agit d'un programme définissant les actions à effectuer avant et après l'exécution de chaque instruction selon le type de l'instruction considérée.

L'analyse d'un programme malveillant a lieu sur une machine virtuelle munie de Pin et ne souffre pas des limitations de l'émulation : les appels systèmes sont correctement analysés et exécutés. Dans la suite de cette thèse, nous avons exclusivement utilisé cette approche pour l'analyse dynamique.

4.5 Conclusion

Afin d'analyser les programmes auto-modifiants, nous proposons de découper toute exécution en une trace et une suite d'instantanés d'exécution : chaque instantané représente une partie non auto-modifiante du programme. Nous avons implémenté ce découpage par émulation avec BAP et par instrumentation avec Pin.

Le chapitre suivant est consacré à l'analyse statique du problème de chevauchement de code. Le chapitre 6 portera sur l'analyse d'un programme auto-modifiant : nous combinerons les notions abordées dans ce chapitre et le chapitre suivant afin de reconstruire les vagues correspondant à chaque niveau d'exécution ainsi qu'un graphe de flot de contrôle global pour le binaire étudié.

Chapitre 5

Traitement du chevauchement de code par analyse statique

Nous cherchons à parcourir le plus de code atteignable dans le but d'approximer le graphe de flot de contrôle parfait. Il faut alors pallier d'une part à l'incomplétude d'un parcours récursif du à la présence de sauts dynamiques et d'autre part au manque de précision d'un parcours linéaire. Une des difficultés vient de l'obscurcissement par chevauchement de code.

Dans ce chapitre nous commençons par dresser un état de l'art de quelques techniques d'analyse statique et de leur approche du chevauchement de code, puis nous proposerons une caractérisation des binaires utilisant cette technique. Nous ne considérons que des programmes non auto-modifiants.

5.1 Revue de littérature

Bien que le problème du chevauchement de code ne soit pas une technique d'obscurcissement récente et soit bien documenté [SH12], la littérature portant sur le désassemblage fait souvent l'hypothèse qu'un octet à une adresse spécifique ne peut être présent que dans une seule instruction [Krü+06]. Cette contrainte empêche de détecter tout chevauchement mais permet un désassemblage plus précis sur un binaire qui n'utilise pas cette technique de protection.

Parcours spéculatif. Le parcours récursif étant moins sensible à des obscurcissements très simples comme l'injection de code mort, il est souvent pris comme point de départ dans les recherches sur le désassemblage statique. Une fois une première recherche de code avec un parcours récursif effectuée, les octets restants peuvent subir un parcours linéaire qui cherchera à déterminer s'ils sont du code ou des données à l'aide d'heuristiques. Une de ces approches évalue la probabilité qu'une suite d'octets soit effectivement du code en apprenant au préalable des suites d'octets codant réellement des instructions lancées lors de l'exécution de programmes [KDF09]. On appelle cet enchaînement des deux parcours un parcours spéculatif.

Prasaf et Chiueh [PC03], après un premier désassemblage récursif, tentent d'identifier les adresses de début de fonctions assembleur à l'aide de la suite d'instruction `push` puis `mov`, caractéristique de fonctions compilées : elles commencent par empiler le pointeur de pile de base avec `push ebp` pour le remplacer par le pointeur de pile avec `mov ebp, esp`. Ils identifient également toutes les adresses où est codée une instruction valide. À partir de ces adresses ils effectuent à nouveau un parcours récursif jusqu'à une instruction de saut inconditionnel (`ret` out `jmp`), estimant que la séquence de code identifiée s'arrête sur ce saut. Le risque étant grand que des chemins parcourus ainsi ne soient pas valides, ils éliminent alors les chemins qui aboutissent sur du code invalide ou des adresses connues pour être des données.

Krügel, Roberston, Valeur et Vigna [Krü+06] proposent aussi de séparer le code en fonctions assembleur. Ils effectuent une première analyse récursive pour détecter le maximum d'instructions atteignables et, lorsque que deux ou plusieurs instructions atteignables se chevauchent, ils considèrent qu'il s'agit d'un conflit qui se résoudra par le choix d'une des instructions. Leurs hypothèses sont : (i) les instructions ne peuvent pas se chevaucher, (ii) les sauts conditionnels peuvent être suivis ou non, (iii) le binaire peut contenir du code mort et (iv) le code suivant une instruction `call` n'est pas nécessairement accessible. Pour résoudre les conflits de chevauchement ils favorisent les instructions atteignables depuis le point d'entrée (que l'on sait accessible), puis ils considèrent qu'une instruction permettant d'atteindre directement deux instructions qui se chevauchent n'est pas valide et enfin ils favorisent les instructions les plus connectées au graphe de flot de contrôle. Ils prennent ainsi en compte l'ajout de code mort et certaines modifications du flot de contrôle.

Schwarz, Debray et Andrews [SDA03] introduisent un parcours linéaire capable de détecter certaines injections de données dans le code, comme les tables de saut (souvent présent dans du code compilé en C ou C++). Ils proposent de combiner les parcours récursifs et linéaires pour détecter les anomalies dans le désassemblage. Si, au sein d'une fonction, une instruction provenant du désassemblage récursif chevauche une instruction provenant du désassemblage linéaire, le désassemblage de la fonction est considéré comme contenant une erreur.

Partant du principe qu'un désassemblage correct ne contient pas de chevauchement de code, ces approches ne sont pas satisfaisantes pour désassembler des binaires utilisant du chevauchement de code, tels les programmes protégés avec tElock.

Désassembleurs disponibles. Les désassembleurs existants, qu'ils utilisent un parcours linéaire ou récursif, font l'hypothèse que le code ne peut pas se chevaucher et ne parviennent pas à afficher un désassemblage cohérent dans le cas contraire.

Le désassemblage récursif de l'exemple de tElock (figure 2.5) avec IDA Pro (version 6.3) [Hexb] est le suivant :

```
01006E7A    inc     byte ptr [ebx+ecx]
01006E7D    jmp     short near ptr loc_1006E7D+1
; Les octets qui suivent n'ont pas été désassemblés
01006E7F    db 0C9h
01006E80    db  7Fh
01006E81    db 0E6h
```

```
01006E82    db  8Bh
01006E83    db  0C1h
```

Radare [rad] effectue le désassemblage linéaire suivant :

```
01006e7a    fe 04 0b    inc byte [ebx+ecx]
01006e7d    eb ff       jmp 6e7e
01006e7f    c9          leave
01006e80    7f e6       jg 6e68
01006e82    8b c1       mov eax, ecx
```

Ni l'un ni l'autre n'est capable de suivre le saut de l'instruction `jmp` : la cible du saut a déjà été prise en compte comme faisant partie d'une autre instruction.

De même ni Radare ni IDA ne détectent le second chemin d'exécution dans l'extrait d'UPX (figure 2.8) et désassemblent cet extrait comme suit.

```
010059f0    89 f9          mov ecx, edi
010059f2    79 07          jns 0x10059fb
010059f4    0f b7 07       movzx eax, word [edi]
010059f7    47             inc edi
010059f8    50             push eax
010059f9    47             inc edi
010059fa    b9 57 48 f2 ae mov ecx, 0xaef24857
010059ff    55             push ebp
```

Approches prenant en compte le chevauchement. Les auteurs de la technique de chevauchement détaillée au chapitre 2, permettant d'encoder une séquence de code cachée dans une séquence [JLH13], proposent de détecter la protection qu'ils exposent. L'idée est qu'il est improbable qu'une longue séquence d'octets représente une séquence valide de code. Si une telle séquence existe, c'est sûrement du code. Ainsi si deux longues séquences valides de code se chevauchent, il s'agit d'un obscurcissement délibéré et une des deux séquences contient du code caché. Cette approche fonctionne pour la protection qu'ils exposent mais n'est pas applicable aux cas d'UPX par exemple car les séquences d'octets sur lesquels des instructions se chevauchent sont très courtes et il est plausible que le chevauchement soit accidentel et que le code en chevauchement ne soit pas atteignable.

Kinder [Kin10] indique que si la technique de désassemblage autorise que différentes instructions dans le graphe de flot de contrôle se chevauchent et réalise des désassemblages atomiques d'une seule instruction à la fois à partir de chaque adresse, alors on peut voir deux instructions qui se chevauchent comme indépendantes. Dans ce cas il n'y a pas besoin d'hypothèses spécifiques pour les instructions se chevauchant qui, de fait, coexistent dans le graphe de flot de contrôle.

Effectivement les travaux présentés dans cet état de l'art ainsi que les désassembleurs existants prennent pour hypothèse que le code ne doit pas se chevaucher alors que nous avons représenté les chevauchements de tElock et UPX très simplement. L'hypothèse d'alignement permet en général de simplifier les critères de désassemblage et est justifiée par la non occurrence de ce phénomène dans des programmes légitimes. Nous verrons qu'en pratique l'utilisation du chevauchement de code est rare même dans un binaire

protégé et nous proposerons une technique de désassemblage permettant de détecter les cas de chevauchement et ainsi de recenser l'usage de cette technique de protection.

5.2 Analyse statique du chevauchement de code

Nous proposons une formalisation du problème du chevauchement de code. Du point de vue du désassemblage, un programme qui présente une unique instruction qui en chevauche une autre peut se voir comme composé d'un chemin principal de désassemblage et d'un chemin secondaire dans lequel l'instruction en chevauchement se place. Reprenons l'exemple de tElock : le segment d'octets `eb ff c9 7f e6` peut se voir comme composé des deux couches de code données à la figure 5.1 : il y a deux couches, la première contient les instructions `jmp +1`, `leave` et `jg 0x1006e68` et la seconde contient l'instruction `dec ecx`, chevauchant `jmp +1`. En fait le segment d'octets `eb ff c9 7f e6` contient exactement les quatre instructions précédentes : il y a au maximum une instruction valide à chaque adresse et la dernière instruction potentielle, codée sur `e6`, n'est pas valide.

Adresses	0x01006e7d	0x01006e7e	0x01006e7f	0x01006e80	0x01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1		leave	jg 0x1006e68	
Couche 2		dec ecx			

FIGURE 5.1 – Découpage cohérent en couches de l'extrait de tElock

5.2.1 Définitions

Pour une instruction I on note l'intervalle d'adresses mémoires sur lesquelles elle est codée $\mathcal{C}[I]$, comme indiqué à la définition 4.1 du chapitre précédent. Formellement on définit une couche comme un ensemble d'instructions qui ne se chevauchent pas (définition 5.1). Par conséquent lors du désassemblage on cherche à effectuer un découpage cohérent des instructions inclues dans le graphe de flot de contrôle en différentes couches (définition 5.2). L'exemple précédent pour tElock est un découpage cohérent.

Définition 5.1. *Une couche de code L est un ensemble d'instructions qui ne se chevauchent pas :* $\forall D_1, D_2 \in L,\ \mathcal{C}[D_1] \cap \mathcal{C}[D_2] = \emptyset$.

Définition 5.2. *Étant donné un ensemble d'instructions E, un découpage cohérent est un ensemble de couches deux à deux disjointes et recouvrant l'ensemble des instructions de E.*

Borne du nombre minimal de couches. Le nombre minimal de couches permettant de former un découpage cohérent est borné par la taille maximale des instructions, c'est à dire 15 octets pour l'assembleur x86. Pour s'en convaincre il suffit de définir le découpage cohérent suivant. Pour un segment d'octets à l'adresse `0xa`, on place dans la couche i,

pour $1 \leq i \leq 15$ toutes les instructions aux adresses congrues à $a + i - 1 (mod\ 15)$. C'est à dire que la couche 1 contient les instructions aux adresses `a`, `a+15`, `a+30`, etc ; la couche 2 celles aux adresses `a+1`, `a+16`, `a+31`, etc. Puisque les instructions dans chaque couche sont distantes de 15 octets, il ne peut pas y avoir de chevauchement au sein d'une couche et les couches sont disjointes ; il s'agit bien d'un découpage cohérent une fois qu'on enlève les couches ne contenant pas d'instructions valides. Ce découpage contient au plus 15 couches. Un exemple d'une suite de 15 octets définissant 15 couche en utilisant des préfixes d'instruction est donné en figure 5.2.

Octets	26	26	26	26	26	2e	26	26	26	2e	e9	66	83	eb	40
Couche 1	jmp 0x40eb8375														
Couche 2		jmp 0x40eb8374													
Couche 3			jmp 0x40eb8373												
Couche 4				jmp 0x40eb8372											
Couche 5					jmp 0x40eb8371										
Couche 6						jmp 0x40eb8370									
Couche 7							jmp 0x40eb836f								
Couche 8								jmp 0x40eb836e							
Couche 9									jmp 0x40eb836d						
Couche 10										jmp 0x40eb836c					
Couche 11											jmp 0x40eb836b				
Couche 12												sub bx, 0x40			
Couche 13													sub ebx, 0x40		
Couche 14														jmp 0x42	
Couche 15															inc eax

FIGURE 5.2 – Exemple de séquence d'octets définissant 15 couches

Nous définirons par la suite plusieurs découpages cohérents en couches et discuterons de leur pertinence et de leurs implémentations.

5.2.2 Couches linéaires

Une approche naturelle des couches consiste à construire une première couche par parcours linéaire à partir du début de la section de code du binaire. Cette première couche contient exactement les instructions qu'un désassembleur par parcours linéaire aurait détecté. Les couches suivantes seront construites également par parcours linéaire, à partir de chaque adresse du binaire. Ainsi on peut construire une couche de code à partir de chaque adresse du binaire. Un tel découpage pour tElock donne les layers indiqués en figure 5.3.

Ce découpage est donc, par définition, basé sur l'alignement des instructions : au sein d'une couche, les instructions sont alignées, c'est à dire qu'un désassemblage linéaire depuis la première instruction de la couche parcourt toutes les autres instructions de la couche. Il est à noter qu'en assembleur x86 ou x86_64 les instructions ont tendance à

Adresses	0x01006e7d	0x01006e7e	0x01006e7f	0x01006e80	0x01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1		leave	jg 0x1006e68	
Couche 2		dec ecx		jg 0x1006e68	
Couche 3			leave	jg 0x1006e68	
Couche 4				jg 0x1006e68	
Couche 5					(invalide)

FIGURE 5.3 – Couches linéaires de l'extrait de tElock, les instructions dupliquées sont grisées

se réaligner rapidement, cette propriété a pour conséquence que la plupart des couches linéaires se réalignent, après quelques instructions, sur une couche précédente, en général la première. Sur la figure 5.3 les couches 1 et 2 se réalignent et partagent l'instruction `jg 0x1006e68`. Cette propriété n'est évidemment pas vérifiée si les instructions ont été spécialement choisies pour provoquer de longs chemins de chevauchement, comme avec l'obscurcissement précédemment cité [JLH13].

Nous souhaitons obtenir un découpage cohérent des couches sous forme d'ensembles deux à deux disjoints. Il suffit alors d'enlever des couches les instructions existant déjà dans les couches inférieures (colorées en gris sur la figure 5.3) et de ne garder que les couches contenant des instructions valides. Au final il reste les deux couches données précédemment à la figure 5.1, obtenues linéairement. L'algorithme 5.1 détaille le découpage cohérent en couches linéaires d'un programme composé d'un seul segment.

Binaire composé de plusieurs segments. On peut étendre la définition précédente applicable à un segment d'octets à un binaire composé de plusieurs segments de code disjoints. L'extension consiste à simplement à réaliser un découpage cohérent pour chaque segment puis à les réunir au sein d'un seul découpage par une union des couches qui les composent, lui aussi cohérent puisque les segments sont deux à deux disjoints.

Couches linéaires et désassemblage. Le découpage en couches linéaires n'est pas en soi un algorithme de désassemblage : il s'agit d'une analyse exhaustive de toutes les instructions codées dans le segment analysé et ne cherche pas à discriminer les instructions atteignables des autres. On peut voir ce découpage comme une caractéristique du binaire qui contient l'ensemble des instructions potentiellement exécutées (hors auto-modification) et qui dépend de leur agencement en mémoire. La première couche correspond à un désassemblage linéaire depuis la première adresse du segment analysé tandis les suivantes ne sont pas présentes dans le désassemblage linéaire.

Caractérisation des binaires obscurcis. Si chaque instruction du graphe de flot de contrôle, présente dans le segment s, fait partie de la première couche du découpage linéaire de s, il est clair qu'il n'y a pas de chevauchement de code possible. En pratique c'est le cas la plupart du temps avec des binaires non obscurcis.

Algorithme 5.1 : Découpage cohérent en couches linéaires

Entrées : Un programme P composé d'un unique segment
Résultat : Un découpage cohérent en couches

```
decoupageLineaire(P)
    debut ← premiereAdresse(P)
    fin ← dernieresAdresse(P)
    C ← ∅
    pour a de debut à fin faire
        // On cherche à construire une couche à partir de chaque
        adresse
        C ← C, coucheLineaire(a, C, P, fin)
    fin
    retourner C
coucheLineaire(a, C, P, fin)
    // On va désassembler linéairement de a à la dernière adresse
    mémoire de P
    c ← ∅
    tant que a ≤ fin faire
        // On récupère l'instruction de P présente à l'adresse a
        D ← decode(a, P)
        si ∀c' ∈ C, D ∉ c' alors
            c ← D
        fin
        a ← a + |C[D]|
    fin
    retourner c
```

Lors du désassemblage d'un binaire plusieurs métriques peuvent être observées. On peut d'une part observer combien de couches différentes sont utilisées : chaque couche, en dehors de la première de chaque segment, atteste du départ d'un chemin désaligné avec le chemin principal. D'autre part on peut compter le nombre de sauts d'une instruction d'une couche vers une instruction désalignée d'une autre couche. Dans le cas de tElock, le saut depuis l'instruction `dec ecx` à l'adresse `0x01006e7e` vers l'instruction `jg 0x1006e68` à l'adresse `0x01006e80`, bien qu'il provoque un changement de couche, n'est pas désaligné. Le nombre de ces sauts de désalignement atteste de la prévalence des chemins représentés par chaque couche.

5.2.3 Couches désassemblées par parcours récursif

Une autre approche consiste à construire des couches de code au fur et à mesure du désassemblage du binaire. Le désassemblage parfait de l'extrait de tElock précédent (figure 5.1) à partir du point d'entrée `0x1006e7d` contient exactement les trois instructions `jmp +1`, `dec ecx`, `jg 0x1006e68`. La première instruction va provoquer la création d'une première couche et la seconde instruction, en chevauchement avec la première, ne peut

être placée que dans une nouvelle couche. La dernière instruction peut être placée dans n'importe laquelle des deux couches existantes. On la placera, arbitrairement, dans la première couche. Un tel découpage est donné en figure 5.4.

Adresses	0x01006e7d	0x01006e7e	0x01006e7f	0x01006e80	0x01006e81
Octets	eb	ff	c9	7f	e6
Couche 1	jmp +1			jg 0x1006e68	
Couche 2		dec ecx			

FIGURE 5.4 – Découpage cohérent en couches lors du désassemblage récursif de tElock

L'algorithme 5.2 explicite le choix fait lors de l'insertion d'une instruction dans un découpage cohérent existant : on choisit simplement la première couche dont les instructions ne chevauchent pas l'instruction à ajouter. La création d'une nouvelle couche peut être nécessaire. Ce découpage est cohérent et résulte en un maximum de 15 couches.

Algorithme 5.2 : Ajout d'une instruction à un découpage cohérent en couches

Entrées : Un découpage cohérent C en n couches L_i, une instruction D
Résultat : Un découpage cohérent contenant l'instruction D

```
ajoutInstruction(C, D)
  si l'instruction D n'est pas comprise dans C alors
    pour i de 1 à n faire
      si ∀ instruction D' ∈ L_i, D et D' ne se chevauchent pas alors
        L_i ← L_i ∪ {D}
        retourner C
      fin
    fin
    L_{n+1} ← {D}
    C ← C ∪ {L_{n+1}}
    retourner C
  sinon
    retourner C
  fin
```

Choix du découpage. Dans la suite de cette thèse nous favorisons ce découpage pour la définition des couches parce qu'il nous permet d'observer les difficultés qu'impliquent le chevauchement de code lors d'un désassemblage, quel que soit l'algorithme de désassemblage choisi.

5.2.4 Parcours récursif

Nous appliquons la seconde méthode de découpage en couches à un désassemblage fonctionnant par parcours récursif. L'algorithme 1.2 du chapitre 1 explicite ce parcours.

Non-unicité du découpage. La technique de désassemblage que nous proposons est basée sur un parcours récursif. Il est à noter qu'avec ce type de parcours les couches

obtenues en applicant l'algorithme précédent dépendent de l'ordre dans lequel les fils de chaque sommet sont explorés.

Reprenons un graphe de flot simplifié de l'échantillon d'UPX donné au chapitre 2, donné en figure 5.5. Le sommet 1 est le point d'entrée, le sommet 4 le point d'arrêt et les instructions aux sommets 2 et 3 se chevauchent. Le sommet 1 a deux fils : le sommet 2 qui est accessible séquentiellement (les instructions se suivent) et le sommet 3 qui est la cible d'un saut.

Le découpage sera composé de deux couches dans tous les cas mais si l'on parcourt d'abord le sommet 2 alors les deux couches sont $L_1 = \{1, 2, 4\}$ et $L_2 = \{3\}$. Au contraire si l'on choisit de d'abord parcourir le sommet 3, les couches sont $L_1 = \{1, 3, 4\}$ et $L_2 = \{2\}$.

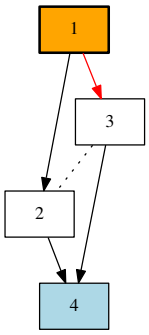


FIGURE 5.5 – Graphe de flot de contrôle simplifié de l'échantillon d'UPX

Nous faisons le choix de désassembler d'abord la cible du saut et ensuite l'instruction accessible séquentiellement.

Métriques. On utilisera le nombre de couches pour observer la complexité des chevauchements de code tandis que le nombre de sauts de changement de couches indique la fréquence d'utilisation de cette technique d'obscurcissement.

5.3 Conclusion

Le chevauchement de code est rarement abordé dans la littérature et, lorsqu'il l'est, il n'est pas analysé comme une technique d'obscurcissement à part entière mais plutôt contourné en ne prenant pas en compte l'intervalle d'adresses sur lesquelles sont codées les instructions désassemblées. Nous avons proposé une notion de couche de code, permettant d'observer et de quantifier l'utilisation du chevauchement de code par un binaire. Nous proposons un algorithme pour compter ces chevauchements lors du désassemblage récursif d'un binaire.

Cette approche sera reprise et implémentée dans le chapitre suivant afin d'évaluer l'utilisation du chevauchement par des logiciels malveillants.

Chapitre 6

Désassemblage d'un programme auto-modifiant

Nous avons précédemment détaillé plusieurs techniques d'obscurcissement de code. Nous avons décrit des méthodes d'analyse de programmes non auto-modifiants présentant des chevauchements de code et une méthode permettant de ramener l'analyse d'un programme auto-modifiant à l'analyse de plusieurs programmes non auto-modifiants par découpage en instantanés.

Dans ce chapitre nous présentons une méthode d'analyse combinant l'approche dynamique pour gérer les programmes auto-modifiants et une analyse statique de chaque sous programme considéré à partir d'un instantané et de la trace d'exécution.

6.1 Désassemblage parfait

Nous avons, au chapitre 1, défini le désassemblage parfait d'un programme non auto-modifiant comme la donnée de l'ensemble des instructions qui peuvent être atteintes lors de l'exécution. Dans le cas d'un programme auto-modifiant, cette définition n'est pas suffisante puisque l'instruction présente à une adresse dépend du contexte d'exécution et des éventuelles modifications qui ont été faites à cette adresse.

Nous considérons toujours que la donnée que l'analyste cherche à déterminer et représenter est l'ensemble des exécutions possibles du programme : nous notons E_T l'ensemble des traces d'exécution. En particulier une entrée spécifique provoque une exécution que nous observons sous la forme d'une trace. Cette trace peut être découpée en niveaux d'exécution et en autant d'instantanés de la mémoire (définition 4.6). Enfin chaque vague est le désassemblage parfait de l'instantané.

Le désassemblage, pris au sens de l'opération inverse de l'assemblage, consiste à déterminer l'ensemble des instructions potentiellement exécutées et à séparer les adresses contenant ces instructions des autres adresses contenant alors des données. Lors de l'exécution d'un programme auto-modifiant on peut séparer les instructions dont le niveau d'écriture est 0 et celles dont le niveau d'écriture est strictement supérieur à 0. Les instructions de cette seconde catégorie sont codées sur au moins un octet qui a été modifié

lors de l'exécution et dont la valeur a donc été modifiée. Ces instructions ne sont donc pas présentes dans la représentation d'origine du programme. Ainsi le désassemblage d'un programme auto-modifiant en tant qu'opération inverse de l'assembleur, ne prend en compte que les instructions dont le niveau d'écriture est 0. Par la suite nous nous intéresserons plutôt au désassemblage comme méthode d'analyse des exécutions d'un binaire et chercherons donc à y inclure toutes les instructions exécutables et pas seulement celles présentes dans sa représentation d'origine.

Nous étendons la notion de désassemblage parfait aux programmes auto-modifiants pour qu'elle inclue toutes les instructions exécutées (définition 6.1) en les caractérisant par leur adresse, l'instruction assembleur exécutée et le niveau d'exécution auquel elles ont été exécutées.

Définition 6.1. *Étant donné un programme P auto-modifiant et l'ensemble de ses traces d'exécution E_T, le désassemblage parfait de P est la donnée de l'ensemble des adresses exécutées dans une exécution du programme, de l'instruction alors exécutée et de son niveau d'exécution, c'est à dire $\{(X, \mathcal{A}[D], \mathcal{I}[D]), (i, X, D) \in T, T \in E_T\}$.*

6.1.1 Graphe de flot de contrôle simple

Le graphe de flot de contrôle ne peut plus se contenter de représenter une instruction par son adresse puisque plusieurs instructions différentes peuvent être présentes à la même adresse. Le graphe de flot de contrôle simple, extension du GFC parfait défini pour les programmes non auto-modifiants, est le graphe dont les sommets sont des couples (a, I), où I est une instruction présente à l'adresse a, provenant du désassemblage parfait du programme. Il y a un arc entre deux sommets si le second suit directement le premier dans une trace d'exécution (définition 6.2).

Définition 6.2. *Étant donné un programme auto-modifiant P et l'ensemble de ses traces d'exécution E_T, le graphe de flot de contrôle simple de P est le graphe orienté $G = (V, E)$ tel que :*

- $V = \{(\mathcal{A}[D], \mathcal{I}[D]), (i, X, D) \in T, T \in E_T\}$.
- $((a_1, I_1), (a_2, I_2)) \in E$ *si et seulement si* (a_1, I_1) *et* (a_2, I_2) *se suivent dans une trace :* $\exists T \in E_T,\ i \in \mathbb{N}, (i, X_1, D_1)$ *et* $(i+1, X_2, D_2) \in T$ *avec* $\mathcal{A}[D_1] = a_1,\ \mathcal{I}[D_1] = I_1,\ \mathcal{A}[D_2] = a_2,\ \mathcal{I}[D_2] = I_2$.

Le problème de cette définition d'un GFC pour un programme auto-modifiant est qu'elle ne permet pas de visualiser l'enchaînement des niveaux d'exécution et peut conduire à une interprétation biaisée des exécutions possibles.

Exemple. Prenons le programme donné en figure 6.1 dont le graphe de flot de contrôle construit par analyse statique à l'aide d'un parcours récursif est en figure 6.2.

La branche du saut effectué à l'adresse `0x8048063` dans le cas où `eax` est différent de zéro revient vers le point d'entrée, incrémente `eax` et passe à nouveau dans le saut conditionnel. Cette boucle provoquera à terme un dépassement d'entier sur la valeur `eax` qui sera remise à zéro et l'autre branche du saut sera appelée. Cette seconde branche a pour effet de modifier les instructions aux adresses `0x8048060`, `0x8048061` et `0x8048063`

Adresse	Octets	Instruction	Instruction écrite
8048060 <debut>	31 ff	xor edi,edi	
8048062	40	inc eax	
8048063	74 07	je 804806c <si_zero>	
8048065	bf 02 00 00 00	mov edi, 0x2	
804806a	eb f4	jmp 8048060 <debut>	
804806c <si_zero>	bf 01 00 00 00	mov edi, 0x1	
8048071	bb 60 80 04 08	mov ebx, 0x8048060	
8048076	66 c7 03 47 00	mov [ebx], 0x47	inc edi
804807b	66 c7 43 01 47 00	mov [ebx+0x1], 0x47	inc edi
8048081	66 c7 43 03 c3 00	mov [ebx+0x3], 0xc3	ret
8048087	eb d7	jmp 8048060 <debut>	

FIGURE 6.1 – Code auto-modifiant

pour y placer les instructions `inc edi`, `inc edi` et `ret` respectivement, puis de sauter sur l'adresse `0x8048060`, provoquant l'exécution des instructions écrites ainsi que de l'instruction `inc eax`, non modifiée.

Ainsi toutes les exécutions possibles, paramétrées par la valeur initiale d'`eax`, bouclent sur la branche conditionnelle un nombre fini de fois, qui peut être nul, et exécutent la seconde branche, qui met la valeur 1 dans `edi`, provoque une auto-modification, incrémente `edi` deux fois puis quitte la fonction. Dans toutes les exécutions possibles la valeur finale de `edi` est toujours 3.

Le GFC simple de ce programme est donné en figure 6.3. En raison du mélange des instructions à différents niveaux d'exécution, il n'est pas possible de distinguer les vagues d'exécution et un analyseur statique simple approxime la valeur de `edi` à 2 ou 3.

Cette difficulté n'est pas intrinsèque aux programmes auto-modifiants et provient du fait même que le graphe de flot de contrôle est sur une sur-approximation de l'ensemble des exécutions possibles. Il est cependant exacerbé dans le cas des programmes auto-modifiants puisque souvent les vagues successives n'ont que peu de code commun entre elles et il n'est pas cohérent de les mélanger dans le GFC.

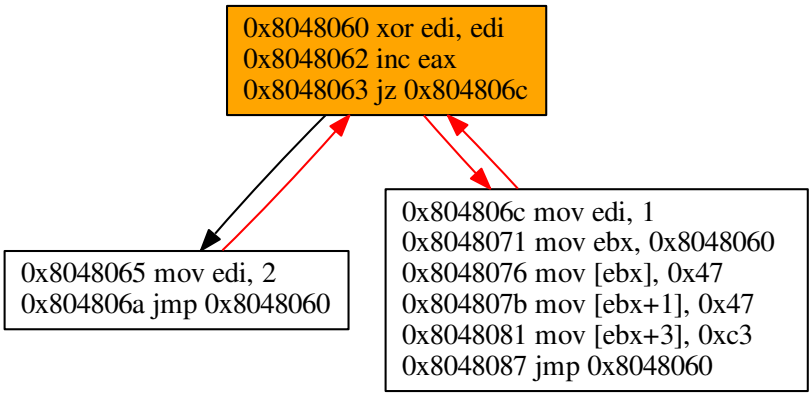


FIGURE 6.2 – GFC par analyse statique et récursive

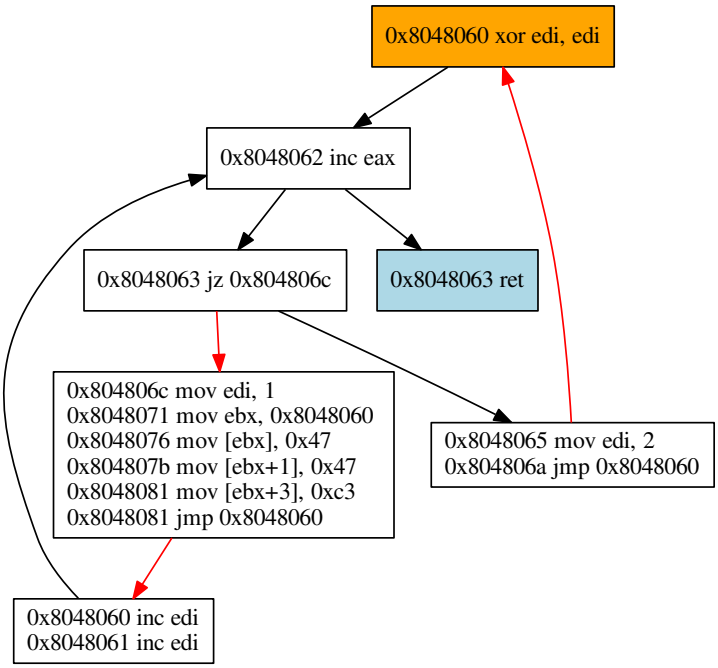


FIGURE 6.3 – GFC simple

6.1.2 Graphe de flot de contrôle parfait

Chaque trace est constituée d'une suite d'instructions et d'un enchaînement d'instantanés de la mémoire, chacun représentant la mémoire au cours d'un niveau d'exécution.

Il est possible qu'au sein d'une même trace deux instantanés à des niveaux d'exécution différents soient identiques. De même plusieurs traces différentes peuvent converger vers les mêmes instantanés.

Prenons les quatre enchaînements de niveaux d'exécution et leurs instantanés respectifs suivants : $T_1 = s_1, s_2, s_1, s_5, s_6$; $T_2 = s_1, s_2, s_3, s_5, s_6$; $T_3 = s_1, s_2, s_3, s_4$; $T_4 = s_1, s_2, s_4$. On peut représenter ces enchaînements sous la forme d'un graphe acyclique (figure 6.4) ayant une racine qui est l'instantané du premier niveau d'exécution : il s'agit toujours de la représentation initiale du binaire.

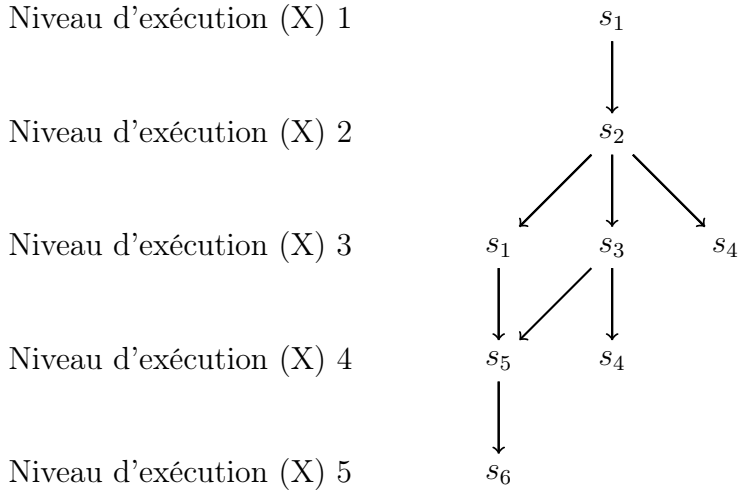


FIGURE 6.4 – Différents instantanés des traces d'exécution et leur niveau d'exécution

Nous considérons que mélanger dans le GFC les instructions atteignables à des niveaux d'exécution différents ne permet pas au GFC d'être une bonne représentation des exécutions possibles puisqu'on perd la notion d'enchaînement des vagues. Ainsi chaque sommet du graphe est caractérisé par le niveau d'exécution de l'instruction, l'instantané dans lequel elle a été analysée, l'adresse de l'instruction et enfin l'instruction. Cette représentation permet d'isoler chaque couple (niveau d'exécution, instantané) dans une sous partie du GFC qui sera le désassemblage parfait, non auto-modifiant, de l'instantané, c'est à dire la vague.

Un exemple de cette représentation est donné en figure 6.5 où le point d'entrée est coloré en orange : il est présent au niveau d'exécution 1 dans l'instantané s_1 tandis que les points de sortie sont colorés en bleu clair. Nous reprenons l'enchaînement des instantanés présenté à la figure 6.4 : chaque encadré regroupe des instructions ayant le même niveau d'exécution X et le même instantané s.

Nous appelons exécution le résultat de l'analyse dynamique définie au chapitre 4 via l'algorithme 4.3 : à partir d'un programme et d'une entrée I, l'analyse dynamique fournit le couple (T, S) où T est une trace et S une liste d'instantanés d'exécution, un pour

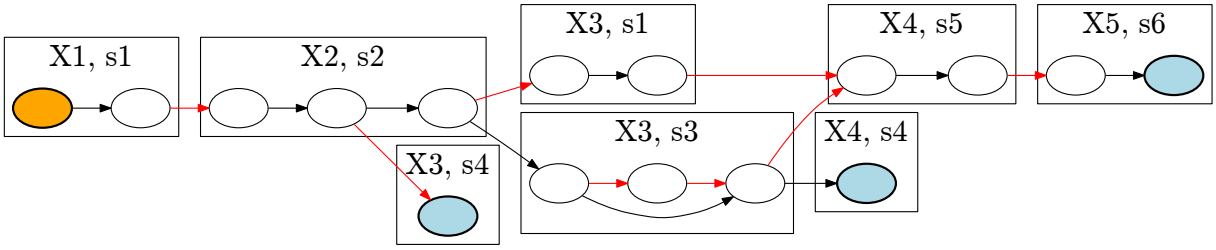


FIGURE 6.5 – GFC parfait pour un programme auto-modifiant

chaque niveau d'exécution. L'ensemble des exécution contient tous ces couples quelle que soit l'entrée (définition 6.3).

Définition 6.3. *Soit un programme P prenant une entrée et ayant pour point d'entrée ep. On note E_X l'ensemble des exécutions de P :* $E_X = \bigcup_I \{$`exécution(P, I, ep)`$\}$.

Formellement le GFC parfait (définition 6.4) est constitué de toutes les instructions présentes dans une trace, représentées par leur niveau d'exécution, leur instantané, leur adresse et l'instruction ; deux sommets sont liés si et seulement si ils se suivent dans une trace d'exécution.

Définition 6.4. *Étant donné un programme P auto-modifiant et E_X l'ensemble des exécutions de P. Nous appelons GFC parfait le graphe $G = (V, E)$ tel que :*

- $V = \{(X,\ S[X],\ \mathcal{A}[D],\ \mathcal{I}[D]), (T, S) \in E_X, (i, X, D) \in T\}$.
- $((X_1, S[X_1], a_1, I_1), (X_2, S[X_2], a_2, I_2)) \in E$ *si et seulement si* $(X_1, S[X_1], a_1, I_1)$ *et* $(X_2, S[X_2], a_2, I_2)$ *se suivent dans une trace :* $\exists (T, S) \in E_X,\ i \in \mathbb{N}, (i, X_1, D_1)$ *et* $(i + 1, X_2, D_2) \in T$ *avec* $\mathcal{A}[D_1]= a_1$, $\mathcal{I}[D_1]= I_1$, $\mathcal{A}[D_2]= a_2$ *et* $\mathcal{I}[D_2]= I_2$.

Il est à noter que les instructions représentées dans le graphe de flot de contrôle parfait sont exactement les mêmes que celles présentes dans le graphe de flot simple de la définition 6.2. La différence tient dans l'agencement des sommets dans le graphe : un sommet du GFC parfait prenant en compte le niveau d'exécution, une instruction dynamique peut être présente dans plusieurs sommets différents si elle est exécutée à différents niveaux d'exécution.

6.1.3 Revue de littérature

Anckaert, Madou et Bosschere [AMB06] proposent d'introduire la représentation sous forme d'octets de chaque instruction au sein du graphe de flot de contrôle et d'ajouter aux arcs les conditions sur les octets en mémoire. Sur l'exemple précédent, leur GFC, présenté en figure 6.6, est identique au GFC simple mais il est alors possible de différencier les chemins selon l'état de la mémoire et est donc plus précis pour une analyse automatique. En fait le graphe qu'ils représentent est un automate, permettant de compléter la notion d'automate de flot de contrôle introduit comme extension au graphe de flot de contrôle [Hen+02] pour des programmes auto-modifiants.

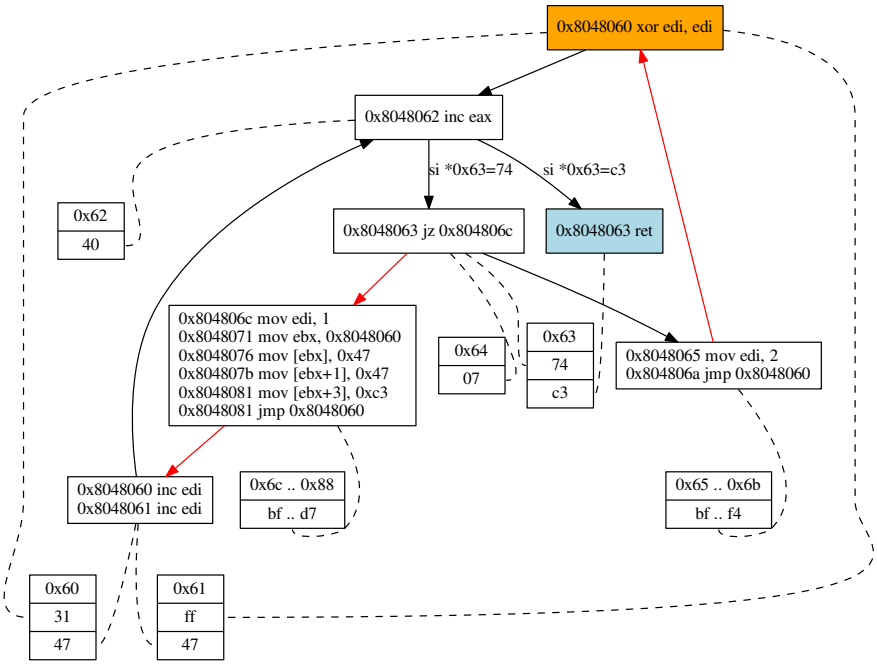


FIGURE 6.6 – GFC simple augmenté des octets et des conditions de transition

6.2 Graphe de flot de contrôle paramétré par une exécution

Nous proposons une représentation du GFC basée sur une exécution particulière et son découpage en différents niveaux d'exécution. Cette représentation nous donne une sous-approximation du graphe de flot parfait présenté en section 6.1.2. L'exécution choisie, dite de référence, consiste en une trace et un enchaînement d'instantanés de chaque niveau d'exécution.

Nous appelons GFC paramétré par cette exécution le GFC parfait du même programme dont l'ensemble des exécutions serait réduit à cette seule exécution choisie.

Le GFC paramétré (définition 6.5) représente chaque instruction contenue dans la trace comme un sommet en prenant en compte son niveau d'exécution, son adresse et l'instruction assembleur. Deux sommets sont reliés par un arc si, dans la trace, leurs deux instructions se suivent immédiatement. Nous ne caractérisons plus, comme dans le graphe de flot de contrôle parfait, les sommets par leur instantané parce que nous ne prenons en compte qu'une exécution et il n'y a alors pas d'ambiguïté : à chaque niveau d'exécution correspond un unique instantané.

Définition 6.5. *Étant donné un programme P auto-modifiant et (T, S) une exécution de P. Nous appelons GFC paramétré par T le graphe $G = (V, E)$ tel que :*

— $V = \{(X, \mathcal{A}[D], \mathcal{I}[D]), (i, X, D) \in T\}$.

— $((X_1, a_1, I_1), (X_2, a_2, I_2)) \in E$ *si et seulement si* (X_1, a_1, I_1) *et* (X_2, a_2, I_2) *se suivent dans la trace :* $\exists i \in \mathbb{N}, (i, X_1, D_1)$ *et* $(i + 1, X_2, D_2) \in T$ *avec* $\mathcal{A}[D_1]= a_1$, $\mathcal{I}[D_1]= I_1$, $\mathcal{A}[D_2]= a_2$, $\mathcal{I}[D_2]= I_2$.

Un tel GFC pour le programme auto-modifiant précédent est donné en figure 6.7 : l'exécution prise en compte est celle démarrant avec `eax`=-2. La boucle directe est exécutée une fois, puis comme `eax`=0, la partie auto-modifiante est activée. Dans l'exemple choisi ici, toutes les exécutions partagent le même découpage en vagues et le GFC est séparé en deux GFC partiels (définition 6.6), un pour chaque niveau d'exécution. Le GFC partiel paramétré à un niveau d'exécution k est une approximation de la vague au niveau k, définie comme le GFC parfait du programme représenté par l'instantané et considéré comme non auto-modifiant(définition 4.7).

Définition 6.6. *On appelle GFC partiel paramétré par une exécution (T, S) au niveau d'exécution X le GFC paramétré par T restreint aux sommets dont le niveau d'exécution est X.*

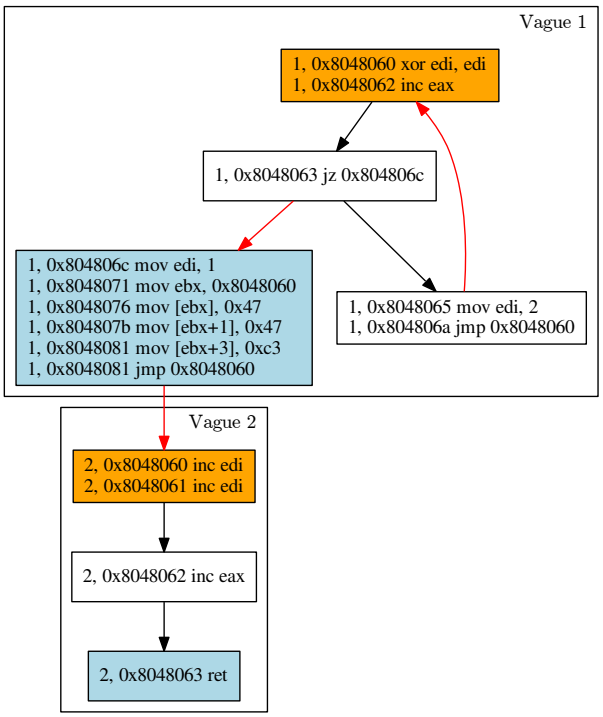


FIGURE 6.7 – GFC paramétré par l'exécution démarrant avec `eax`=-2

L'idée consiste donc à obtenir une exécution particulière d'un programme puis à augmenter la couverture de code à l'aide d'une analyse statique sur chaque instantané de l'exécution.

6.3 Approche hybride

6.3.1 Architecture générale

Notre objectif est donc de construire un GFC paramétré par une exécution représentative du programme à analyser. L'analyse dynamique du programme est faite selon l'algorithme 4.3 du chapitre 4 et fournit une trace d'exécution générale et un découpage de l'exécution en plusieurs instantanés de la mémoire. L'architecture générale du désassembleur est donnée par la figure 6.8.

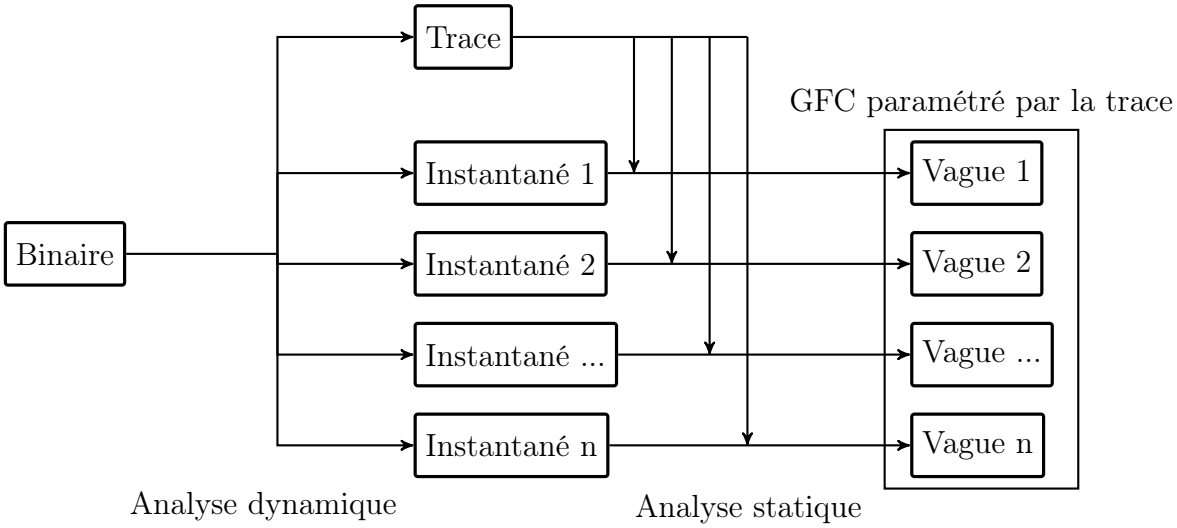


FIGURE 6.8 – Architecture générale du désassembleur hybride

Pertinence de l'approche. L'exécution à partir de laquelle on cherche à reconstruire le GFC se doit d'être représentative d'une exécution standard du programme sur une machine cible. Dans le cas des programmes malveillants on tente donc de se faire passer pour une machine ciblée par l'attaquant pour laisser le programme mener son attaque. À part les informations provenant du système d'exploitation, un programme malveillant ne prend en général pas d'entrée et ne nécessite pas d'interaction avec l'utilisateur. Une seule exécution devrait donc permettre d'analyser son comportement. D'autre part la plupart des logiciels malveillants sont compilés sans comportement auto-modifiant et sont ensuite empaquetés. C'est le binaire empaqueté qui est obscurci et auto-modifiant. Dans ce cas il est fréquent [Cal13] que la charge utile, c'est à dire le binaire d'origine, ne soit activé qu'à la dernière vague de l'exécution. Par conséquent c'est principalement le logiciel de protection qui peut empêcher l'exécution de référence d'être représentative en détectant

que le programme est instrumenté, débogué ou émulé et en détournant l'exécution dans ce cas.

Dans beaucoup de cas le programme de protection essaie de détecter qu'une analyse est en cours et oriente le programme vers un arrêt n'exécutant pas la charge finale. La difficulté est donc d'obtenir une des deux exécutions activant la charge finale présente à l'instantané s_6, comme illustré sur la figure 6.9 reprenant l'arbre des traces possibles présenté lors de la définition du désassemblage parfait (section 6.1.2) : les deux points de sortie présents dans l'instantané s_4 ne sont activés que parce que la protection a détecté que le programme était analysé et celui-ci n'a donc pas activé la charge finale.

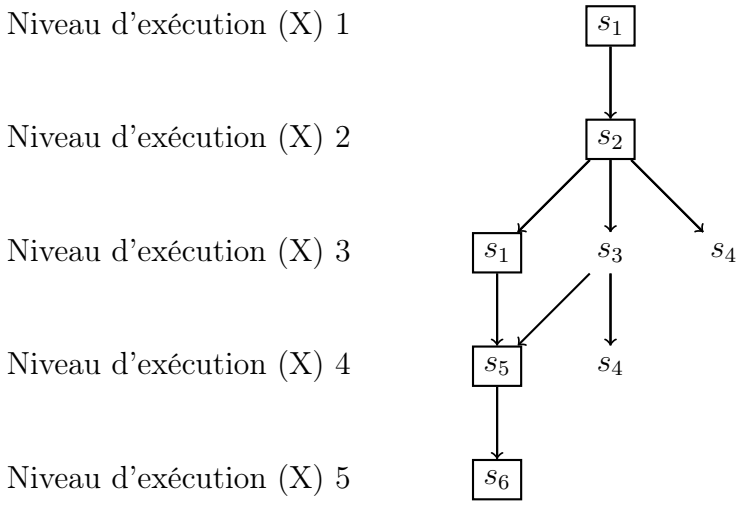


FIGURE 6.9 – Différents instantanés des traces d'exécution et trace privilégiée

6.3.2 Analyse statique de chaque instantané

Pour l'analyse statique nous disposons de la trace et des instantanés munis de leurs points d'entrée et sortie respectifs. Nous cherchons, à partir de la trace et d'un instantané, à reconstruire la vague correspondante, c'est à dire le désassemblage parfait de l'instantané. Nous appelons trace partielle la liste des éléments de la trace qui sont exécutés au même niveau d'exécution que l'instantané étudié. Nous utilisons la trace partielle comme guide dans l'analyse de l'instantané. Toute instruction présente dans la trace est nécessairement du code et doit être incluse dans la vague et le GFC.

Reconstruction du GFC par parcours récursif. Nous effectuons un désassemblage par parcours récursif de l'instantané à partir de chaque instruction présente dans la trace partielle. Dans le GFC partiel reconstitué, les instructions présentes dans la trace sont en rose, les autres en blanc. Le point d'entrée et le point de sortie sont colorés en orange et bleu clair respectivement. Si deux instructions se suivent dans la trace alors l'arc dirigé les reliant provient de l'analyse dynamique et est plein. Si elles se suivent dans l'analyse récursive alors l'arc dirigé provient de l'analyse statique et est en pointillés. Un arc parcouru

par l'analyse dynamique et statique est en gras. Les arcs en noir relient une instruction et l'instruction séquentielle suivante tandis que les arcs en noir représentent le saut d'un appel ou d'un saut (`call` ou `jmp` par exemple).

Point de sortie. La dernière instruction présente dans la trace partielle constitue le point de sortie de l'instantané. Dans le cas où cette instruction n'est exécutée qu'une seule fois dans la trace partielle, c'est à dire que le changement de niveau d'exécution intervient dès qu'elle est exécutée la première fois, nous stoppons le parcours du programme après cette instruction en considérant que les instructions suivantes doivent être analysées dans l'instantané suivant.

Chemins invalides. Si tous les chemins d'exécution passant par une instruction D aboutissent nécessairement sur une instruction provoquant une erreur, soit parce que son adresse n'est pas valide, soit parce qu'elle n'est pas une adresse x86 valide, alors l'instruction D n'est pas valide non plus. Nous n'incluons donc pas ces instructions dans le GFC. Les instructions dont les chemins aboutissent à un saut dont les cibles ne sont pas connues ne sont pas concernées puisqu'il est possible qu'une de ces cibles soit valide.

Mesure du chevauchement à l'aide des couches. Nous utilisons le découpage cohérent en couches adapté au parcours récursif défini à la section 5.2.4 du chapitre 5. Dans le GFC deux instructions qui se chevauchent sont liées par un arc noir non dirigé tracé en pointillés.

Construction des blocs de base. Dans les exemples de GFC déjà vus précédemment, nous avons regroupé certaines instructions dans le même bloc, dit bloc de base, lorsque le regroupement aide à la lisibilité et ne fait pas perdre d'informations. Nous regrouperons deux instructions I_1 et I_2 du GFC si et seulement si :

- Elles se suivent séquentiellement : elles sont reliées par un arc dirigé de I_1 vers I_2 de couleur noire, et
- I_1 n'a pas d'autres arcs sortant que celui aboutissant sur I_2, et
- I_2 n'a pas d'autres arcs entrant que celui provenant de I_1, et
- I_1 et I_2 ne sont pas en chevauchement avec d'autres instructions.

6.4 Revue de littérature

6.4.1 Analyse dynamique

Une application directe de l'analyse des GFC faite dans ce chapitre est la reconstitution de l'original d'un binaire empaqueté. Renovo [KPY07] et PolyUnpack [Roy+07] implémentent de tels systèmes d'extraction qui fonctionnent par analyse du code généré dynamiquement en utilisant des notions similaires à nos instantanés d'exécution. Ils considèrent que le binaire d'origine est celui disponible dans le dernier instantané qui ne sera plus modifié dans la suite de l'exécution. Leur objectif étant de reconstruire le programme

d'origine, dépaqueté, ces deux approches s'arrêtent à cette étape et n'effectuent pas d'analyse supplémentaire.

Pour l'évaluation les auteurs de PolyUnpack utilisent des programmes malveillants qu'ils analysent avec des antivirus existants (ClamAV et McAfee) et montrent que pour certains échantillons les programmes d'origine (empaquetés) ne sont pas détectés par la solution antivirale tandis que les programmes reconstruits par leur outil sont détectés.

Les auteurs de Renovo utilisent des empaqueteurs pour protéger un binaire légitime, `notepad.exe`. Ils analysent ensuite les binaires protégés et montrent qu'ils arrivent à récupérer le programme d'analyse en comparant la section `.text` du `notepad.exe` d'origine avec celle du programme reconstruit.

6.4.2 Interprétation abstraite et analyse hybride

Nous présentons ici des approches qui pourraient remplacer l'analyse statique simple par parcours récursif effectuée dans le cadre de ce chapitre sur les instantanés à l'aide de la trace d'exécution. Cependant elles ne sont pas aptes à remplacer l'analyse dynamique puisqu'elles sont effectuées sur des programmes non auto-modifiants.

L'analyse statique est souvent réalisée à l'aide d'une interprétation abstraite [CC77], méthode basée sur la définition d'une sémantique permettant d'obtenir une sur-approximation des actions de chaque instruction et donc de travailler sur une sur-approximation du GFC. Kinder et Kravchenko [KK12] proposent une technique basée sur une sémantique concrète et une sémantique de sur-approximation, alternant entre l'une et l'autre pour obtenir des résultats plus précis : il s'agit donc aussi en une sens d'une approche hybride.

Bardin, Herrmann et Védrine [BHV11] proposent une méthode d'interprétation abstraite permettant de définir des niveaux de précision souhaités pour certaines variables, en particulier pour les instructions de saut dynamique. Une première analyse trouve des approximations grossières sur les valeurs, puis les variables dont on cherche à connaître précisément la valeur sont à nouveau analysées, jusqu'à obtenir le niveau de précision suffisant (ou échouer). Cette approche a montré son efficacité pour déterminer les cibles de sauts dynamiques et donc pour reconstruire les GFC.

6.5 Implémentation

Analyse dynamique. Nous avons implémenté l'analyse dynamique avec l'outil d'instrumentation d'Intel, Pin, comme expliqué au chapitre 4 traitant de l'auto-modification. L'implémentation permet de suivre les instructions exécutées dans la mémoire du processus instrumenté et gère les allocations dynamiques. Chaque instantané est enregistré sous la forme d'un fichier exécutable (PE sous Windows) dans lequel les sections ont été mises à jour et des sections supplémentaires sont ajoutées dans le cas où le programme utilise des allocations dynamiques.

Désassembleur. Le désassembleur fonctionnant par analyse statique et récursive a été codé en 3500 lignes de C++. Il utilise l'outil XED d'Intel [Intb] fournissant les informations

sémantiques suffisantes, comme détaillé au chapitre 3.

6.6 Expérience sur hostname.exe empaqueté

Afin de valider notre approche nous avons testé notre désassembleur sur une version protégée d'un binaire existant. Cette expérience permet de montrer que l'on arrive a récupérer les parties utiles du programme d'origine dans le binaire protégé.

Nous avons utilisé des empaqueteurs existants pour protéger un programme standard présent sous Windows, `hostname.exe`, et avons cherché à analyser le programme protégé ainsi généré avec notre méthode hybride. L'analyse dynamique est réussie si le programme protégé instrumenté présente la même sortie que le programme d'origine, c'est à dire s'il affiche le nom de la machine. Sur les 34 logiciels d'empaquetage que nous avons évalués, 6 produisent un binaire protégé que nous n'avons pas réussi à instrumenter jusqu'à obtenir le nom de la machine. L'échec de l'instrumentation signifie que le logiciel de protection détecte l'utilisation d'une instrumentation et arrête prématurément l'exécution du programme.

Difficultés. Outre les quelques échantillons que nous n'avons pas réussi à instrumenter, l'empaqueteur `pespin` modifie en mémoire l'entête du fichier PE qui le représente. Ce n'était pas une possibilité que nous avions anticipée et puisqu'il ne s'agit pas d'une allocation d'une nouvelle zone mémoire nous ne sommes pas en mesure de prendre en compte cette modification dans les instantanés. Bien qu'il s'agisse d'un problème surmontable en enregistrant à part les entêtes modifiés, nous n'avons à ce jour par implémenté cette solution et donc certaines vagues de `pespin` présentent des erreurs lors du désassemblage.

Métriques. Nous avons analysé chaque niveau d'exécution et reconstruit chaque vague ainsi que le graphe de flot de contrôle paramétré par l'exécution. Nous avons retenu les informations suivantes.

- Le nombre d'instructions dans chaque vague, d'une part exclusivement dans la trace d'exécution, d'autre part en incluant toutes les instructions récupérées par le désassemblage hybride.
- Le nombre d'instructions étant en chevauchement avec une autre instruction, d'une part pour les instructions de la trace, d'autre part pour toutes les instructions désassemblées.
- Le nombre de couches de code, d'une part en ne prenant en compte que les instructions de la trace, d'autre part en prenant toutes les instructions désassemblées, selon l'algorithme 5.2 du chapitre précédent.
- Le nombre de sauts entre des couches de code différentes, d'une part en ne comptant que ceux provenant de la trace, d'autre part en comptant tous les sauts du graphe de flot de contrôle reconstruit par le désassembleur.
- Le part du graphe de flot de contrôle du programme `hostname.exe` d'origine que l'on retrouve dans le GFC partiel de chaque niveau d'exécution : 100% indique que l'intégralité du graphe de flot de contrôle de `hostname.exe` est retrouvé dans

le graphe de flot de contrôle partiel. Cette métrique sera formalisée dans la partie suivante de cette thèse, au chapitre 8.

— Le statut : cette colonne indique si l'instrumentation a été un échec (✗), si le désassemblage est un succès complet (✓) ou si le désassemblage a rencontré des erreurs ("partiel").

Les données résultant de cette expérience sont données en annexe A et un extrait donnant les résultats sur le fichier d'origine, celui protégé par tElock99 et celui protégé par UPX est donné en figure 6.10. Le premier binaire détaillé est `hostname.exe` non protégé. Pour chaque binaire étudié la première ligne donne des informations générales sur le désassemblage de ce binaire : nombre total d'instructions (caractérisées par leur adresse et leur instruction assembleur) différentes dans la trace et dans le désassemblage hybride ; et la part du GFC de `hostname.exe` que l'on retrouve dans le GFC paramétré.

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Part de hostname détectée
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	
(aucun)	/	154	354	/	/	/	/	/	/	/
	0	154	354	0	0	1	1	0	0	/
tElock99	/	2044	3022	/	/	/	/	/	/	100%
	0	46	130	0	42	1	3	0	16	0%
	1	62	106	0	7	1	2	0	2	0%
	2	19	20	2	2	2	2	2	2	0%
	3	25	25	8	8	2	2	8	8	0%
	4	402	624	0	173	1	3	0	73	0%
	5	15	15	0	0	1	1	0	0	0%
	6	15	15	0	0	1	1	0	0	0%
	7	13	13	0	0	1	1	0	0	0%
	8	16	16	0	0	1	1	0	0	0%
	9	18	34	0	2	1	2	0	2	0%
	10	15	15	0	0	1	1	0	0	0%
	11	15	16	0	0	1	1	0	0	0%
	12	76	80	0	5	1	2	0	2	0%
	13	196	283	0	6	1	2	0	2	0%
	14	183	401	0	37	1	2	0	17	0%
	15	564	727	0	16	1	2	0	2	0%
	16	138	178	0	0	1	1	0	0	0%
	17	226	430	0	0	1	1	0	0	100%
upx	/	322	523	/	/	/	/	/	/	100%
	0	168	169	4	4	2	2	2	2	0%
	1	154	354	0	0	1	1	0	0	100%

FIGURE 6.10 – Extrait du résultat du désassemblage de `hostname.exe` protégé

Vagues et binaire d'origine. Nos expériences confirment un fait étudié précédemment [Cal13 ; KPY07 ; Roy+07] indiquant que la charge utile, cachée par l'empaqueteur, se

trouve dans la dernière vague. Nous montrons que tous les cas de détection massive de `hostname.exe` au sein d'un binaire protégé se font dans la dernière vague. De plus nous validons également l'utilisation systématique de l'auto-modification comme technique de protection : le seul empaqueteur ne présentant pas d'auto-modification est vmprotect qui utilise une technnique d'émulation de code et n'exécute pas directement le code d'origine sur la machine.

On peut distinguer deux cas différents de protection. UPX, comme d'autres empaqueteurs simples, présente le programme d'origine dans son intégralité et non modifié en dernière vague : les caractéristiques de la dernière vague d'UPX sont exactement les mêmes que celle de l'unique vague de `hostname.exe`. L'empaqueteur tElock99 présente le code de `hostname.exe` également en dernière vague mais il est mêlé à du code supplémentaire : la dernière vague est également obscurcie.

Utilisation du chevauchement de code et couches. De nombreux binaires protégés (22 sur 28) présentent, dans le graphe que nous reconstruisons, des chevauchements de code et donc plus d'une couche de code. Pourtant seuls 9 de ces binaires ont utilisé des instructions en chevauchement lors de leur exécution : les autres binaires utilisent en fait des techniques d'obscurcissement qui conduisent notre désassembleur à considérer des instructions se chevauchant sans en faire un usage actif.

Ainsi seules les variantes de tElock, UPX et pespin utilisent réellement le chevauchement de code et ils l'utilisent avec parcimonie : au plus une douzaine d'instructions sont en chevauchement dans la trace (sur plusieurs centaines) et il n'y a au plus que deux couches de code dans la trace.

Nous concluons que peu d'empaqueteurs utilisent le chevauchement de code lors de leur exécution mais que la plupart obscurcissent le code de manière à amener le désassembleur à considérer des instructions en chevauchement.

6.7 Conclusion

Nous avons combiné les notions d'analyse statique et d'analyse dynamique vues aux chapitres précédents pour réaliser et implémenter un désassembleur capable de construire un graphe de flot de contrôle approchant le graphe de flot parfait d'un programme auto-modifiant, notion que nous avons définie.

L'expérience réalisée sur des versions protégées de `hostname.exe` montre que nous sommes capables d'identifier le binaire d'origine au sein des versions protégées dans de nombreux cas. Nous constatons une utilisation systématique de l'auto-modification et un emploi fréquent du chevauchement de code pour gêner le travail du désassembleur bien que peu de programmes de protection exécutent réellement ces instructions en chevauchement.

Deuxième partie

Détection et analyse morphologique

Chapitre 7

Détection des logiciels malveillants

Le second objet d'étude de cette thèse est une technique de détection de logiciels malveillants fonctionnant par comparaison de leurs graphes de flot de contrôle (GFC).

Cette partie est organisée comme suit. Nous présentons en premier lieu différentes approches de la détection de programmes malveillants, puis nous détaillons l'analyse morphologique fonctionnant par comparaison des graphes de flot de contrôle et expliquons comment optimiser cette approche. Enfin nous donnons une application de l'analyse morphologique à la détection de similarités logicielles et des exemples d'études de codes malveillants basées sur cette approche.

Dans ce chapitre nous présentons plusieurs techniques de détection de programmes malveillants et introduisons l'analyse morphologique.

7.1 Contexte de détection

Nous rappelons la définition d'un programme malveillant, donnée en introduction (définition 7.1).

Définition 7.1. *Un logiciel est dit malveillant s'il réalise volontairement des opérations allant à l'encontre de l'intérêt de l'utilisateur, et ce à l'insu de celui-ci.*

La définition varie en fait d'un utilisateur à un autre et une technique plus formelle permettant de classifier un programme fait intervenir la spécification d'une politique de sécurité. Cette politique de sécurité décrit les différents flots de données autorisés et ceux étant interdits. Par exemple elle pourrait interdire de lire le fichier /etc/shadow contenant des informations relatives aux mots de passe d'un ordinateur sous Linux et d'exfiltrer ces informations hors de la machine (via le réseau ou un support amovible).

Il n'existe pas de programme capable d'analyser tout programme binaire pour connaître a priori leur comportement de manière exacte : on ne peut pas construire un analyseur déterminant si les programmes analysés lisent /etc/shadow. La raison est que le problème du désassemblage est déjà indécidable ; celui de la détection l'est également.

Puisqu'il n'existe pas de méthode de détection parfaite mais que l'on est en pratique capable d'analyser des programmes à la main, nous disposons d'un corpus de programmes

malveillants d'une part et d'un corpus de programmes légitimes d'autre part. Certaines méthodes mesurent une distance entre le programme analysé et les programmes malveillants connus, puis entre le programme analysé et les programmes légitimes connus. Si le programme analysé est suffisamment proche d'un programme malveillant il sera considéré comme malveillant. S'il est identique ou très proche d'un programme légitime, il pourra être classifié comme légitime.

Programmes obscurcis et auto-modifiants. La plupart des programmes malveillants utilisent de l'obscurcissement et sont auto-modifiants. De plus certains programmes légitimes sont également auto-modifiants, tels ceux permettant la compilation à la volée. De nombreux éditeurs de logiciels non libres, cherchant à empêcher ou ralentir l'analyse de leur code afin de protéger leur propriété intellectuelle, utilisent le même arsenal de protection que les auteurs de logiciels malveillants.

Critères de classification. La classification en logiciel légitime ou malveillant peut s'effectuer sur divers critères, que l'on compare à ceux de programmes ou de comportements connus :

— sa représentation sous forme de fichier,
— des traces d'exécution,
— son code assembleur désassemblé,
— le code assembleur du programme dont on aurait enlevé les protections,
— son graphe de flot de contrôle (GFC).

Ces caractéristiques et les techniques permettant de les déterminer ont été étudiées dans la première partie de ce document. Nous considérons que leur extraction ne fait pas à proprement partie de la détection : il s'agit de données à partir desquelles nous pouvons construire des signatures de programmes malveillants ou des comportements de référence.

Dans le cas d'une détection par signatures, les corpus de logiciels malveillants et légitimes contiennent des programmes dont ont préalablement été extraits les critères précédents et à partir desquels des règles de détection ont été décidées. Le schéma général de détection par signatures est donné en figure 7.1.

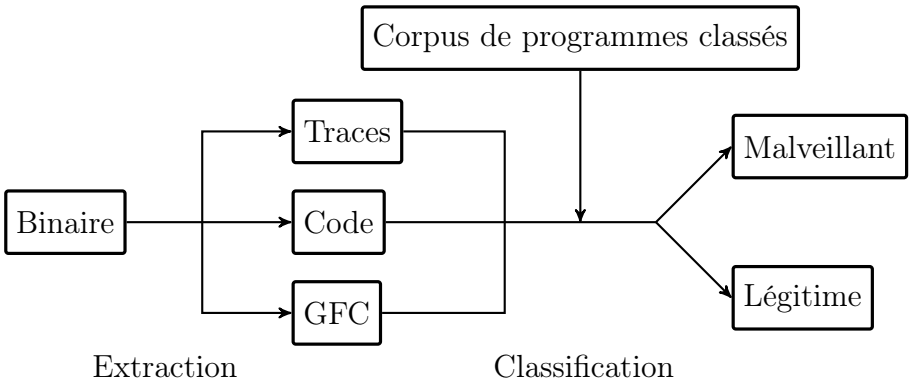


FIGURE 7.1 – Architecture générique d'un détecteur par signatures

Une détection comportementale contient également une liste de comportements connus servant à classifier un programme analysé.

7.2 État de l'art

Approche syntaxique par expressions régulières. L'approche traditionnelle de la classification consiste en l'extraction, pour chaque logiciel malveillant du corpus, d'une chaîne de caractères présente dans sa représentation sous forme de fichier [Szo05]. Elle peut être complétée dans le but de gérer, pour chaque logiciel du corpus, une expression régulière plutôt qu'une chaîne exacte. Cette approche, purement syntaxique, nécessite qu'un analyste choisisse l'expression régulière à prendre en compte pour chaque logiciel du corpus : il s'agit en général d'une chaîne de données caractéristique (un numéro de version, une clé de chiffrement, un nom, etc.) ou de la représentation en octets de quelques instructions représentatives du programme. Son avantage est que, mise en œuvre correctement, elle génère très peu de faux positifs. Son premier inconvénient majeur est qu'elle est très sensible à une légère modification du programme malveillant : il suffit souvent de rajouter ou de réorganiser le code pour empêcher la détection [CJ04]. Le second inconvénient de cette approche est qu'elle nécessite une analyse manuelle pour extraire la signature, opération d'autant plus contraignante si de nombreuses variantes du programme existent et qu'une nouvelle signature doit être générée pour chaque variante.

Approches comportementales. Les méthodes de détection comportementales cherchent à connaître le comportement d'un programme et à le comparer, soit à une liste de comportements connus comme légitimes, soit à des comportements classés comme malveillants [JDF08]. Il s'agit en général dans un premier temps de définir un certain nombre de comportements de haut niveau, comme l'utilisation de fichiers, de flux réseau ou de certaines fonctionnalités sensibles du système d'exploitation. Ensuite des scénarios de détection sont définis : il peut s'agir d'un enchaînement d'actions (lecture d'un fichier puis ouverture d'une connexion distante) ou simplement la définition d'un seuil (au delà d'un certain nombre d'actions sensibles effectuées). La réalisation d'un de ces scénarios par le programme analysé provoque sa classification comme logiciel malveillant. Certaines implémentations ont montré l'efficacité de l'approche, comme sur l'analyse d'extensions malveillantes pour Internet Explorer [Kir+06].

Mesure automatique de distance. Une technique générique de détection par signatures consiste à chercher des similarités entre deux programmes en définissant par exemple une distance entre eux. Ensuite la classification se fait par mesure de la distance entre le programme analysé et les différents programmes du corpus. Une possibilité est de décomposer le code des programmes en n-grammes contenant n instructions séquentielles d'un même bloc de base. La signature du programme est alors l'ensemble de ces groupes de n instructions. Pour rendre les instructions plus génériques, Upchurch et Zhou [UZ13] ne prennent pas les instructions entières mais seulement leur mnémonique (`mov eax, 3` devient `mov`).

Familles de logiciels malveillants. Jang, Brumley et Venkataraman [JBV11] proposent un outil générique de classification des logiciels en familles, quelles que soient les caractéristiques et donc les mesures choisies. Ce type d'approches permet d'aller plus loin que la simple classification en logiciel légitime ou malveillant et peut donner des informations sur ce qu'on peut attendre d'un logiciel en fonction de la famille à laquelle il appartient.

7.3 Analyse morphologique

L'analyse morphologique, introduite par Kaczmarek [Kac08], Bonfante et Marion [BKM09], propose une détection basée sur la comparaison des graphes de flot de contrôle. L'idée est de comparer la forme des graphes de flot de contrôle plutôt que les instructions exactes qui y sont présentes. Cette approche fonctionne donc par signatures, les signatures étant des graphes de flot de contrôle, et cherche à mesurer une distance automatique entre un programme et le corpus de programmes malveillants connus. Prenons le programme assembleur défini à la figure 7.2. La structure utile de son graphe de flot de contrôle réside dans les instructions de saut inconditionnel `jne` aux adresses `0x8048068` et `0x8048074`. Ce sont ces deux instructions qui donnent sa forme au GFC : il est constitué de deux chemins principaux dont l'un est constitué d'une boucle.

7.3.1 Comparaison des graphes de flot de contrôle

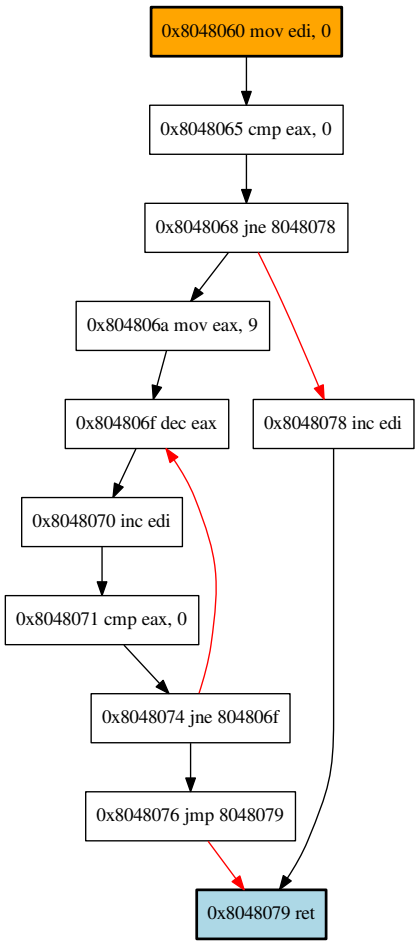
La technique consiste à rechercher des parties du graphe de flot de contrôle de programmes du corpus dans le programme analysé. Si une portion suffisante du graphe de flot de contrôle de programmes malveillants connus est présente dans le programme testé, on peut le classer comme indésirable. Formellement il s'agit de trouver des isomorphismes de sous-graphes entre deux graphes de flot de contrôle. Nous définirons plus précisément le problème et détaillerons plusieurs algorithmes de résolution dans le chapitre suivant. Avant de comparer les GFC, ceux-ci sont mis dans une forme canonique à l'aide de réductions : ces réductions réduisent la taille du graphe et le rendent plus générique.

Simplification des sommets. Les sommets des GFC sont modifiés afin que chaque instruction soit d'un des types génériques suivants : saut inconditionnel (JMP), saut conditionnel (JCC), appel de fonction ($CALL$), retour de fonction (RET), instruction séquentielle ($INST$) ou instruction inconnue ($UNDEF$). De même la distinction entre les arcs reliant deux instructions séquentielles, en noir dans les GFC, et ceux reliant une instruction de saut et la cible du saut, en rouge, est oubliée. Cette transformation permet aux signatures d'être plus génériques.

Réduction des graphes de flot de contrôle. Plusieurs réductions ont été proposées (figure 7.3) afin de résister à certaines formes d'obscurcissement comme l'insertion de code mort, la réorganisation d'instructions séquentielles, l'ajout de sauts inconditionnels ou d'appels inutiles.

Adresse	Octets	Instruction
8048060	bf 00 00 00 00	mov edi, 0
8048065	83 f8 00	cmp eax, 0
8048068	75 0e	jne 8048078 <suite>
804806a	b8 0a 00 00 00	mov eax, 9
<boucle>		
804806f	48	dec eax
8048070	47	inc edi
8048071	83 f8 00	cmp eax, 0
8048074	75 f9	jne 804806f <boucle>
8048076	eb 01	jmp 8048079 <fin>
<suite>		
8048078	47	inc edi
<fin>		
8048079	c3	ret

(a) Code assembleur



(b) Graphe de flot de contrôle

FIGURE 7.2 – Programme et son graphe de flot de contrôle

Les graphes comparés au final sont les graphes sous forme réduite. Le graphe de flot du programme pris en exemple précédemment est rappelé, simplifié et réduit en figure 7.4.

Résistance à l'obscurcissement. L'emploi de réductions a pour but de rendre les signatures plus génériques et moins sensibles à l'obscurcissement. Certains types d'obscurcissement visent spécifiquement le graphe de flot de contrôle pour le rendre moins intelligible. Nous avons présenté une de ces techniques, l'aplatissement de graphe de flot de contrôle lors de la première partie (section 2.2). Nous n'avons pas étudié en détail l'impact de ces méthodes sur la précision de la détection faite par l'analyse morphologique mais nous pensons que des techniques d'obscurcissement spécialement créées pour contrer notre technique de détection seraient efficaces et pourraient mener à un grand nombre de faux positifs ou de faux négatifs.

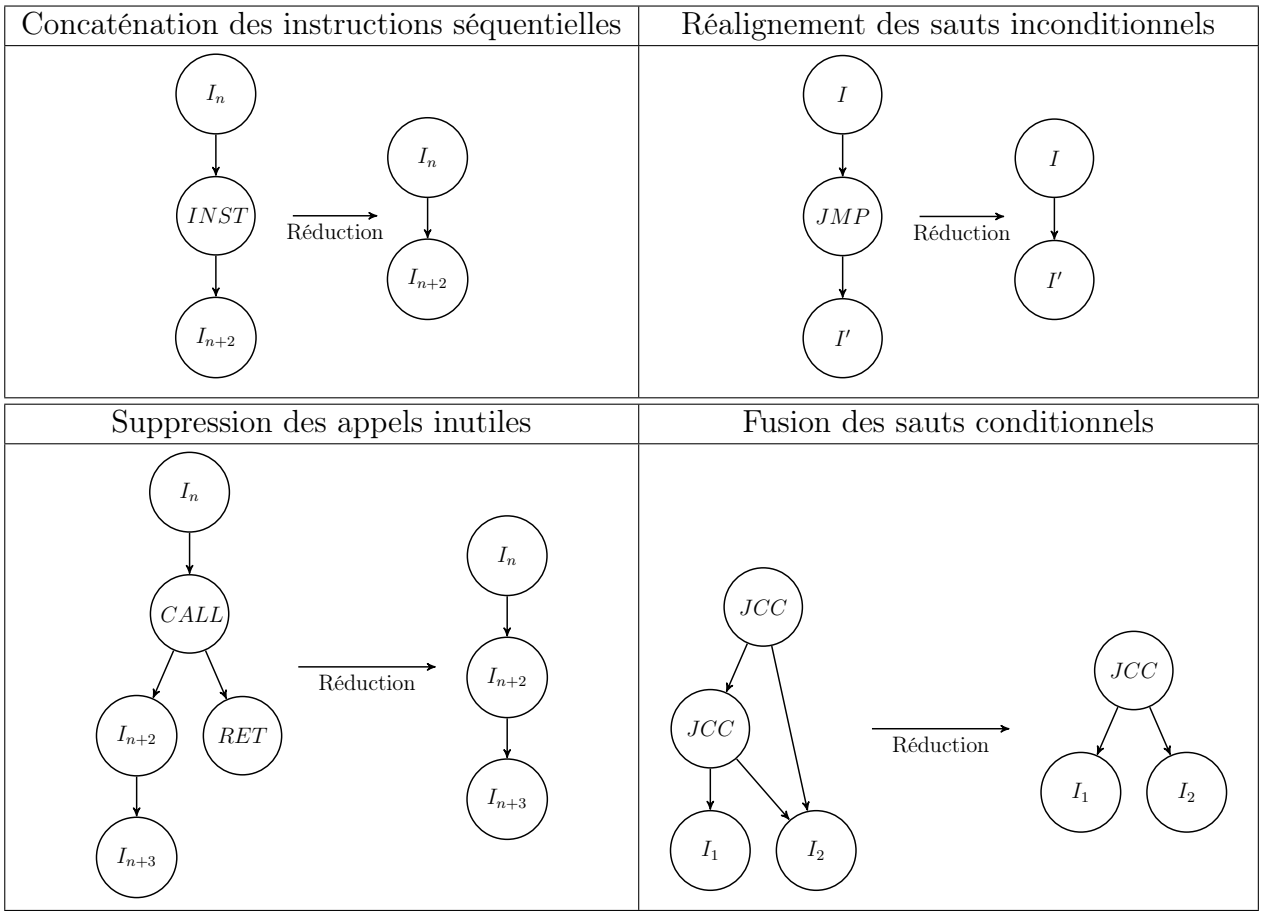


FIGURE 7.3 – Réductions proposées du GFC

7.3.2 Algorithme de comparaison

L'algorithme de comparaison des graphes de flot de contrôle et de détection d'isomorphismes de sous-graphes développé à l'origine pour l'analyse morphologique [BKM09] transforme les GFC en arbres et crée un automate d'arbres dans lequel les programmes du corpus sont ajoutés. Le processus de classification d'un programme à analyser découpe son GFC réduit en sous-graphes de petite taille (typiquement entre 12 et 24 sommets), les transforme en arbres et détermine s'ils sont reconnus ou non par l'automate d'arbres. Un pourcentage de reconnaissance est extrait et selon sa valeur le programme est considéré malveillant ou légitime.

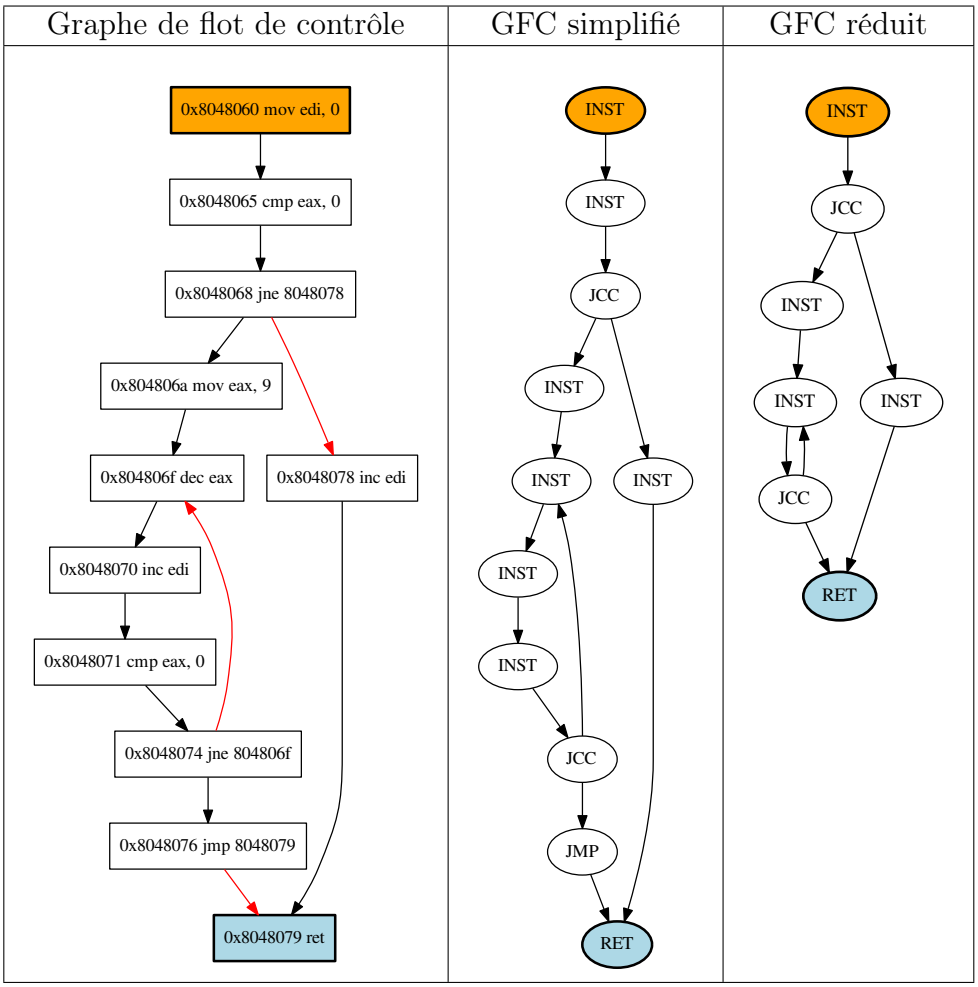


FIGURE 7.4 – Transformations du graphe de flot de contrôle de l'exemple de la figure 7.2

7.3.3 Revue de littérature

Cesare et Xiang [CX10] proposent une approche similaire cherchant également à classer de manière automatique un programme par comparaison de son GFC avec celui des programmes du corpus. Ils détectent les fonctions assembleur en utilisant des heuristiques et créent un GFC pour chacune de ces fonctions. Ces GFC sont ensuite transformés dans un langage intermédiaire pour les simplifier, à l'instar des réductions que nous leur appliquons. Une base de GFC est donc créée et les auteurs formalisent une mesure de distance entre un programme et cette base. La principale différence avec notre approche est donc que les auteurs cherchent à détecter des fonctions dans le programme assembleur tandis que nous construisons un GFC global que nous découpons seulement ensuite.

Zynamics commercialise un outil de comparaison de binaires, BinDiff [Zyna], qui sépare également les binaires utilisés en fonctions assembleur. L'outil permet de trouver des fonctions identiques et de comparer des fonctions présentant certaines similarités [TR05]. Il opère par comparaison de critères sur les fonctions comme leur nom, les appels systèmes qu'elles effectuent ou encore le nombre d'arcs en entrée et en sortie des sommets de leur

GFC partiel. Bien qu'il n'utilise pas à proprement parler de techniques d'isomorphismes de graphes, BinDiff fonctionne sur des graphes de flot de contrôle en utilisant certains critères qui leur sont spécifiques et va donc plus loin qu'une simple comparaison syntaxique des fonctions.

L'application de techniques de réduction dans le but de rendre des codes malveillants plus génériques et ainsi que les signatures soient plus efficaces est fréquente dans la littérature. Certains travaux [Chr+05] proposent des méthodes luttant spécifiquement contre certaines techniques d'obscurcissement comme la réorganisation de code ; c'est l'objectif des réductions proposées à la figure 7.3 dont fait partie le réalignement des sauts inconditionnels. Bruschi, Martignoni et Monga [BMM06] proposent d'utiliser un panel de techniques d'analyse statique existantes pour définir une forme réduite, dite canonique, pour les binaires, puis de comparer les graphes de flot de contrôle résultants.

7.4 Conclusion

Les approches de la détection des logiciels malveillants fonctionnent par comparaison de signatures ou de comportements. Les signatures prennent en compte différentes caractéristiques du binaire tandis que les approches comportementales cherchent à obtenir des informations de plus haut niveau sur les exécutions du programme.

L'analyse morphologique, que nous présenterons en détail dans le prochain chapitre, est une approche de la détection des logiciels malveillants fonctionnant par comparaison de signatures. Les signatures, étant des graphes de flot de contrôle réduits, sont plus génériques que les signatures syntaxiques classiquement utilisées et permettent de détecter des structures similaires dans les différents graphes de flot de contrôle des programmes considérés.

Chapitre 8

Algorithmes de détection de sous-graphes

Nous avons vu au chapitre précédent que l'analyse morphologique fonctionne par comparaison des graphes de flot de contrôle. Dans ce chapitre nous décrivons le fonctionnement du détecteur par analyse morphologique et nous nous intéressons au problème de la détection d'isomorphismes de sous-graphes. Nous définissons ce qu'est un graphe de flot, notion plus spécifique que la notion générale de graphe, au sein duquel les fils d'un sommet sont ordonnés. Cette restriction rend le problème de la recherche d'isomorphisme de sous-graphes de flot solvable en temps polynômial alors qu'il est NP-complet dans le cas général.

8.1 Détection par analyse morphologique

8.1.1 Graphes de flot

Nous avons vu que les graphes de flot de contrôle sur lesquels nous travaillons sont étiquetés (définition 8.1) et, une fois réduits, les étiquettes possibles sont les suivantes : $INST$, JMP, JCC, $CALL$, RET, HLT et $UNDEF$. Le nombre de fils d'un sommet est sa valence (définition 8.2) ; à chaque étiquette est associée une valence maximale. Par exemple les instructions de saut conditionnel JCC ont au maximum deux successeurs. Nous notons k la valence maximale quelle que soit l'étiquette. Les étiquettes ayant le plus de successeurs étant JCC et $CALL$; $k = 2$.

Définition 8.1. *Un graphe orienté et étiqueté $T = (V, L, l, E)$ est composé par les ensembles finis V, L, E, et la fonction totale $l : V \to L$. V est l'ensemble des sommets de T, L un ensemble d'étiquettes, l la fonction donnant l'étiquette d'un sommet et E l'ensemble des arcs de T.*

Définition 8.2. *La valence d'un sommet d'un graphe est le nombre de fils de ce sommet.*

Afin de refléter l'incomplétude des graphes de flot de contrôle reconstruits, les graphes sur lesquels nous sommes amenés à travailler peuvent être partiels : il y a des branches qui peuvent être inconnues. Par exemple un sommet d'étiquette JCC ayant un seul fils est incomplet : la cible du saut est inconnue. Nous forçons les fils de chaque sommet à être ordonnés. Il s'agit d'une contrainte forte qui permet de réduire grandement la complexité des algorithmes mis en œuvre pour résoudre l'isomorphisme de sous-graphe. Nous définissons ainsi formellement un graphe de flot à la définition 8.3. Un exemple de graphe de flot est donné en figure 8.1.

Définition 8.3. *Un graphe de flot est un graphe orienté et étiqueté* (V, L, l, E) *vérifiant :*

1. *À chaque étiquette est associée une arité dans* $\mathbb{N}$ *: il s'agit de la valence maximale pour tous les sommets ayant cette étiquette, et*
   (a) *Si la valence de chaque sommet est égale à l'arité de son étiquette, le graphe de flot est dit complet*
   (b) *Dans le cas contraire, il s'agit d'un graphe de flot incomplet*
2. *Les fils de chaque sommet sont ordonnés : l'ensemble des arcs est* $E \subset V \times \mathbb{N} \times V$*. Les éléments* (t, i, h) *de* E *vérifient :*
   — $1 \leq i \leq a$ *où* a *est l'arité de l'étiquette de* t*, et*
   — *il n'y a pas d'autre fils de* t *numéroté* i *:* $\{(t, i, h') \in E, h' \neq h\} = \emptyset$*.*

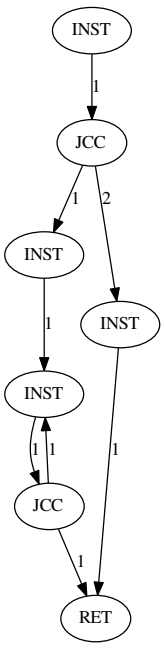


FIGURE 8.1 – Exemple de graphe de flot

8.1.2 Isomorphisme de graphes et de sous-graphes de flot

Deux graphes de flot sont isomorphes s'ils vérifient les propriétés de la définition 8.4, c'est à dire s'il existe une permutation des sommets du premier tel que le graphe modifié soit identique au second.

Définition 8.4. *Soit $T = (V_T, L_T, l_T, E_T)$ et $P = (V_P, L_P, l_P, E_P)$ deux graphes de flot. Ils sont dits isomorphes si et seulement si il existe une bijection $f : V_T \to V_P$ entre les sommets de T et ceux de P telle que f préserve la structure d'adjacence des graphes avec l'ordre des fils et que les étiquettes sont identiques, c'est à dire si et seulement si :*

— $\forall u, v \in V_T$ *et* $i \in \mathbb{N} : (u, i, v) \in E_T \Leftrightarrow (f(u), i, f(v)) \in E_P$, *et*

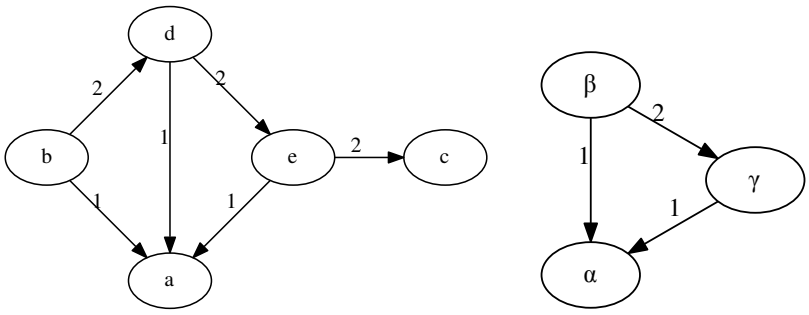
— $\forall u \in V_T, l_T(u) = l_P(f(u))$.

On définit un sous-graphe d'un graphe T selon la définition 8.5 et un sous-graphe de flot induit par un ensemble de sommets en définition 8.6.

Définition 8.5. *Un graphe de flot $P = (V_P, L_P, l_P, E_P)$ est un sous-graphe du graphe de flot $T = (V_T, L_T, l_T, E_T)$ si et seulement si $V_P \subset V_T$, $E_P \subset \{(t, i, h) \in E_T / t, h \in V_P\}$, et $\forall u \in V_P, l_P(u) = l_T(u)$*

Définition 8.6. *Le sous-graphe induit d'un graphe de flot $T = (V_T, L_T, l_T, E_T)$ à partir de l'ensemble de sommets $V_P \subset V_T$ est le graphe de flot $P = (V_P, L_P, l_P, E_P)$ tel que $E_P = \{(t, i, h) \in E_T / t, h \in V_P\}$, $L_P = L_T$, et $l_P = l_{T|V_P}$.*

Notre problème consiste donc, à partir d'un graphe de flot P connu et d'un graphe de flot T à analyser, à déterminer si P est isomorphe à un sous-graphe de flot de T. Par exemple sur la figure 8.2, le graphe P est isomorphe au sous-graphe de T induit par les sommets $\{d, e, a\}$ comme à celui formé induit par les sommets $\{b, d, a\}$.



(a) Graphe de flot T

(b) Graphe de flot P, isomorphe à un sous-graphe de T

FIGURE 8.2 – Exemple de graphe de flot et d'un sous-graphe

8.1.3 Sites

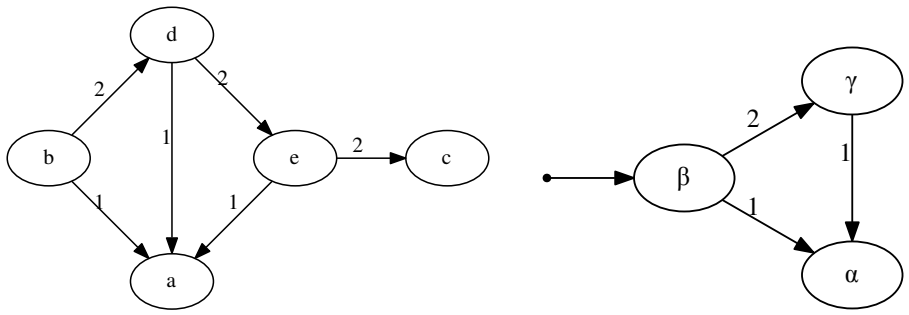
L'analyse morphologique [BKM09] ne cherche pas directement à savoir si un programme malveillant est un sous-graphe du programme analysé ou inversement. Cette approche est vouée à l'échec puisqu'une modification mineure de l'un des deux conduirait à une non détection. Au contraire on va découper les graphes de flot du corpus en sous-graphes tout en gardant le graphe de test T intact. On cherchera alors des isomorphismes de sous-graphes entre chaque petit graphe de la base et un sous-graphe du graphe de flot de test.

En pratique on va générer un certain nombre de sous-graphes, appelés sites, à partir du corpus de graphes de flot qui seront les motifs que l'on cherche à retrouver dans le graphe de flot analysé : on notera généralement P un graphe de motif. Ces sous-graphes sont définis par le sous-graphe induit par les sommets parcourus à l'aide d'un parcours en largeur limité à un certain nombre de sommets, que l'on notera W (typiquement entre 12 et 24), à partir de chaque sommet du graphe d'origine. Ils ont donc une racine qui est le sommet à partir duquel ils ont été générés.

Ainsi un site est un graphe de flot ayant une racine (définition 8.7). Un site peut être un sous-graphe d'un graphe de flot (définition 8.8) comme dans l'exemple de la figure 8.3 : le sous-site est généré à partir du sommet b en construisant le sous-graphe induit par les trois premiers sommets atteignables par un parcours en largeur.

Définition 8.7. *Un site P est un graphe de flot ayant une racine : un sommet à partir duquel on peut atteindre n'importe quel autre sommet de P.*

Définition 8.8. *On appelle sous-site d'un graphe de flot T tout site étant un sous-graphe de T.*



(a) Graphe de flot T

(b) Site P, de racine b, isomorphe à un sous-graphe de T

FIGURE 8.3 – Exemple de graphe de flot et d'un sous-site

Générer les site de manière homogène, toujours à l'aide d'un parcours en largeur de taille fixe, permet d'obtenir automatiquement des signatures pour les programmes du corpus. Nous choisissons le parcours en largeur parce qu'il donne rapidement une structure au site construit tandis qu'un parcours en profondeur donne souvent de longs chemins sans branchements, par exemple. L'algorithme 8.1 détermine, s'il existe, le site

d'un graphe de flot généré par parcours en largeur à partir d'un sommet spécifié et limité à un certain nombre de sommets. L'algorithme 8.2 donne l'ensemble des sites d'un graphe de flot obtenus par parcours en largeur limité à un W sommets.

Algorithme 8.1 : Site obtenu par parcours en largeur limité à partir d'un sommet particulier

Données : Un graphe de flot G, un sommet R de G, un entier W
Résultat : Le site, s'il existe, ayant W sommets construit par parcours en largeur à partir du sommet R de G

```
site(G, R, W)
    E ← ∅                          /* E contient les sommets du futur site */
    M ← ∅                          /* Les sommets visités seront ajoutés à M */
    L ← pile vide                  /* L : file d'attente pour les sommets à visiter */
    L.empiler(R)
    tant que |E| < W et L ≠ ∅ faire
        s ← L.dépiler()
        E ← E ∪ s
        M ← M ∪ s
        pour chaque f fils de s faire
            si f ∉ M alors
                L.empiler(f)
                M ← M ∪ f
            fin
        fin
    fin
    si |E| = W alors
        retourner le sous-site de G induit par les sommets présents dans E, de
        racine R
    sinon
        retourner FAIL
    fin
```

Complexité. Calculons la complexité de l'obtention d'un site par parcours en largeur limité à W sommets en prenant en compte le nombre d'opérations de dépilement ou d'empilement dans la file L. Nous avons vu que chaque sommet a au plus k fils. De plus le parcours s'arrête dès que W sommets ont été ajoutés à l'ensemble E. Il peut arriver que les fils du W^e sommet soient ajoutés à la liste d'attente (mais ils ne seront pas ajoutés à E). Dans le pire des cas on empile donc la racine puis k fils pour W sommets et on dépile W sommets. La complexité de l'algorithme 8.1 est donc $O((k+1).W)$.

L'algorithme 8.2 itère l'opération précédente sur les n_G sommets du graphe G : sa complexité est $O(n_G.(k+1).W)$.

Algorithme 8.2 : Sites obtenus par parcours en largeur limité

Données : Un graphe de flot G, un entier W
Résultat : L'ensemble de sites ayant W sommets construits par parcours en largeur à partir de chaque sommet de G

```
sites(G, W)
    E ← ∅
    pour chaque s sommet de G faire
        S ← site(G, s, W)
        si S ≠ FAIL alors
            E = E ∪ {S}
        fin
        retourner E
    fin
```

8.1.4 Fonctionnement d'un détecteur morphologique

Nous avons à présent tous les éléments nécessaires à une définition précise de la méthode de détection par analyse morphologique.

Constitution des signatures et du corpus

Nous cherchons à construire un corpus contenant les signatures d'un programme malveillant. La signature d'un programme malveillant est un ensemble de sites extraits du graphe de flot de ce programme. Afin que les signatures soient extraites automatiquement nous cherchons donc à construire tous les sites de taille limitée, par parcours en largeur à partir de chaque sommet du graphe de flot. Un exemple de signature du graphe de flot précédent est donné en figure 8.4. Nous avons pris, pour l'exemple, des sites de taille 3, bien qu'en réalité nous prenions des sites de taille $W = 24$.

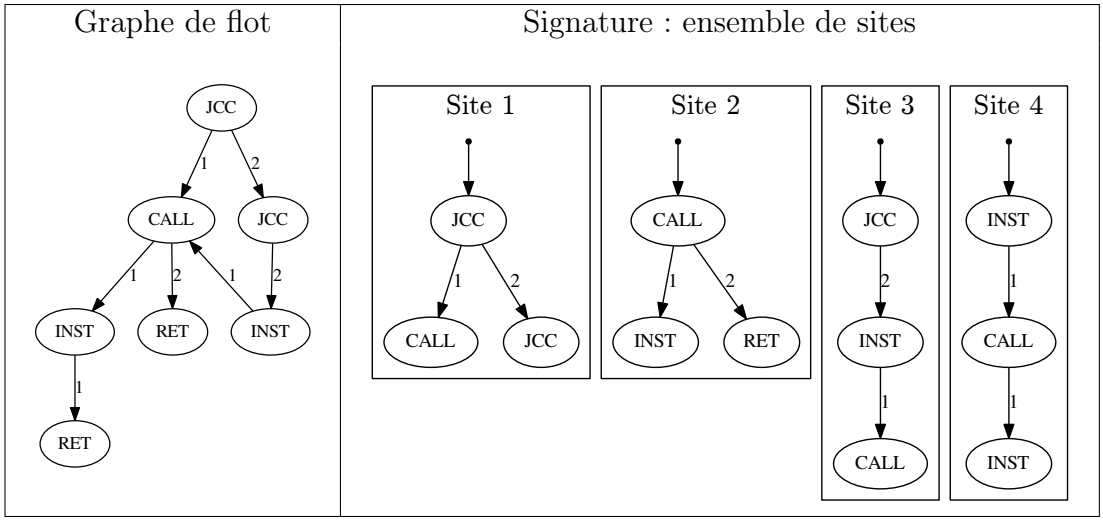


FIGURE 8.4 – Graphe de flot d'un programme et sa signature dans le corpus

La taille des sites considérés influe sur la précision des correspondances que l'on pourra trouver : plus un site a de sommets, plus il est discriminatoire et moins il risque d'être retrouvé dans des graphes de flot ne présentant pas le même comportement. Il s'agit donc d'un compromis entre faux positifs (correspondances sur des graphes indépendants) et faux négatifs (graphes similaires mais non détectés) qui aboutit à une valeur pour W entre 12 et 24, choisie en pratique à 24.

La phase d'apprentissage consiste donc à récupérer une signature sous forme d'un ensemble de sites pour chaque programme du corpus et à regrouper ces signatures dans une base, comme illustré à la figure 8.5.

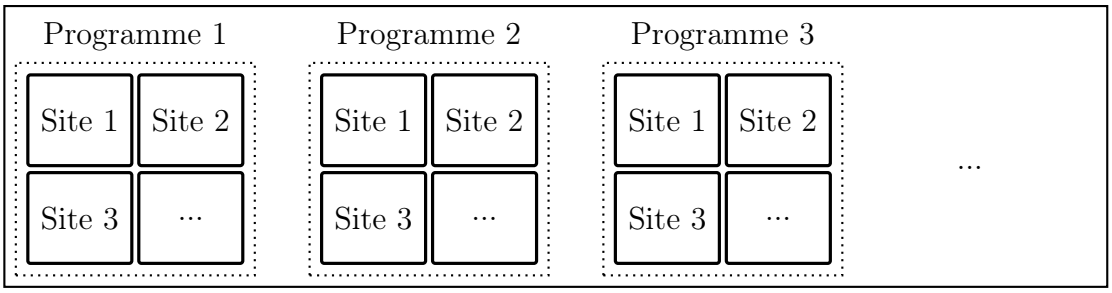


FIGURE 8.5 – Corpus pour l'analyse morphologique

Détection

Supposons que l'on dispose du corpus comprenant la signature du programme précédent et du graphe de flot de test provenant d'un programme à analyser. Le détecteur morphologique cherche à trouver tous les sites du corpus qui sont des sous-sites du graphe de flot de test et, selon les programmes dans lesquels ils étaient présents, à rapprocher le programme analysé de ceux du corpus.

Prenons le programme de test de la figure 8.6. Le site 1 du corpus (figure 8.4) est un sous-site de ce graphe de flot : en partant du premier sommet *JCC*, dans le site comme dans le graphe de flot de test le fils numéroté 1 est un *CALL* et le fils numéroté 2 est un *JCC*. De même les sites 3 et 4 sont des sous-sites du graphe de flot de test. Par contre le site 2 n'est pas un sous-site : aucun sommet d'étiquette *CALL* n'a pour fils numéroté 2 un sommet d'étiquette *RET*.

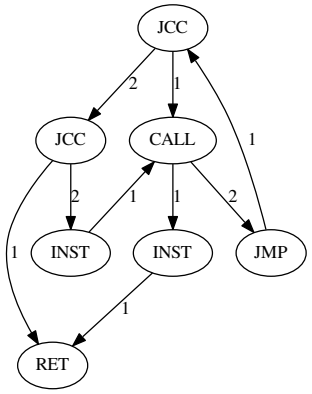


FIGURE 8.6 – Graphe de flot de test d'un programme à analyser

8.1.5 Problème de la détection des sites isomorphes

Le problème se pose alors de la manière suivante. On dispose d'une liste de sites et on veut déterminer, dans le graphe de flot de test T, les sites qui sont isomorphes à (au moins) un site présent dans la liste connue.

On a alors trois problèmes successifs, de difficulté croissante et faisant intervenir différents algorithmes.

Problème 8.1. *Comment savoir si deux graphes sont isomorphes ?*

Problème 8.2. *À partir d'un site P et d'un graphe de flot T, comment savoir si P est un sous-site de T ?*

Problème 8.3. *À partir d'un ensemble de sites L_S et d'un graphe de flot T, comment déterminer l'ensemble des $P \in L_S$ tels que P est un sous-site de T ?*

Dans le cas général, le problème 8.1 de l'isomorphisme de graphes est connu pour être NP sans que l'on sache s'il est NP-complet [KST93], tandis que le problème de l'isomorphisme de sous-graphes est NP-complet [Weg05]. Nous allons présenter quelques approches de ces problèmes et proposer des solutions tirant partie des spécificités des graphes de flot.

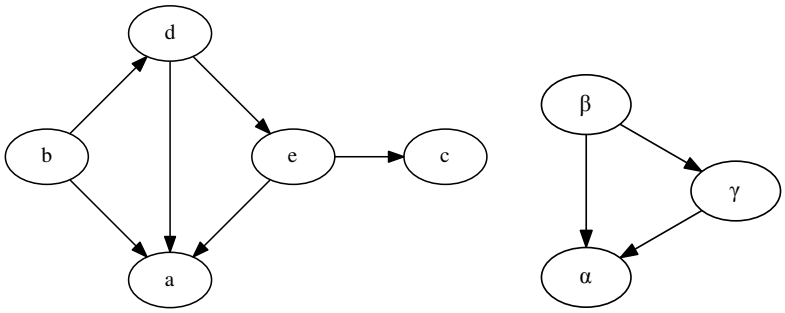
Nous verrons que les problèmes de l'isomorphisme de graphes et de sous-graphes de flot sont polynomiaux. C'est l'utilisation de fils ordonnés pour chaque sommet qui permet de classer ces problèmes dans P.

Nous introduirons dans les deux prochaines sections des approches générales au problème d'isomorphisme de graphes et de sous-graphes puis détaillerons par la suite deux techniques tirant partie des spécificités des graphes de flot et donnant donc une solution accessible en temps polynomial.

8.2 Approches générales à l'isomorphisme de sous-graphes

Nous présentons deux méthodes générales pour la résolution du problème d'isomorphisme de sous-graphes : l'approche matricielle par recherche exhaustive et l'algorithme d'Ullmann. Nous tenterons d'adapter ce second algorithme au problème des graphes de flot.

Dans cette section on traite donc le cas général de l'isomorphisme de graphes et de sous-graphes orientés comme l'illustre la figure 8.7. En particulier ces graphes ne sont, sauf mention du contraire, pas étiquetés et les fils des sommets ne sont pas ordonnés.



(a) Graphe T

(b) Graphe P, isomorphe à un sous-graphe de T

FIGURE 8.7 – Exemple de graphe et d'un sous-graphe

Nous donnons les définitions précédentes de graphe orienté, de sous-graphe et sous-graphe induit ainsi que d'isomorphisme de graphes, dans le cas habituel des graphes orientés quelconques, aux définitions 8.9, 8.10, 8.11 et 8.12, respectivement.

Définition 8.9. *Un graphe orienté $T = (V, E)$ est composé par les ensembles finis V et E. V est l'ensemble des sommets de T, $E \subset V \times V$ est l'ensemble des arcs de T tel que $(t, h) \in E$ si et seulement si il y a un arc entre le sommet t et le sommet h.*

Définition 8.10. *Un graphe orienté $P = (V_P, E_P)$ est un sous-graphe du graphe orienté et étiqueté $T = (V_T, E_T)$ si et seulement si $V_P \subset V_T$ et $E_P \subset \{(t, h) \in E_T / t, h \in V_P\}$.*

Définition 8.11. *Le sous-graphe induit d'un graphe orienté et étiqueté $T = (V_T, E_T)$ à partir de l'ensemble de sommets $V_P \subset V_T$ est le graphe $P = (V_P, E_P)$ tel que $E_P = \{(t, h) \in E_T / t, h \in V_P\}$.*

Définition 8.12. *Soit $T = (V_T, E_T)$ et $P = (V_P, E_P)$ deux graphes orientés. Ils sont dits isomorphes si et seulement si il existe une bijection $f : V_T \to V_P$ entre les sommets de T et ceux de P telle que f préserve la structure d'adjacence des graphes, c'est à dire si : $\forall u, v \in V_T, (u, v) \in E_T \Leftrightarrow (f(u), f(v)) \in E_P$.*

8.2.1 Représentation sous forme matricielle et recherche exhaustive

On peut représenter les graphes sous forme de matrices. On définit la matrice d'adjacence M d'un graphe T suivant les arcs qu'il contient si ses sommets sont ordonnés dans $V = \{v_1, v_2, ..., v_n\}$. Ensuite 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 8.13). Pour un graphe donné dont les sommets ne sont pas ordonnés, on peut choisir arbitrairement un ordre à ses sommets et dans ce cas il y a autant de matrices d'adjacence possibles pour ce graphe qu'il y a de permutations de ses sommets.

Par la suite la matrice M_T d'adjacence d'un graphe T représente une de ces matrices, arbitrairement fixée à la première occurence de M_T. Le choix d'une matrice d'adjacence n'a pas d'impact sur les solutions trouvées puisque toutes les matrices d'adjacence d'un graphe sont les mêmes à une permutation des sommets près, et par la suite, nous n'utilisons des égalités entre ces matrices qu'à une permutation près.

Définition 8.13. *Soit T=(V, E) un graphe orienté à $n \in \mathbb{N}$ sommets ordonnés. La matrice d'adjacence M associée à T est la matrice $n \times n$ définie par : $M_{i,j} = \begin{cases} 1 & si\ (i,j) \in E \\ 0 & sinon. \end{cases}$*

Par exemple, les graphe P et T de la figure 8.7 ont pour matrice d'adjacence les matrices de la figure 8.8.

$M_{i,j} =$

	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_T

$M_{i,j} =$

	α	β	γ
α	0	0	0
β	1	0	1
γ	1	0	0

(b) M_P

FIGURE 8.8 – Matrices d'adjacence des graphes T et P

Le problème de l'isomorphisme peut alors se reformuler. Deux graphes T et P sont isomorphes si et seulement si il existe une permutation σ des sommets du graphe T telle que $M_{\sigma(T)} = M_P$. Il est donc nécessaire et suffisant qu'il existe une matrice de permutation K telle que $M_P = K.M_T.K^t$ (car $K^{-1} = K^t$ pour une matrice de permutation), comme indiqué à la proposition 8.1.

Proposition 8.1. *Deux graphes T et P sont isomorphes si et seulement si il existe une matrice de permutation K telle que $M_P = K.M_T.K^t$.*

Le problème de l'isomorphisme de sous-graphes entre T et P, de tailles respectives n_T et n_P avec $n_T \geq n_P$, nécessite de trouver une permutation entre un sous-graphe de T et P. Puisqu'il est possible de prendre un sous-graphe qui ne contient pas tous les arcs de T et qu'un arc est représenté par un 1, la condition d'isomorphisme de sous-graphe sera une inégalité, comme indiqué à la proposition 8.2.

Proposition 8.2. *Il existe un isomorphisme de graphes entre un sous-graphe de T et P si et seulement si il existe une matrice de permutation K, carrée de taille n_T, telle que, en notant $Q = I_{n_P,n_T}.K$, $\forall k, l \leq n_P, (M_P)_{k,l} \leq (Q.M_T.Q^t)_{k,l}$.*

Preuve. *On cherche un sous-ensemble $\{s_1, s_2, ..., s_{n_P}\}$ de n_P sommets de T et un sous-ensemble d'arcs de T entre les sommets $\{s_1, s_2, ..., s_{n_P}\}$ tels que le sous-graphe défini par ces sommets et ces arcs soit isomorphe à P. Supposons qu'il existe un tel isomorphisme et on note la permutation σ de $\mathfrak{S}(n_T)$ telle que $\forall i \leq n_P$, $\sigma(s_i) \leq n_P$ donnant les n_P sommets de T y participant. Les autres éléments n'ont pas d'importance puisqu'ils ne participent pas à l'isomorphisme recherché. Soit K la matrice de permutation associée à σ. Les arcs présents dans P doivent être présents dans le sous-graphe de T induit à partir de $\{s_1, s_2, ..., s_{n_P}\}$, ce qui en terme de matrices d'adjacence équivaut à ce que, si on note la matrice du graphe induit $M_{\sigma(T)} = K.M_T.K^t$, ses éléments, pour les sommets qui participent à l'isomorphisme (les n_P premiers sommets), soient supérieurs aux éléments à la même place dans M_P : $\forall k, l \leq n_P, (M_P)_{k,l} \leq (K.M_T.K^t)_{k,l}$. Il suffit alors de réduire la taille de la matrice en utilisant une matrice I_{n_P,n_T} définie par $(I_{n_P,n_T})_{i,j} = \begin{cases} 1 & si\ i = j \\ 0 & sinon \end{cases}$.*

Dans ce cas, et si on pose $Q = I_{n_P,n_T}.K$ avec $Q \in \mathbb{M}_{n_P,n_T}$, on a $Q.M_T.Q^t = I_{n_P,n_T}.K.M_T.K^t.I_{n_T,n_P}$, la condition devient : il est nécessaire qu'il existe une matrice $Q \in \mathbb{M}_{n_P,n_T}$ telle que pour toute ligne k et colonne l de M_P, $(M_P)_{k,l} \leq (Q.M_T.Q^t)_{k,l}$.

Inversement, s'il existe une matrice de permutation K (carrée de taille n_T) telle que, en notant $Q = I_{n_P,n_T}.K$, $\forall k, l \leq n_P, (M_P)_{k,l} \leq (Q.M_T.Q^t)_{k,l}$ alors la matrice $Q.M_T.Q^t$ est la matrice d'un sous-graphe induit de T. De plus, en raison de l'inégalité, tous les arcs présents dans M_P sont présents dans ce sous-graphe induit : on en déduit que P est bien un sous-graphe de T.

Dans l'exemple des graphes de la figure 8.7, P est isomorphe au sous-graphe de T composé des sommets $\{d, e, a\}$ en faisant correspondre les couples (a, a), (d, b) et (e, c) entre T et P, soit la permutation σ telle que $\sigma(1) = 1$, $\sigma(4) = 2$ *et* $\sigma(5) = 3$. La matrice de permutation K est donnée en figure 8.9(a). La transformation qu'elle induit est donnée par la matrice $M_{\sigma(T)}$ en figure 8.9(b). Il paraît alors évident qu'en ne prenant que la sous-matrice 3×3, on obtient M_P, opération réalisée à l'aide de I_{n_P,n_T} (figure 8.9(c)).

L'approche par force brute demande de parcourir toutes les matrices de permutation de taille n_T : il y en a $n_T!$. Pour chaque matrice de permutation, une matrice transposée est déterminée et plusieurs multiplications matricielles sont faites sur des matrices de taille $n_T.n_T$, opération réalisable au pire en n_T^3. La complexité de cette approche dans le pire des cas est donc $O(n_T^3.n_T!)$ (proposition 8.3).

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

(a) Matrice de permutation K

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

(b) $M_{\sigma(T)} = K.M_T.K^t$

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

(c) I_{n_P,n_T}

FIGURE 8.9 – Matrices pour la démonstration sur l'exemple de T et P

Proposition 8.3. *La complexité de la recherche exhaustive d'un sous-graphe de T (de taille n_T) isomorphe à P est de $O(n_T^3.n_T!)$.*

8.2.2 Algorithme d'Ullmann

L'algorithme d'Ullmann est le plus connu pour résoudre le problème 8.2 d'isomorphisme de sous-graphes [Ull76] dans le cas général des graphes orientés. Il s'applique à un graphe de motif P à détecter et à un graphe T dans lequel faire la reconnaissance. Comme détaillé par la suite, sa complexité est exponentielle en le nombre de sommets maximal du graphe de motif P.

De plus, si on veut l'utiliser pour résoudre le problème 8.3 de recherche d'isomorphismes de sous-graphes dans une base de graphes de motifs, il est nécessaire de l'appliquer pour chaque graphe dans la base. La complexité est alors linéaire en le nombre de graphes dans la base. Puisqu'il est complet pour le problème d'isomorphisme de sous-graphes, ce sera l'algorithme de référence auquel on pourra comparer les heuristiques sur le plan de la complétude, de la complexité et du temps d'exécution.

Nous présenterons d'abord l'algorithme d'Ullmann dans le cas de graphes orientés quelconques puis nous y ferons des restrictions pour l'adapter au cas des graphes de flot.

Backtracking. L'algorithme d'Ullmann repose sur le concept de retour sur traces (ou *backtracking*) pour trouver successivement tous les sous-graphes isomorphes au graphe de motif. Cette approche a ensuite été largement améliorée par Ullmann à l'aide de l'utilisation d'une technique de vérification a priori permettant d'éliminer de nombreux candidats sans avoir à les tester entièrement.

Backtracking simple

Soit $T = (V_T, E_T)$ un graphe et $P = (V_P, E_P)$ un graphe dont on cherche à connaître les sous-graphes de T avec lesquels il est isomorphe. On note n_T et n_P le nombre de sommets de chaque graphe. L'idée consiste à associer un sommet de P à un sommet de T puis à vérifier si l'association créée est bien un isomorphisme de sous-graphe. Si c'est le cas, on prend un autre sommet de P à associer avec un sommet de T 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. Lorsque l'on a une transformation utilisant tous les sommets de P et définissant un isomorphisme avec un sous-graphe de T, on la met de côté (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 K, carrée de taille n_T, de la transformation, puis la matrice I_{n_P,n_T} permettant le redimensionnement d'une matrice, et on regarde si la relation $\forall k, l \leq n_P, (M_P)_{k,l} \leq (I_{n_P,n_T}.K.M_T.K^t.I_{n_T,n_P})_{k,l}$ est vérifiée.

Optimisation. On peut dès le départ savoir que certaines associations ne donneront pas lieu à un sous-graphe. Avec les graphes P et T de la figure 8.7, il est clair que le sommet γ du graphe P ne peut pas être associé au sommet c du graphe T. En effet γ a strictement plus d'arcs sortants que c, la structure autour de γ ne pourrait donc pas être préservée par un isomorphisme. On construit une matrice $C^0 \in \mathbb{M}_{n_P,n_T}$ telle que, pour $i \leq n_P, j \leq n_T$, on n'essaie d'associer les sommets i et j que si $C^0_{i,j} = 1$. En prenant en compte la remarque précédente, on définit alors

$$C^0_{i,j} = \begin{cases} 1 & \text{si le nombre d'arcs entrants est plus petit pour i dans P que pour j dans T, et si le nombre d'arcs sortants est plus petit pour i dans P que pour j dans T} \\ 0 & \text{sinon.} \end{cases}$$

Les conditions sur les inégalités sont prises au sens large.

Le calcul effectif du nombre d'arcs entrants (respectivement sortants) 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).

Algorithme. L'algorithme est détaillé dans la figure 8.3 : il consiste à associer chaque sommet de P à un sommet de T. On note i le numéro du prochain sommet de P à associer à un sommet de T. On démarre donc avec i=1. On conserve la permutation σ sous la forme d'une liste F de couples de la forme (i, j) avec $i \in T$, et $j \in P$. La première étape consiste donc à initialiser les matrices que l'on va utiliser (M_T, M_P, I_{n_P,n_T}, *et* C^0). Une fois l'initialisation faite, pour parcourir toutes les possibilités, on utilise la procédure *backtrack* en partant d'une matrice de possibilités C^0, de i=1, et d'une liste F vide. Pour chaque association possible de i dans T avec un élément j de P (possibilité inscrite dans C^0), on ajoute (i, j) à F, on met à jour C (les associations choisies ne pourront plus l'être ensuite), on détermine les matrices d'adjacence des sous-graphes induits de T et P par les associations choisies dans F, notées S_T et S_P respectivement, ainsi que la matrice de permutation K définie par F. On peut alors vérifier qu'il y a un isomorphisme de sous-graphe entre les graphes induits de P et T avec la relation suivante : $\forall k, l \leq i, (S_P)_{k,l} \leq (K.S_T.K^t)_{k,l}$, détaillée dans la section précédente.

— Si c'est le cas et que $i = n_P$ alors on a trouvé un isomorphisme de sous-graphe entre P et T, on le renvoie.
— Si c'est le cas mais que $i \neq n_P$ alors on continue à chercher à ajouter des éléments à l'isomorphisme en appelant *backtrack* avec C', $i+1$ et F.
— Si ce n'est pas le cas, on ne fait rien.

Ensuite il suffit de retirer la dernière association (i, j) et de tester l'association suivante.

Il est à noter que l'on peut déterminer la matrice de permutation K 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_T et S_P de T et P 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 $\exists i,\ (i,\ j) \in F$.

Algorithme 8.3 : Retour sur traces simple

Données : Deux graphes P (de taille n_P) et T (de taille n_T)
Résultat : L'ensemble des listes d'association donnant un isomorphisme de sous-graphes entre P et T

Initialiser C^0
backtrack(C^0, 1, $\emptyset$)

```
backtrack(C, i, F) /* C : matrice des associations possibles, i :
numéro du prochain sommet de P à associer, F : liste des couples
d'associations faites                                            */
```

$E \leftarrow \emptyset$; /* E contiendra les solutions */
pour $j \in [\![1, n_T]\!]$ *tel que* $C_{i,j} = 1$ **faire**
  $F \leftarrow F, (i, j)$
  $C' \leftarrow C$ et $\forall k > i,\ C'_{k,j} \leftarrow 0$
  $S_P \leftarrow$ la matrice d'adjacence du sous-graphe de P induit par les sommets $1..i$
  $S_T \leftarrow$ la matrice d'adjacence du sous-graphe de T induit par les sommets de $\{j, (k, j) \in F\}$
  $K \leftarrow$ la matrice de permutation définie par F
  si $\forall k, l \leq i, (S_P)_{k,l} \leq (K.S_T.K^t)_{k,l}$ **alors**
    si $i = n_P$ **alors**
      $E \leftarrow E \cup \{F\}$
    sinon
      $E \leftarrow E \cup backtrack(C',\ i+1,\ F)$
    fin
  fin
  retirer le dernier élément de F
fin
retourner E

Raffinement et vérification a priori

L'algorithme détaillé précédemment est une application directe du backtracking au problème d'isomorphisme de sous-graphes et Ullmann propose un deuxième algorithme basé sur une vérification a 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 est donc faite par la fonction *forwardChecking* (algorithme 8.4) avant l'appel récursif à la nouvelle procédure *backtrack* (algorithme 8.5). Cette vérification tient de la définition d'un isomorphisme en tant que graphe. Pour qu'un sous-graphe S_T de T choisi soit isomorphe au sous-graphe S_P de P auquel il est associé, il est nécessaire que tous les arcs de S_P soient présents dans S_T : si deux sommets sont reliés par un arc dans S_P alors leurs correspondants dans S_T doivent aussi être reliés par un arc orienté de la même manière. Si cette condition est vérifiée alors la matrice C 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 tester au vue des modifications faites à C, la fonction *forwardChecking* renvoie *false*. Elle renvoie *true* dans le cas contraire.

Complexité

On peut étudier la complexité temporelle dans le pire des cas de l'algorithme d'Ullmann en gardant les notations prises précédemment. Le pire des cas se présente lorsque toutes les associations sont possibles [Mes95] (C^0 n'est composée que de 1), c'est à dire que les deux graphes sont entièrement connectés. La complexité de l'algorithme d'Ullmann avec vérification a priori est donnée à la proposition 8.4. Nous verrons par la suite qu'elle peut se réduire dans le cas des graphes de flot.

Proposition 8.4. *La complexité dans le pire des cas de l'algorithme d'Ullmann est en* $O(n_P.n_T^{n_P+1})$.

Preuve. *Notons BT(i) et FW(i) respectivement la complexité des fonctions* backtrack *et* forwardChecking *avec en entrée l'entier i. L'algorithme d'Ullmann consistant à un appel de la fonction récursive BT(1), c'est la complexité de cet appel que l'on cherche à évaluer.*

À chaque appel de BT ou FW, on a déjà attribué $i-1$ *associations et on cherche à attribuer l'association i. Avant la première association, tout est possible, on a donc* $n_P.n_T$ *associations possibles, puis* C^0 *est mis à jour par BT et FW, on a alors* $(n_T-1)(n_P-1)$ *possibilités. Après avoir attribué* $i-1$ *associations on a alors* $(n_T-(i-1))(n_P-(i-1))$ *possibilités d'associations au maximum, et* $n_T-(i-1)$ *associations possibles dans le graphe test T pour le i-ème sommet du graphe de motif P. La fonction BT(i) vérifie, pour toutes ces associations encore possibles, si FW(i) est vrai (ce qui est le cas car toutes les associations sont possibles) et appelle BT(i+1).*

Ainsi $BT(i) = (n_T-(i-1))(FW(i)+BT(i+1))$. *De plus* $FW(n_P)$ *se fait en temps constant* f_0.

La fonction FW vérifie, $\forall k = (i+1)..n_P$ *et* $l = 1..n_T$ *si* $C_{k,l} = 1$, *et dans ce cas effectue un nombre de vérifications borné par* $n_P.(2n_T+2n_P)$ *car elle vérifie une condition sur tous les fils de deux sommets de P et une condition sur tous les fils de deux sommets de*

Algorithme 8.4 : Vérification a priori

Données : Les associations possibles dans C, le numéro courant i du sommet de P, les associations déjà réalisées dans F

Résultat : Les associations mises à jour et vrai ou faux selon si une future association est possible

```
forwardChecking(C, i, F)
  pour (k,l) ∈ [[i+1, n_P]] × [[1, n_T]] faire
    si C_{k,l} = 1 alors
      // On regarde si on peut faire correspondre k dans P à l
      dans T
      pour (v,w) ∈ F faire
        si (k,v) ∈ E_P et (l,w) ∉ E_T alors
          C_{k,l} ← 0
        fin
        si (v,k) ∈ E_P et (w,l) ∉ E_T alors
          C_{k,l} ← 0
        fin
      fin
    fin
  fin
  // On vérifie si chaque k ≥ i+1 dans P peut encore être associé à
  au moins un élément de T
  pour k ∈ [[i+1, n_P]] faire
    si ∀l ∈ [[1, n_T]], C_{k,l} = 0 alors
      retourner (C, false)
    fin
  fin
  retourner (C, true)
```

T, et ce pour toute association dans F. Une vérification sur tous les fils d'un sommet de P se fait en un temps maximum de n_P si les fils sont représentés sous forme d'une liste et la vérification sur tous les fils d'un sommet de T au maximum en n_T. De plus $n_P \leq n_T$ donc la complexité est au pire de $4.n_T$. Ainsi, au pire, $FW(i) = 4.n_P.n_T(n_T-(i-1))(n_P-(i-1))+(n_P-i).n_T \leq 4.n_P.n_T(n_T-(i-1)+1)(n_P-(i-1))$ sauf pour $FW(1) = (n_P-1).n_T$.

Le problème est alors le suivant. On cherche à déterminer $BT(1)$.

$$\begin{cases} FW(1) = (n_P-1).n_T \\ FW(i) = 4.n_P.n_T.(n_T-i+2)(n_P-i+1) \text{ pour } 2 \leq i \leq n_P-1 \\ FW(n_P) = f_0 \\ BT(i) = (n_T-(i-1))(FW(i)+BT(i+1)) \text{ pour } 1 \leq i \leq n_P-1 \\ BT(n_P) = (n_T-(n_P-1)).FW(n_P) \end{cases}$$

On peut remarquer que, quel que soit i, le terme $FW(i)$ n'apparaît qu'en multiple du terme $BT(i)$ et on peut compter son nombre d'apparition : il apparaît $n_T(n_T-1)(n_T-2)...(n_T-(i-1))$ fois, soit $A_{n_T}^i$ fois avec $A_n^k = \frac{n!}{(n-k)!}$

Algorithme 8.5 : Retour sur traces avec vérification a priori

Données : Deux graphes P (de taille n_P) et T (de taille n_T)
Résultat : L'ensemble de listes d'association donnant un isomorphisme de sous-graphes entre P et T

Initialiser C^0
backtrack(C^0, 1, $\emptyset$)

```
backtrack(C, i, F) /* C : matrice des associations possibles, i :
numéro du prochain sommet de P à associer, F : liste des couples
d'associations faites                                            */
```

$E \leftarrow \emptyset$
pour $j \in [\![1, n_T]\!]$ *tel que* $C_{i,j} = 1$ **faire**
  $F \leftarrow F, (i, j)$
  $C' \leftarrow C$ et $\forall k > i,\ C'_{k,j} \leftarrow 0$
  $(C'', continue) \leftarrow forwardChecking(C', i, F)$
  si *continue* **alors**
    si $i = n_P$ **alors**
      $E \leftarrow E \cup \{F\}$
    sinon
      $E \leftarrow E \cup backtrack(C'',\ i+1,\ F)$
    fin
  fin
  retirer le dernier élément de F
fin
retourner E

Notons C la complexité totale de $BT(1)$.

$$C = A^1_{n_T} FW(1) + A^2_{n_T} FW(2) + ... + A^{n_P-2}_{n_T} FW(n_P - 2) + A^{n_P-1}_{n_T} FW(n_P - 1) + A^{n_P}_{n_T}.f_0$$

On a vu que $FW(i) = 4.n_P.n_T.(n_T - i + 2)(n_P - i + 1) \leq 4.n_P^2.n_T^2$, notons $F = 4.n_P^2.n_T^2$.
De même $FW(1) = (n_P - 1).n_T \leq F$.
De plus $A^i_{n_T} = O(n_T^i)$, donc $\forall i \leq n_P - 3,\ A^i_{n_T} = O(n_T^{n_P-3})$
Ainsi il existe une constante k telle vérifiant l'inégalité suivante.

$$A^1_{n_T} FW(1) + A^2_{n_T} FW(2) + ... + A^{n_P-3}_{n_T} FW(n_P - 3) \leq (n_P - 3).F.k.n_T^{n_P-3}$$

De même il existe k', k'' et k''' telles que

$$C \leq (n_P - 3).F.k.n_T^{n_P-3} + k'.FW(n_P - 2).n_T^{n_P-2} + k''.FW(n_P - 1).n_T^{n_P-1} + k'''.f_0.n_T^{n_P}$$

avec les inégalités suivantes :
- $FW(n_P - 2) \leq 4.n_P.n_T.(n_T - n_P + 4).3 \leq 12.n_P.n_T^2$ *et*
- $FW(n_P - 1) \leq 4.n_P.n_T.(n_T - n_P + 3).2 \leq 8.n_P.n_T^2$, *et*
- $n_P \leq n_T$.

Donc

$$C <= F.k.n_T^{n_P-2} + 12.n_P.n_T^2.k'.n_T^{n_P-2} + 8.n_P.n_T^2.k''.n_T^{n_P-1} + k'''.f_0.n_T^{n_P},$$

$$C <= 4.k.n_P.n_T^{n_P+1} + 12.k'.n_P.n_T^{n_P} + 8.k''.n_P.n_T^{n_P+1} + k'''.f_0.n_T^{n_P+1}.$$

Ainsi $C = O(n_P.n_T^{n_P+1})$.

Optimisation dans le cas des graphes de flot

Les sommets sont étiquetés. Pour que deux sommets correspondent dans un isomorphisme entre deux graphes de flot, on rend nécessaire qu'ils aient la même étiquette. À partir d'un échantillon de 43 logiciels malveillants pris comme référence, on tente de voir le gain qu'une telle approche apporte dans le pire des cas avec l'algorithme d'Ullmann. Voici la répartition de ces 48218 instructions, les valeurs numériques ont été grossièrement arrondies.

— 1994 RET, 4.135%, soit $l_1 = 5\%$
— 3078 JMP, 6.384%, soit $l_2 = 5\%$
— 8868 JCC, 18.391%, soit $l_3 = 20\%$
— 9604 CALL, 19.918%, soit $l_4 = 20\%$
— 24582 INST, 50.981%, soit $l_5 = 50\%$
— 192 autres (UNDEFINED ou HLT), 0.394%, soit 0%

Formellement, on suppose que l'on a n différents types d'instructions (ici 5) dont les probabilités d'apparition sont l_i avec $1 \leq i \leq n$. Ce qui est modifié, c'est la matrice C^0 initiale, qui au lieu de n'être composée que de 1, vaut 0 si $L_i \neq L_j$ où L_i est le label de i et L_j celui de j. Si on note h la probabilité pour deux sommets pris au hasard d'avoir le même label, à chaque fois que l'on veut faire correspondre un élément du graphe de motif au graphe à tester, il y a le nombre normal de possibilités multiplié par h, dans l'hypothèse où n_T est grand devant n_P et donc à chaque étape la probabilité h reste constante. Ainsi dans le calcul de la complexité, il suffit de multiplier n_T par h pour obtenir la complexité modifiée.

Supposons que les graphes P et T ont pour chaque label les même probabilités d'apparition et $n_T >> n_P$. On a alors :

$$h = P(m_{i,j}^0 = 1) = P(L_i = L_j) = P(\cup_k(L_i = k \cap L_j = k)),$$

$$h = \sum_k P(L_i = k \ \cap \ L_j = k) \text{ car les labels sont tous différents (événements disjoints)},$$

$$h = \sum_k P(L_i = k).P(L_j = k) \text{ car ces événements sont indépendants},$$

$$h = \sum_k l_k^2.$$

Dans le cas que l'on traite les sous-sites P sont de petite taille (inférieure à 24 sommets) tandis que les graphes de flot T peuvent avoir plusieurs milliers de sommets. Comme

$n_T >> n_P$, chaque choix de correspondance ne modifie pas la probabilité h que $C^0_{i,j} = 1$ parmi les sommets restants bien que certaines correspondances aient déjà été choisies (leur nombre est très faible par rapport au nombre total de correspondances possibles).

Dans notre exemple h=0.335, soit 34%. Prendre en compte les types d'instructions est équivalent à diviser par 3 la taille du graphe à tester en terme de complexité temporelle dans le pire des cas.
Ainsi, dans notre cas la complexité est $O(n_P.(n_T/3)^{n_P+1})$ alors qu'elle est, dans le cas général, de $O(n_P.(n_T.h)^{n_P+1})$ avec $h = \sum_k l_k^2$.

L'arité des étiquettes est bornée. On sait qu'il existe une borne k au nombre de fils de tous les sommets de nos graphes, avec $k = 2$: les instructions ayant le maximum de fils sont les sauts conditionnels et les appels de fonction. La complexité de la fonction *forwardChecking* en est modifiée car les vérifications des conditions sur les enfants de chaque sommet se font désormais en temps maximal fixé à k. On a donc dans le pire des cas $FW = 4.k.n_T.n_P^2$, ce qui se traduit par la perte en complexité totale d'un multiple de n_T.

Les fils sont ordonnés. De plus on doit maintenant forcer les sommets correspondants entre P et T à être ordonnés de la même manière par rapport à leur père. La nouvelle fonction *forwardChecking*, prenant en compte les optimisations discutées, est l'algorithme 8.6. Cette modification semble diminuer grandement la complexité mais nous ne l'avons pas quantifiée.

Optimisation de la vérification a priori. Dans la fonction *forwardChecking*, il n'est pas utile de vérifier tous les éléments de F à chaque appel, mais seulement les derniers élements ajoutés (qui correspondent exactement à la dernière association ajoutée). Cette modification ne change pas le comportement ni les résultats de l'algorithme puisque les éléments précédents dans F ont déjà été vérifiés. Cette modification enlève un multiple de n_P à la complexité de la fonction *forwardChecking* et donc enlève un multiple de n_P à la complexité totale.

Conclusion sur la complexité. Pour le cas d'un graphe de flot, la complexité a été ramenée à $O((n_T.h)^{n_P})$ avec $h = \sum_k l_k^2 \simeq 0.34$ où les l_i sont les probabilités d'apparition de chacun des labels (proposition 8.5).

Proposition 8.5. *La complexité de la recherche d'un sous-graphe de T (de taille n_T) isomorphe à P (de taille n_P) avec l'algorithme d'Ullmann optimisé est de $O((n_T.h)^{n_P})$ avec $h \simeq 0.34$.*

Algorithme 8.6 : Vérification a priori pour graphes de flot

Par rapport à l'algorithme 8.4, on ne vérifie que le dernier élément de F s'il existe et on vérifie que les ordres des fils correspondent.

Entrées : Les associations possibles dans C, le numéro courant i du sommet de P, les associations déjà réalisées dans F

Résultat : Les associations mises à jour et vrai ou faux selon si une future association est possible

```
forwardChecking(C, i, F)
  pour (k,l) ∈ [[i+1, n_P]] × [[1, n_T]] faire
    si C_{k,l} = 1 alors
      // On regarde si on peut faire correspondre k dans P à l
      dans T
      Soit (v,w) ← dernier élément de F
      si (k,v) ∈ E_P avec l'ordre i_P et ((l,w) ∉ E_T ou (l,w) ∈ E_T avec un
      ordre i_T ≠ i_P) alors
        C_{k,l} ← 0
      fin
      si (v,k) ∈ E_P avec l'ordre i_P et ((w,l) ∉ E_T ou (w,l) ∈ E_T avec un
      ordre i_T ≠ i_P) alors
        C_{k,l} ← 0
      fin
    fin
  fin
  // On vérifie si chaque k ≥ i+1 dans P peut encore être associé à
  au moins un élément de T
  pour k ∈ [[i+1, n_P]] faire
    si ∀l ∈ [[1, n_T]], C_{k,l} = 0 alors
      retourner (C, false)
    fin
  fin
  retourner (C, true)
```

8.2.3 Revue de littérature

Optimisation de l'algorithme d'Ullmann. Foggia, Vento, Sansone et Cordella [Cor+99] proposent des améliorations à l'algorithme d'Ullmann au sein d'un nouvel algorithme appelé VF. Ils ajoutent des règles de vérification a priori permettant d'éliminer à nouveau de nombreuses vérifications lors de l'appel de la fonction de retour sur traces. Les améliorations en termes de complexité sont importantes : ils réussissent à diviser la complexité d'un facteur de n_T^2 par rapport à l'algorithme d'Ullmann. Ils comparent également en pratique plusieurs algorithmes [FSV01] dont celui d'Ullmann, le leur et une version modifiée, VF2. Ils montrent la pertinence de leur approche sur différents graphes dont la taille varie jusqu'à un millier de sommets.

Calcul au préalable des isomorphismes. Messmer [Mes95] propose plusieurs approches pour la comparaison de graphes lorsqu'on peut générer une base de détection au préalable. Pour l'isomorphisme de sous-graphes, Messmer et Bunke proposent [MB95] de construire la matrice d'adjacence de chaque graphe du corpus, de générer toutes les matrices d'adjacence possibles par diminution du nombre de sommets et par permutations et de les ranger dans un arbre de décision. L'analyse d'un graphe inconnu est extrêmement rapide puisqu'il suffit de parcourir l'arbre de décision avec sa matrice d'adjacence pour vérifier s'il est un sous-graphe d'un graphe du corpus. La limite de l'approche est donc qu'il faut non seulement générer tous les isomorphismes, mais en plus les stocker. Les auteurs montrent, en appliquant quelques optimisations, que l'on peut générer et stocker tous les isomorphismes de sous-graphes pour des graphes allant jusqu'à 22 sommets.

Le cas des graphes de flot est assez différent puisque l'on va chercher à comparer des petits graphes de motif (moins de 24 sommets) et des graphes de test de grande taille (plusieurs dizaines de milliers de sommets). L'algorithme d'Ullmann ainsi que les deux approches citées dans cette section ne sont pas adaptés pour ce problème. Nous avons alors cherché des approches prenant en comptes les spécificités des graphes de flot.

8.3 Approches spécifiques aux graphes de flot

8.3.1 Algorithme par parcours de graphes de flot

Tarjan [Tar71] propose un parcours en profondeur d'un graphe qui permet d'en simplifier la structure en construisant un arbre le recouvrant. On peut adapter cette approche pour tester l'isomorphisme de sous-sites. Tout d'abord on formalise le parcours d'un graphe à la définition 8.14 et plus spécifiquement le parcours d'un site à la définition 8.15. Le parcours d'un site peut être bijectif car tous les sommets du site sont accessibles depuis sa racine.

Définition 8.14. *Soit T un graphe de taille n. On appelle parcours de T depuis le sommet s toute numérotation $\pi : V \to [\![1, n]\!]$ de ses sommets vérifiant les propriétés suivantes.*

— $\pi(s) = 1$
— *π est injective*
— *$\forall\ i \geq 2$, $\pi^{-1}(i)$, s'il existe, est le fils d'au moins un sommet s' vérifiant $\pi(s') < i$*

Définition 8.15. *On appelle parcours d'un site S tout parcours bijectif du graphe représenté par S à partir de sa racine.*

Exemples. Le parcours en profondeur d'un site à partir d'un sommet est récursif. Il part de ce sommet (à l'origine la racine numérotée 1), numérote le site à partir du fils 1 du sommet, puis numérote le site à partir du fils 2... jusqu'au dernier fils du sommet. Le parcours en largeur d'un site numérote la racine puis ses fils directs, puis les fils du premier fils, les fils du second, etc. Un exemple de parcours en profondeur d'un graphe de flot est donné en figure 8.10. Nous exploitons donc le fait que les fils des sommets d'un graphe de flot sont numérotés.

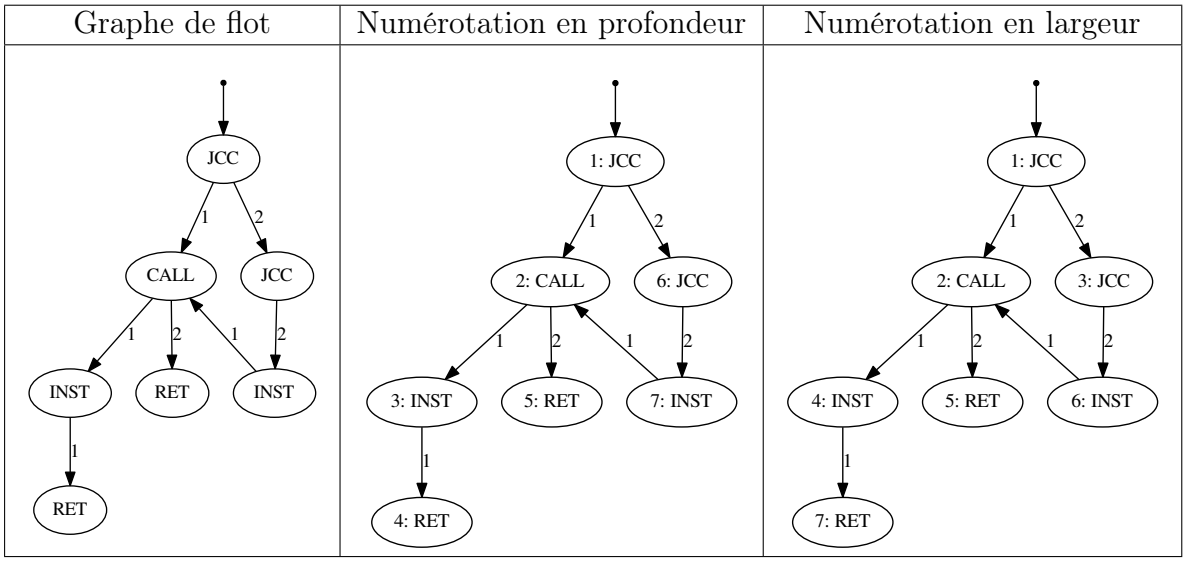


FIGURE 8.10 – Graphe de flot numéroté en profondeur et en largeur

On peut représenter le parcours en profondeur de ce site de la manière suivante. On commence par la racine, que l'on numérote 1, c'est un JCC, puis le premier fils nous amène à un nouveau sommet, numéroté 2, c'est un CALL. Son premier fils nous amène à numéroter 3 : INST qui amène, par son premier fils, à 4 : RET. Une fois au 4e sommet, on retourne au sommet 2 pour prendre le second fils que l'on numérote 5 : RET. On remonte au sommet 1, on numérote le second fils : c'est le sommet 6 : JCC puis on prend son seul fils (fils 2) et on numérote 7 : INST. De là on prend le premier fils pour retourner sur le fils 2. Le parcours s'arrête puisque l'on a visité tous les arcs du site.
On peut noter le parcours ainsi (avec l'algorithme 8.7) :
$1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{1} 3 : INST \xrightarrow{1} 4 : RET \xrightarrow{R} 2 \xrightarrow{2} 5 : RET \xrightarrow{R} 1 \xrightarrow{2} 6 : JCC \xrightarrow{2} 7 :$

$INST \xrightarrow{1} 2$.
De même, le parcours en largeur du site peut être décrit ainsi (avec l'algorithme 8.8) :
$1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{R} 1 \xrightarrow{2} 3 : JCC \xrightarrow{R} 2 \xrightarrow{1} 4 : INST \xrightarrow{R} 2 \xrightarrow{2} 5 : RET \xrightarrow{R} 3 \xrightarrow{2} 6 : INST \xrightarrow{R} 4 \xrightarrow{1} 7 : RET$.

Exemple de graphe présentant un cycle. La présence d'un cycle dans le graphe de flot n'est pas un obstacle à cette approche puisque chaque arête n'est représentée qu'une seule fois. La figure 8.11 donne un exemple de graphe de flot et de sa numérotation en largeur et en profondeur. On peut noter le parcours en profondeur ainsi :
$1 : JCC \xrightarrow{1} 2 : INST \xrightarrow{1} 3 : JCC \xrightarrow{1} 1 \xrightarrow{R} 3 \xrightarrow{2} 4 : RET \xrightarrow{R} 1 \xrightarrow{2} 5 : INST$.
Et le parcours en largeur de cette manière :
$1 : JCC \xrightarrow{1} 2 : INST \xrightarrow{R} 1 \xrightarrow{2} 3 : INST \xrightarrow{R} 2 \xrightarrow{1} 4 : JCC \xrightarrow{R} 3 \xrightarrow{1} 5 : RET \xrightarrow{R} 4 \xrightarrow{1} 1 \xrightarrow{R} 4 \xrightarrow{2} 5$.

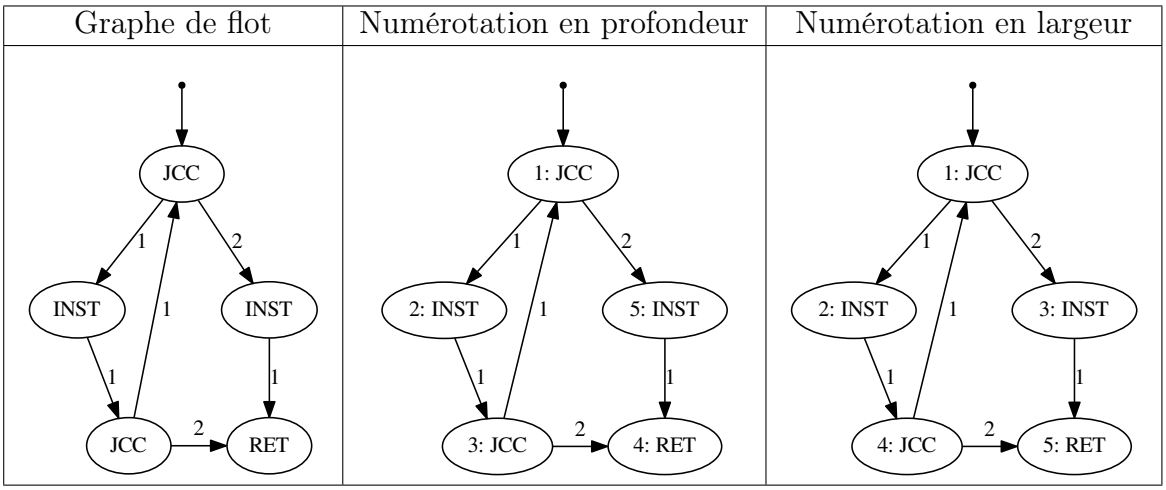


FIGURE 8.11 – Graphe de flot présentant un cycle et sa numérotation en profondeur et en largeur

Algorithme 8.7 : Codage du parcours en profondeur d'un site

Données : Un site de racine R
Résultat : Le codage du parcours en profondeur de ce site

$(i, c) \leftarrow parcoursProfondeur(R, 1, \emptyset, etiq, \emptyset)$
retourner c

```
parcoursProfondeur(s, i, E, etiq, num) /* s: sommet, i: premier numéro
non attribué, E: sommets déjà explorés, etiq: associe une étiquette à
un sommet, num: associe un sommet à son numéro                    */
```

si $s \notin E$ **alors**
  $E \leftarrow E \cup \{s\}$
  $num(s) \leftarrow i$
  $C \leftarrow$ "$i : etiq(s)$"
  $i \leftarrow i + 1$
  pour *f fils numéro k de s* **faire**
    $C \leftarrow C +$ " $\xrightarrow{k}$ "
    $(i, c) \leftarrow parcoursProfondeur(f, i, E, etiq, num)$
    $C \leftarrow C + c$
    si *s a encore au moins un fils* **alors**
      $C \leftarrow C +$ " $\xrightarrow{R} num(s)$"
    fin
  fin
  retourner (i, C)
sinon
  retourner $(i,$ "$num(s)$"$)$
fin

Algorithme 8.8 : Codage du parcours en largeur d'un site

```
Données : Un site de racine R
Résultat : Le codage du parcours en largeur de ce site
parcoursLargeur(etiq) /* etiq: associe une étiquette à un sommet     */
s_c ← ϵ;                                  /* Le sommet courant devient ϵ. */
i ← 1;                                     /* On numérote à partir de 1. */
C ← ""
A ← [(ϵ, 0, R)];                        /* A contient les arcs à ajouter. */
E ← {};                  /* E est l'ensemble des sommets déjà visités. */
tant que il y a un premier élément a=(p, k, s) dans A faire
    Retirer le premier élément de A
    si s_c n'est ni p ni ϵ alors
        C ← C + " -R-> num(p)"
        s_c ← p
    fin
    si s n'est pas dans E alors
        si p est ϵ alors
            C ← C + "1 : etiq(R)"
        sinon
            C ← C + " -k-> i : etiq(s)"
        fin
        num(s) ← i
        i ← i + 1
        s_c ← s
        pour fils f numéro k' du sommet s faire
            A ← A + (s_c, k', f)
        fin
        E ← E ∪ {s_c}
    sinon
        C ← C + " -k-> num(s)"
        s_c ← s
    fin
fin
retourner C
```

En fait, codé de cette manière, un parcours définit un site unique que l'on peut reconstruire puisque l'on sait quelle est l'étiquette de chaque sommet et l'on connaît tous les arcs entre les sommets à l'aide du choix des fils fait lors du parcours (ici en profondeur). On définit les mots constituant un codage de parcours et un parcours bien formé (définitions 8.16 et 8.17) afin que l'on puisse reconstruire un site à partir de n'importe quel codage de parcours bien formé (algorithme 8.9).

Définition 8.16. *Un mot de parcours est soit :*

- *de type m_1, c'est à dire $1 : L$ où L est une étiquette, ou*
- *de type m_2, c'est à dire $\xrightarrow{\alpha} i[: L]$ où i est un entier, L est une étiquette et α est soit un entier k, soit R.*

Définition 8.17. *On appelle codage de parcours bien formé tout mot vérifiant les quatre conditions suivantes.*

1. *Il est de la forme $m_1(m_2)^*$*
2. *L'étiquette de chaque sommet i est spécifiée une unique fois, lors la première référence à ce sommet : à i fixé, si un mot de la forme $[\xrightarrow{\alpha}]i : L$ est présent alors aucun mot i ne le précède et aucun mot suivant n'est de la forme $i : L$*
3. *Pour tout mot de la forme $\xrightarrow{R} i$, un mot $i : L$ est présent précédemment dans le codage*
4. *Chaque fils de chaque sommet est défini au plus une fois : à i et k fixés, il ne doit y avoir qu'au plus une occurrence du mot $i[: L] \xrightarrow{k}$*

Algorithme 8.9 : Reconstruction d'un site depuis un codage de parcours bien formé

Données : Un codage de parcours bien formé
Résultat : Un site
`reconstructionSite(`*codage*`) /* Le codage est de la forme` $1 : L(m_2)^*$ `*/`
  Création d'un site vide
  Ajout du sommet 1 d'étiquette L qui est la racine du site et devient le sommet courant
  pour *chaque mot de type* $\xrightarrow{\alpha} i[: L]$ **faire**
    si α *est un entier* k **alors**
      Ajout du sommet i d'étiquette L s'il n'existait pas
      Ajout d'un arc d'étiquette k entre le sommet courant et le sommet i
      Le sommet i devient le sommet courant
    sinon
      Le sommet i devient le sommet courant
    fin
  fin
  retourner le site reconstitué

Un codage de parcours bien formé est le codage d'un site S si S est le site que l'on reconstruit à partir du codage (définition 8.18). Les algorithmes donnant le codage d'un site selon un parcours en profondeur ou en largeur fournissent bien un codage de ce site (propositions 8.6 et 8.7).

Définition 8.18. *On appelle codage d'un site S tout codage de parcours bien formé dont le site associé par l'algorithme 8.9 est S.*

Proposition 8.6. *Le codage du parcours en profondeur fourni par l'algorithme 8.7 à partir de n'importe quel site S est un codage de S.*

Preuve. *Soit un site S (non vide) de racine R. Nous prouvons d'abord que l'algorithme 8.7 appliqué à S fournit un codage de parcours bien formé. La sortie de la fonction* parcoursProfondeur *est de la forme $i[: L](\xrightarrow{\alpha} i[: L])*$ et le mot $1 : etiq(R)$ est présent au début de la sortie de l'algorithme. La sortie de l'algorithme est donc bien de la forme $m_1(m_2)^*$.*

L'étiquette de chaque sommet est spécifiée au moins une fois parce qu'en partant de la racine de S le parcours en profondeur atteint tous les sommets de S et si le sommet s n'a pas encore été traité, un mot de la forme $i : etiq(s)$ est ajouté. L'étiquette ne peut être ajoutée que si s n'a pas encore été parcouru et donc elle est spécifiée au plus une fois.

Les mots de la forme $\xrightarrow{R} i$ ne sont ajoutés qu'avec i étant la numérotation du site s en cours de traitement et dont l'étiquette a déjà été définie.

Chaque sommet n'est traité qu'une seule fois et les mots de la forme $\xrightarrow{k} i$ définissant ses fils ne sont donc ajoutés qu'une seule fois.

Nous cherchons maintenant à prouver que pour tout site S le codage fourni par l'algorithme 8.7 de parcours en profondeur permet de reconstruire, à l'aide de l'algorithme 8.9, le site S. Notons $s_1, s_2, ..., s_n$ les sommets de S numérotés selon l'algorithme de parcours en profondeur. Chaque site s_i est atteint par le parcours en profondeur et est donc ajouté au codage sous la forme d'un mot $i : L$. De plus chaque arête sortant de s_i vers un fils $s_{i'}$ est représentée par un mot de la forme $\xrightarrow{k} i'[: L]$ s'il suit directement le mot définissant le sommet ou $\xrightarrow{R} i \xrightarrow{k} i'[: L]$ dans le cas contraire. Lors de la reconstruction dans l'algorithme 8.9, le mot $i : L$ crée le site s_i et ce site s_i devient le sommet courant. S'il y a un premier fils, le mot $\xrightarrow{k} i'[: L]$ suit directement et crée une arête entre le sommet s_i courant et le sommet s'_i qui sera à son tour créé s'il n'existe pas. Les fils suivants de s_i, codés par des mots de la forme $\xrightarrow{R} i \xrightarrow{k} i'[: L]$, ramènent le sommet courant à s_i puis créent à nouveau une arête entre s_i et s'_i.

Nous avons montré que tous les sommets et les arêtes du site S se retrouvent dans le site recréé à partir du codage. Nous avons en fait utilisé tous les mots écrits par l'algorithme 8.7 de parcours en profondeur et ces mots n'ont pas d'autre effet dans la reconstruction effectuée par l'algorithme 8.9. Ainsi le site reconstruit est exactement le site S d'origine.

Proposition 8.7. *Le codage du parcours en largeur fourni par l'algorithme 8.8 à partir de n'importe quel site S est un codage de S.*

Preuve. *La preuve est similaire à celle de la proposition 8.6. Tout d'abord on montre que l'algorithme 8.8 fournit un codage de parcours bien formé à partir d'un site S de racine R. Le premier élément traité est R et conduit à l'ajout du mot $1 : L$ où L est l'étiquette de R. Il s'agit de la seule occurrence de ce mot car la variable p ne peut plus être égale à $\emptyset$. Le traitement de tout autre sommet conduit à l'ajout d'un mot de la forme $[\xrightarrow{R} i] \xrightarrow{k} i' : L$ ou d'un mot de la forme $[\xrightarrow{R} i] \xrightarrow{k} i'$. Le mot donné en sortie de l'algorithme est donc de la forme $m_1(m_2)^*$.*

On parcourt un site en largeur à partir de sa racine donc on visite chaque sommet exactement une fois et c'est lors de cette visite qu'on spécifie son étiquette : l'étiquette de chaque sommet est spécifiée une unique fois.

On n'ajoute un mot de la forme $\xrightarrow{R} i$ que lorsque l'on connaît le sommet père du sommet courant dans le parcours en largeur. Ce sommet père a forcément été visité au préalable et son étiquette a été spécifiée sous la forme d'un mot $i : L$.

À p et k fixé, il y a donc au plus un élément (p, k, s) ajouté à A lors du parcours. Pour tout mot de la forme $i[: L] \xrightarrow{k}$ présent dans la sortie de l'algorithme, l'ajout du mot contenant i a l'effet suivant : le sommet courant devient le sommet numéroté i. L'ajout du mot $\xrightarrow{k}$ intervient au traitement de l'élément (p, k, s) de A lorsque le sommet courant est p. Les éléments de la forme (p, k, s) sont ajoutés à A une unique fois lors du parcours du site p et ce site p a, pour chaque entier k, au maximum un fils s numéroté k. À un sommet p et un entier k est donc associé un unique mot de la forme $num(p) \xrightarrow{k}$.

Nous cherchons maintenant à prouver que pour tout site S le codage fourni par l'algorithme 8.8 de parcours en largeur permet de reconstruire, à l'aide de l'algorithme 8.9, le site S. Notons $s_1, s_2, ..., s_n$ les sommets de S numérotés selon l'algorithme de parcours en largeur. Chaque site s_i est atteint par le parcours en largeur et est donc ajouté au codage sous la forme d'un mot $i : L$. De plus chaque arête numéroté k sortant de s_i vers un fils $s_{i'}$ est représentée dans l'algorithme par l'ajout à A d'un élément $(s_i, k, s_{i'})$ qui provoque, lors de son traitement, l'ajout d'un mot de la forme $\xrightarrow{k} i'[: L]$ s'il suit directement le mot définissant le sommet ou $\xrightarrow{R} i \xrightarrow{k} i'[: L]$ dans le cas contraire. Lors de la reconstruction dans l'algorithme 8.9, le mot $i : L$ crée le site s_i et ce site s_i devient le sommet courant. S'il y a un premier fils, le mot $\xrightarrow{k} i'[: L]$ suit directement et crée une arête entre le sommet s_i courant et le sommet s'_i qui sera à son tour créé s'il n'existe pas. Les fils suivants de s_i, codés par des mots de la forme $\xrightarrow{R} i \xrightarrow{k} i'[: L]$, ramènent le sommet courant à s_i puis créent à nouveau une arête entre s_i et s'_i.

Nous avons montré que tous les sommets et les arêtes du site S se retrouvent dans le site recréé à partir du codage. Nous avons en fait utilisé tous les mots écrits par l'algorithme 8.8 de parcours en largeur et ces mots n'ont pas d'autre effet dans la reconstruction effectuée par l'algorithme 8.9. Ainsi le site reconstruit est exactement le site S d'origine.

On sait alors reconstruire un site depuis un codage de parcours bien formé et on dispose d'algorithmes de parcours fournissant des codages pour les sites que l'on étudie. On cherche maintenant à définir le parcours d'un site au sein d'un graphe de flot : si ce parcours est possible alors le site parcouru est un sous-site de ce graphe de flot. Ce parcours peut à nouveau être représenté sous la forme d'un codage de parcours bien formé.

L'algorithme 8.10 détermine (s'il existe) le sous-site désigné par un codage dans un graphe de flot T, à partir d'un sommet donné. S'il ne renvoie pas FAIL, il fournit un site qui est effectivement un sous-site du graphe (les sommets et les arcs sont forcément dans T, et ils sont accessibles à partir de la racine vu qu'il s'agit d'un parcours). On explicite à l'aide de cet algorithme le parcours d'un codage de parcours bien formé au sein d'un graphe de flot (définition 8.19).

On voudrait maintenant prouver l'équivalence entre l'existence d'un isomorphisme de sous-site entre un site P et un graphe de flot T d'une part, et la possibilité d'appliquer un parcours bien formé de P à partir d'un sommet de T d'autre part. Les deux propositions 8.8 et 8.9 vont permettre d'établir l'équivalence statuée au théorème 8.1.

Définition 8.19. *On dit que le codage C d'un parcours bien formé peut être parcouru dans un graphe de flot T si et seulement si il existe un sommet R de T tel que l'algorithme 8.10 appliqué à C, T et R renvoie un site. On appelle site résultant le site alors renvoyé par cet algorithme.*

Algorithme 8.10 : Reconstruction, depuis un codage de parcours bien formé, d'un sous-site de graphe de flot à partir d'un sommet donné du graphe de flot

Données : Un codage de parcours bien formé, un graphe de flot T, un sommet R de T

Résultat : Un site ou FAIL

```
reconstructionSousSite(codage, T, R) // Le codage est de la forme
1 : L(α→ i[: L])*
    si le sommet R a l'étiquette L alors
        On numérote R par 1
        Le sommet numéroté 1 devient le sommet courant
    sinon
        retourner FAIL
    fin
    pour chaque suite de symboles de type α→ i[: L] faire
        si α est un entier k alors
            si le sommet courant a un fils k non numéroté, d'étiquette L, et aucun
            sommet n'est numéroté i alors
                On numérote ce fils k par i
                On marque l'arc k entre le sommet courant et le sommet numéroté i
                Le sommet numéroté i devient le sommet courant
            sinon si le sommet courant a un fils k numéroté i alors
                On marque l'arc k entre le sommet courant et le sommet numéroté i
                Le sommet numéroté i devient le sommet courant
            sinon
                // Le parcours n'est pas possible si :
                // un autre sommet est déjà numéroté i, ou
                // si le sommet à numéroter a déjà une autre
                   numérotation, ou
                // si l'étiquette n'est pas la bonne, ou
                // s'il n'y a pas de fils k
                retourner FAIL
            fin
        sinon
            Le sommet i devient le sommet courant
        fin
    fin
    retourner le site constitué des sommets numérotés et des arcs marqués
```

Proposition 8.8. *Si P est un site isomorphe à un sous-site S d'un graphe de flot T alors tout codage de P peut être parcouru dans T à partir d'une racine de S.*

Preuve. *Soit P un site isomorphe à une sous-site S d'un graphe de flot T et C un codage de parcours bien formé de P. C est un codage de P donc l'application de l'algorithme 8.9 à C fournit le site P. Chacun des algorithmes 8.9 et 8.10 prennent le codage C en entrée et traitent les mots qui le composent un par un. Nous montrons qu'à la fin du traitement de chaque mot le sommet courant du site P dans l'algorithme 8.9 est le correspondant, au sens de l'isomorphisme de graphes de flot (définition 8.4), du sommet courant du site S dans l'algorithme 8.10. Comme l'algorithme 8.9 renvoie P et qu'il crée exactement un sommet par entier i présent dans les mots de la forme $\xrightarrow{\alpha} i[: L]$ de C, on peut associer chaque entier i du codage à son sommet correspondant, p_i dans P. Nous notons $s_1, ..., s_n$ les sommets de S tels que p_i est associé à s_i au sens de l'isomorphisme. En particulier p_i et s_i ont la même étiquette. Nous parcourons le graphe de flot T à l'aide de l'algorithme 8.10 à partir de C, et T et de s_1. Nous opérons une démonstration par récurrence sur le nombre n de mots traités, l'hypothèse de récurrence est la suivante : le traitement de jusqu'à n mots n'a pas provoqué d'erreur dans l'algorithme 8.10 et quels que soit $l \leq n$, si le sommet courant à la fin du traitement de l mots dans l'algorithme 8.9 est p_i alors celui dans l'algorithme 8.10 est s_i.*

C est un codage de parcours bien formé : il est de la forme $m_1(m_2)*$. Le traitement du mot $m_1 = 1 : L$ par l'algorithme 8.9 crée un site constitué de sa seule racine, le sommet p_1 d'étiquette L et p_1 devient alors le sommet courant. Parallèlement le traitement du mot m_1 par l'algorithme 8.10 à partir du sommet s_1 ayant la même étiquette que p_1, soit L, numérote s_1 par 1 et le s_1 devient le sommet courant. L'hypothèse de récurrence est donc vérifiée initialement.

Supposons que l'hypothèse de récurrence est vraie pour n. S'il n'y avait que n mots dans C les algorithmes s'arrêtent et l'algorithme 8.10 n'a pas rencontré d'erreur et renvoie bien un site : on a montré que C peut être parcouru dans T à partir de la racine de S.

Dans le cas contraire on s'intéresse au traitement du mot $n+1$ de type $m_2 = \xrightarrow{\alpha} i[: L]$. Plusieurs cas sont possibles.

- *α est un entier k et il n'existe pas de sommet i déjà ajouté dans l'algorithme 8.9 : dans l'algorithme 8.9 le sommet p_i d'étiquette L est ajouté et un arc d'étiquette k est créé entre le sommet courant p et le sommet p_i qui devient le sommet courant. Dans l'algorithme 8.10 le sommet courant s a nécessairement un fils numéroté k car P et S sont isomorphes de telle manière que p et s correspondent. Par définition d'un graphe de flot il y a au plus un fils de s numéroté k : il ne peut s'agir que de s_i dont l'étiquette est L. De plus aucun sommet ne peut être déjà étiqueté par i : cela signifierait qu'un mot précédent aurait déjà ajouté un sommet i dans l'algorithme 8.9. Nous sommes donc dans le premier cas de l'algorithme 8.10 : le fils k de s est numéroté i. Ce fils est le sommet s_i et devient le fils courant.*
- *α est un entier k et un sommet i a déjà été ajouté dans l'algorithme 8.9 : le codage de parcours étant bien formé le sommet i précédemment ajouté avait nécessairement l'étiquette L. Un arc d'étiquette k est ajouté entre le sommet courant p et le sommet p_i. Lorsque le sommet p_i a été ajouté dans l'algorithme 8.9, on était dans le cas précédent et le sommet s_i a également été ajouté dans l'algorithme 8.10. De plus*

on vient d'ajouter un arc numéroté k entre le sommet courant p et p_i. Comme P et S sont isomorphes un arc numéroté k existe entre le sommet courant s et s_i et il est unique. On est dans le second cas de l'algorithme 8.10 : le sommet s_i devient le sommet courant.

— *α est R : dans l'algorithme 8.9 le sommet p_i devient le sommet courant. Dans l'algorithme 8.10 le sommet précédemment numéroté i doit devenir le sommet courant. Un sommet s a été numéroté i depuis un mot précédent de la forme $\xrightarrow{k} i : L$ car C est un codage de parcours bien formé et ce mot conduit dans l'algorithme 8.9 à la création du sommet p_i et donc, selon l'hypothèse de récurrence, s est s_i.*

On a montré qu'aucun cas ne mène à une erreur dans l'algorithme 8.10 : le codage C peut être parcouru dans T à partir de la racine s_1 de S.

Proposition 8.9. *Soit C un codage de parcours bien formé, S_C son site associé par l'algorithme 8.9 et T un graphe de flot. Si C peut être parcouru dans T, en nommant S le site résultant, alors S_C et S sont isomorphes.*

Preuve. *Soit C un codage de parcours bien formé, S_C son site associé par l'algorithme 8.9 et T un graphe de flot. Supposons que C peut être parcouru d'un sommet R du graphe de flot T : on nomme S le site résultant de l'application de l'algorithme 8.10 à C, T et R. Le codage C est de la forme $m_1(m_2)^*$ et les algorithmes 8.9 et 8.10 traitent le codage C un mot (de type m_1 ou m_2) à la fois. Nous notons $C = w_1, ..., w_n$ où w_1 est un mot de type m_1 et w_i avec $i \geq 2$ des mots de type m_2. Nous allons montrer que pour chaque entier $1 \leq l \leq n$, le site $S_{C,l}$ reconstruit par l'algorithme 8.9 à partir du codage de parcours bien formé $C_l = w_1, ..., w_l$ est isomorphe au site S_l reconstruit par l'algorithme 8.10 à partir du même codage de parcours bien formé $w_1, ..., w_l$. Nous effectuons la preuve par récurrence à partir de l'hypothèse de récurrence suivante sur l à prouver pour $1 \leq l \leq n$: l'application des algorithmes 8.9 et 8.10 au codage de parcours bien formé C_l fournit deux sites isomorphes, les dernier sommets courants dans chaque des deux algorithmes correspondent dans l'isomorphisme et les numérotations effectuées par les deux algorithmes sont telles que, quel que soit i, le sommet numéroté i dans l'algorithme 8.9 correspond, au sens de l'isomorphisme, au sommet numéroté i dans l'algorithme 8.10.*

Prenons $l = 1$ et $C_1 = w_1 = 1 : L$. L'algorithme 8.9 crée un site $S_{C,1}$ réduit à une racine d'étiquette L, numérotée 1 et qui devient le sommet courant. L'algorithme 8.10 renvoie un site donc on est dans le cas où le sommet R a pour étiquette L : il est numéroté 1, devient le sommet courant et le site S_1 renvoyé est le site restreint au seul sommet R. Ces deux sites sont restreints à un unique sommet ayant la même étiquette : ils sont isomorphes. Les derniers sommets courants et les seuls sommets numérotés dans les deux algorithmes sont ce sommet unique : ils correspondent dans l'isomorphisme.

Prenons $1 \leq l \leq n-1$ et $C_{l+1} = w_1, ..., w_{l+1}$ en supposant l'hypothèse de récurrence vraie pour l. L'application de l'algorithme 8.9 à C_l fournit un site $S_{C,l}$ isomorphe au site S_l fourni par l'application de l'algorithme 8.10. Le sommet courant $s_{C,l}$ de $S_{C,l}$ correspond au sommet courant s_l de S_l. Traitons le mot $w_l = \xrightarrow{\alpha} i[: L]$. On distingue trois cas pour la forme du mot w_l.

— $w_{l+1} \xrightarrow{k} i : L$ *avec* k *un entier :* C *est un codage de parcours bien formé donc il s'agit de la première occurrence de* i *et l'algorithme 8.9 crée un sommet numéroté* i *d'étiquette* L *et un arc entre le sommet courant et ce sommet qui devient le sommet courant. L'algorithme 8.10, qui n'échoue pas et, car* C *est un codage de parcours bien formé, n'a pas encore numéroté de sommet* i *donc il est dans le premier cas de la boucle : le sommet courant a un fils* k *non numéroté, d'étiquette* L*, qui est numéroté* i *et devient le sommet courant. L'arc entre le sommet courant et le sommet numéroté* i *est marqué. Les deux sommets ont la même étiquette* L*, l'arc ajouté à* $S_{C,l+1}$ *est numéroté* k *et relie le sommet* $s_{C,l}$ *au nouveau sommet numéroté* i *tandis que l'arc marqué dans* T *et ajouté dans* S_{l+1} *est numéroté* k *et relie le sommet* s_l *au nouveau sommet numéroté* i *: les sites* $S_{C,l+1}$ *et* S_{l+1} *sont isomorphes, les nouveaux sommets courants ainsi que les sommets nouvellement ajoutés correspondent dans l'isomorphisme.*

— $w_l \xrightarrow{k} i$ *avec* k *un entier :* C *est codage de parcours bien formé donc un mot de la forme* $i : L$ *a été présent traité précédemment. Un sommet numéroté* i *est présent dans* $S_{C,l}$ *et le traitement du mot* w_l *provoque l'ajout d'un arc numéroté* k *entre le sommet courant et ce sommet numéroté* i *qui devient le sommet courant. Dans l'algorithme 8.10 on est dans le second cas de la boucle : le sommet courant a un fils* k *numéroté* i*. On ajoute alors un arc entre le sommet courant et ce sommet numéroté* i *qui devient le sommet courant. Par hypothèse de récurrence les sommets courants ainsi que les sommets numérotés* i *correspondent dans l'isomorphisme donc l'ajout d'un arc dans chacun des sites préserve l'isomorphisme et les nouveaux sommets courants correspondent.*

— $w_l \xrightarrow{R} i$ *: dans chacun des deux algorithme un sommet a déjà été numéroté* i *et devient le sommet courant. Par hypothèse de récurrence les sites, non modifiés, sont toujours isomorphes et les nouveaux sommets courants correspondent dans l'isomorphisme.*

On a montré que dans tous les cas l'hypothèse de récurrence est également vérifiée au rang $l + 1$.

Théorème 8.1. *Soit* P *un site et* T *un graphe de flot. Les trois propositions suivantes sont équivalentes.*

— *P est isomorphe à un sous-site de T.*
— *Tout codage de parcours bien formé de P peut être parcouru dans T.*
— *Il existe un codage de parcours bien formé de P qui peut être parcouru dans T.*

Preuve.

— *Si P est isomorphe à un sous-site de T, on peut appliquer la proposition 8.8, ce qui donne directement le résultat suivant : tout codage de parcours bien formé de P peut être parcouru dans T.*
— *Si tout codage de parcours bien formé de P peut être parcouru dans T alors l'algorithme 8.7 de parcours en profondeur fournit un codage de parcours bien formé de P (proposition 8.6) qui peut être parcouru dans T.*

— *S'il existe un codage de parcours bien formé de P que l'on peut parcourir dans T, alors on construit son site associé avec l'algorithme 8.9 et on applique la proposition 8.9 qui montre l'existence d'un sous-site de T isomorphe à P.*

Ce résultat permet de montrer l'équivalence entre un isomorphisme de sous-site et la possibilité d'un parcours en profondeur du codage du site (corollaire 8.1). L'équivalence tient également avec le parcours en largeur défini précédemment ou tout parcours pour lequel on dispose d'un algorithme fournissant, à partir d'un site, un codage de ce site.

Corollaire 8.1. *Soit P un site et T un graphe de flot. Les trois propositions suivantes sont équivalentes.*

— *P est isomorphe à un sous-site de T.*
— *Le codage de parcours en profondeur de P peut être parcouru dans T.*
— *Le codage de parcours en largeur de P peut être parcouru dans T.*

L'intérêt de cette équivalence est qu'elle donne un algorithme simple et bien plus efficace que celui d'Ullmann pour résoudre le problème 8.2 d'isomorphisme de sous-site entre un site de motif et un graphe de flot de test.

Application directe

Résolution des problèmes d'isomorphisme. L'équivalence entre isomorphisme et parcours permet de résoudre le problème 8.1 de la manière suivante : deux sites sont isomorphes si et seulement si le codage de parcours en profondeur de l'un [1] peut être parcouru dans l'autre.

Le problème 8.2 de l'existence d'un isomorphisme entre un site et un sous-site d'un site peut être résolu ainsi : il existe un sous-site de T isomorphe au site P si et seulement si il existe un sommet de T à partir duquel le codage du parcours en profondeur[1] de P est possible.

Une solution au problème 8.3 de l'existence, dans une base L_S, d'un site isomorphe à un sous-site d'un site T consiste à itérer la solution précédente de l'isomorphisme de sous-site à chaque site de la base L_S.

Complexité. La résolution du problème de l'isomorphisme de sous-site avec cet algorithme se réalise dans le pire des cas en $O(n_T.n_P)$ puisqu'elle nécessite :

— un parcours en profondeur du site de motif P avec détermination du codage : $O(n_P)$,
— un parcours du graphe de flot T à partir de chacun de ses sommets : chaque parcours se fait en $O(n_P)$, cette étape peut alors se réaliser en $O(n_T.n_P)$.

Si on a une base L_S contenant S sites de taille W, la complexité dans le pire des cas pour déterminer un site de la base est isomorphe à un sous-site de T est $O(S.n_T.W)$.

1. On peut substituer le parcours en profondeur par un parcours en largeur ou n'importe quel parcours fournissant un codage d'un site.

Codages rangés dans un arbre de décision

Création d'un arbre de décision des codages. On peut générer les codages de parcours en profondeur de chaque site de la base puis les ranger dans un arbre de décision. Par exemple on donne quatre sites en figure 8.12, dont les parcours en profondeur sont :

(a) — $1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{1} 3 : INST \xrightarrow{1} 4 : RET \xrightarrow{R} 2 \xrightarrow{2} 5 : RET \xrightarrow{R} 1 \xrightarrow{2} 6 : JCC \xrightarrow{2} 7 : INST$

(b) — $1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{1} 3 : INST \xrightarrow{1} 4 : RET \xrightarrow{R} 2 \xrightarrow{2} 5 : RET \xrightarrow{R} 1 \xrightarrow{2} 6 : JCC \xrightarrow{2} 7 : INST \xrightarrow{1} 2$

(c) — $1 : JCC \xrightarrow{1} 2 : CALL \xrightarrow{1} 3 : INST \xrightarrow{1} 4 : RET \xrightarrow{R} 2 \xrightarrow{2} 5 : RET \xrightarrow{R} 1 \xrightarrow{2} 6 : JCC \xrightarrow{2} 7 : INST \xrightarrow{1} 3$

(d) — $1 : JCC \xrightarrow{1} 2 : RET \xrightarrow{R} 1 \xrightarrow{2} 3 : JCC \xrightarrow{2} 4 : INST \xrightarrow{1} 5 : CALL \xrightarrow{1} 6 : INST \xrightarrow{1} 2 \xrightarrow{R} 5 \xrightarrow{2} 7 : RET$

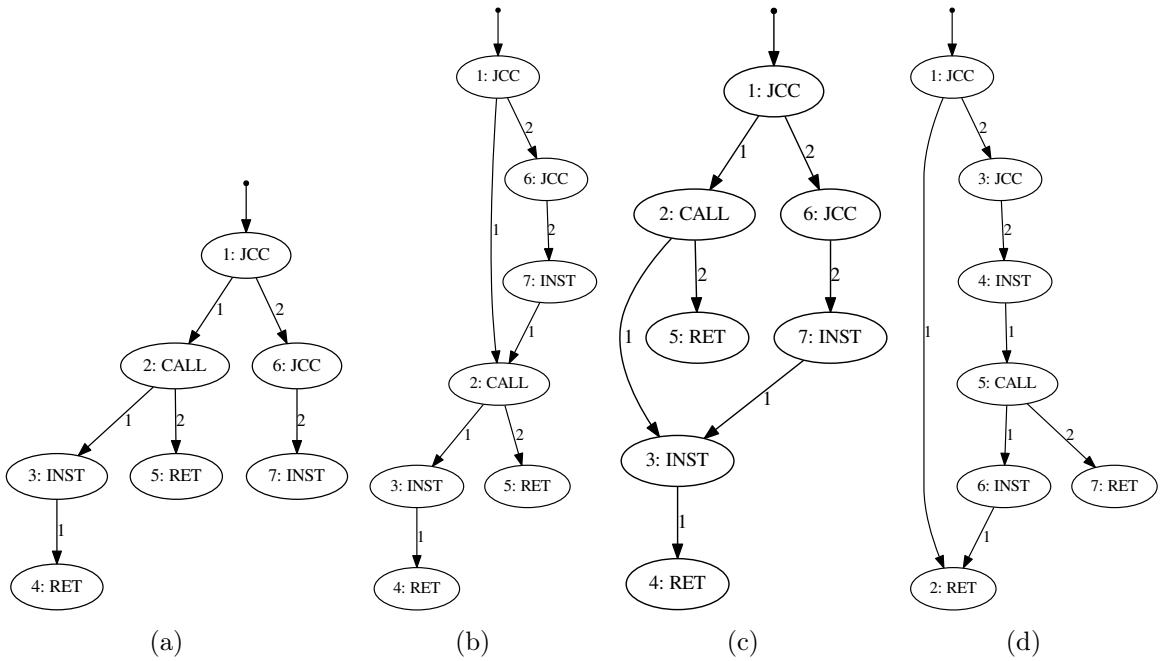


FIGURE 8.12 – Trois sites numérotés selon un parcours en profondeur

L'arbre de décision créé à partir de ces trois codages est donné en figure 8.13. L'ajout d'un codage à l'arbre de décision est standard et ne sera pas détaillé. Un parcours est terminé par un sommet final ayant une étiquette unique identifiant ce parcours. Il est à noter que les sommets finaux ne sont pas nécessairement tous des feuilles de l'arbre de décision.

L'ajout d'un site à la base consiste à ajouter un codage de parcours dans l'arbre de décision, cette opération nécessite un nombre de comparaison en $W \times f$ où f est le nombre de fils maximum de chaque sommet de l'arbre. Vu que les étiquettes des arcs de l'arbre sont de la forme $[\xrightarrow{\alpha}]i[: L]$, en notant $k = 2$ le nombre de fils maximum de chaque sommet

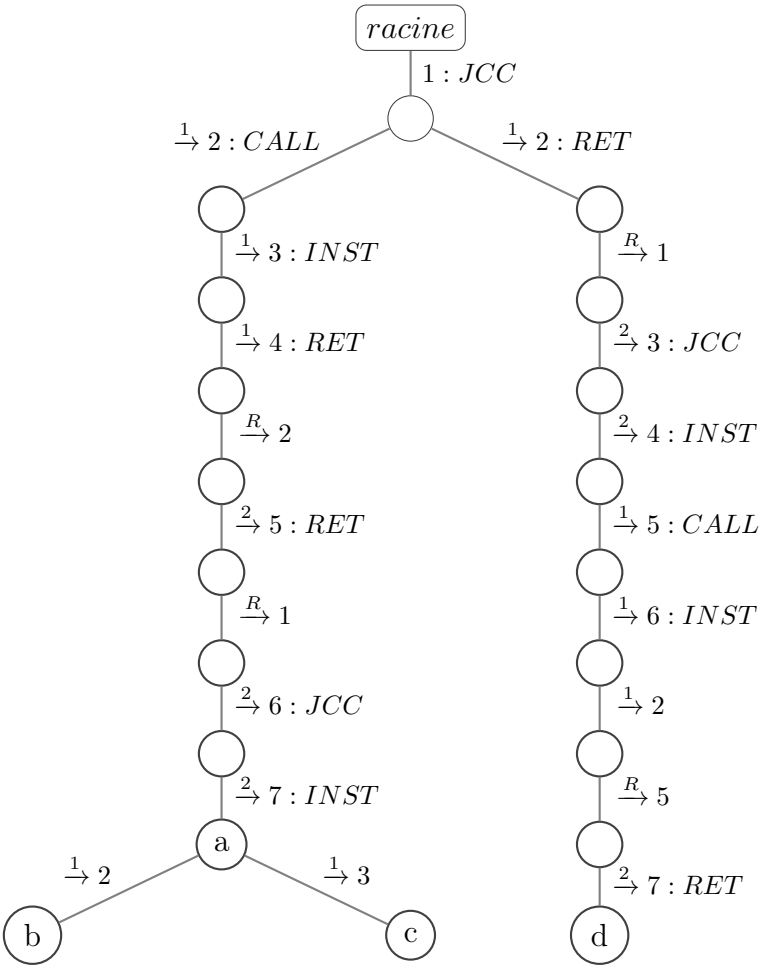


FIGURE 8.13 – Arbre de décision contenant les codages de sites de la figure 8.12

dans un site et $l \leq 10$ le nombre d'étiquettes différentes pour les sommets des sites, $f \leq (k+2) \times W \times l$, l'ajout d'un site à la base se fait en $O(W^2)$ (proposition 8.10).

Proposition 8.10. *La complexité de l'ajout d'un site de taille W à une base, pour l'algorithme par parcours, se fait en $O(W^2)$.*

Recherche des isomorphismes de sous-sites à partir de l'arbre de décision. Une fois l'arbre généré à partir de tous les codages des sites de L_S, on veut l'utiliser pour trouver tous les parcours de L_S possibles dans le graphe de flot T. Pour déterminer les parcours de l'arbre que l'on peut réaliser à partir d'un sommet r de T, on peut chercher à parcourir T à partir de r en parcourant les branches possibles dans l'arbre de décision. À chaque fois que l'on peut parcourir une branche jusqu'à un sommet final c'est qu'un site isomorphe a été trouvé.

L'algorithme 8.11 liste ces sommets finaux. Il est à noter qu'un sommet final est associé à un unique parcours : celui constitué par ce sommet et l'ensemble de ses ancêtres jusqu'à la racine. Ainsi cet algorithme renvoie une liste contenant les parcours de l'arbre de décision qu'il est possible de réaliser dans T à partir du site r.

Dans le pire des cas il y a S branches à l'arbre A qui partent de la racine où S est le nombre de sites dans la base. Le parcours d'une branche avec au maximum un fils à chaque sommet, à partir de la racine se fait en $O(W)$ au vu de la récursion de l'algorithme 8.11 car à chaque profondeur l'appel de la fonction *etape* se fait en $O(1)$ et la concaténation de deux listes (notée @) se fait également en temps constant. Parcourir toutes les branches a donc une complexité de $O(S.W)$. La complexité totale dans le pire des cas pour résoudre le problème 8.3 d'isomorphisme de sous-sites entre un graphe de flot T de taille n_T et une base de sites consiste à itérer cette recherche pour chaque sommet de T et ensuite à faire l'union des ensembles résultants.

La complexité totale est alors de $O(n_T.S.W)$ (proposition 8.11). Ainsi on n'a pas amélioré la complexité dans le pire des cas par rapport à l'utilisation directe de l'approche par parcours vue dans la section précédente.

Proposition 8.11. *La complexité dans le pire des cas de la recherche des sites (de taille W) de la base, construite avec algorithme par parcours, au sein d'un graphe de flot T de taille n_T, se fait en $O(S.n_T.W)$.*

Algorithme 8.11 : Liste, au sein d'un arbre de parcours, les parcours possibles dans un graphe de flot à partir d'un sommet

Données : Un arbre A contenant les parcours de sites, un graphe de flot T, un sommet r de T

Résultat : Une liste des sommets finaux de l'arbre qu'il est possible d'atteindre à partir de r

```
listeParcours(A, a, T, R)
    (finaux, s) ← ([], R)
    si a est un sommet final de A alors
        finaux ← [a]
    fin
    pour f fils de mot m de a dans A faire
        (possible, s', T') ← etape(m, s, T)
        si possible alors
            finaux ← finaux @ listeParcours(A, f, T', s')
        fin
    fin
    retourner finaux
etape(m, s, T) // Le mot m est de la forme [α→]i[: L]
    si m est de la forme 1 : L et s a pour étiquette L alors
        Numéroter le sommet s de T par 1.
        retourner (true, s, T)
    sinon
        si α est un entier k alors
            si s a un fils s' d'ordre k non numéroté dans T, d'étiquette L, et aucun
            sommet n'est numéroté i alors
                On numérote s' par i dans T.
                retourner (true, s', T)
            sinon si le sommet courant a un fils s' d'ordre k numéroté i alors
                retourner (true, s', T)
            sinon
                // Si un autre sommet est déjà numéroté i, ou
                // si le sommet à numéroter a déjà une autre
                   numérotation, ou
                // si l'étiquette n'est pas la bonne, ou
                // s'il n'y a pas de fils k
                retourner (false, s, T)
            fin
        sinon m est de la forme R→ i
            On note s' le sommet numéroté i dans T.
            retourner (true, s', T)
        fin
    fin
```

8.3.2 SIDT : Site Isomorphism Decision Tree

Le problème 8.2 consiste à détecter un site P dans un graphe de flot T. En pratique le site P aura été généré à partir d'un graphe de flot G. Si P est un sous-site de T, puisque l'on a généré le site à partir d'un parcours dans un autre graphe de flot, on peut envisager que le même parcours appliqué à T est susceptible de générer le site P. Ce n'est pas vrai dans le cas général et en ce sens l'algorithme présenté ici n'est pas complet. Inversement il est clair que si le même parcours appliqué à partir de sommets de T et G génère le même site P, alors P est bien un sous-site de T.

On se propose donc, pour approcher la solution au problème 8.2 de détection d'isomorphisme de sous-site entre un site P (généré par parcours avec un algorithme A à partir d'un sommet d'un graphe de flot G) et un graphe de flot T, de générer tous les sous-sites de T avec l'algorithme de parcours A et de regarder si P est dans cet ensemble.

Nous reprenons l'exemple de fonctionnement du détecteur par analyse morphologique décrit à la section 8.1.4. Le corpus est constitué du graphe de flot G découpé en sites donnés en figure 8.14. Afin de détecter le graphe de flot donné à la figure 8.15, au lieu de vérifier si les sites P sont isomorphes à des sous-sites de T, nous découpons T en sites en suivant la même méthode que pour déterminer les sites à ajouter au corpus. Nous comparons ensuite les sites générés à partir de T à ceux présents dans le corpus : il y a une correspondance si on trouve des sites identiques. Ici nous voyons que les sites 1 et α sont identiques. C'est également le cas des sites 4 et ϵ. Ces correspondances sont donc détectées par cette approche. En revanche le site 3, pourtant sous-site du graphe T, n'est pas présent dans la liste des sites générés à partir de T par le parcours en largeur de taille limitée.

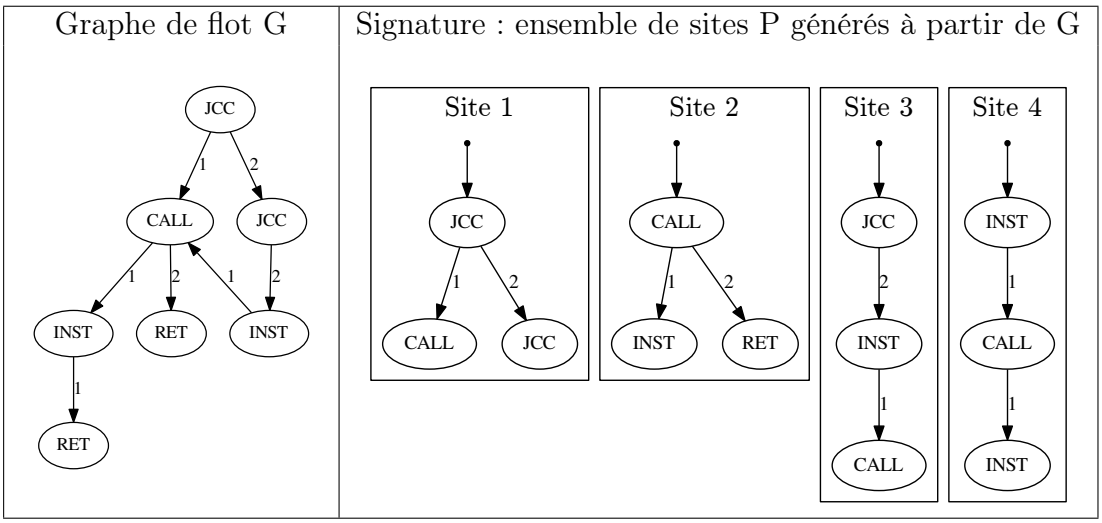


FIGURE 8.14 – Graphe de flot d'un programme et sa signature dans le corpus

Pour régler le problème 8.3 de détection d'un site dans une base de graphes de flot, on va créer un arbre de décision contenant tous les sites générés à partir du parcours en largeur des graphes de flot de la base.

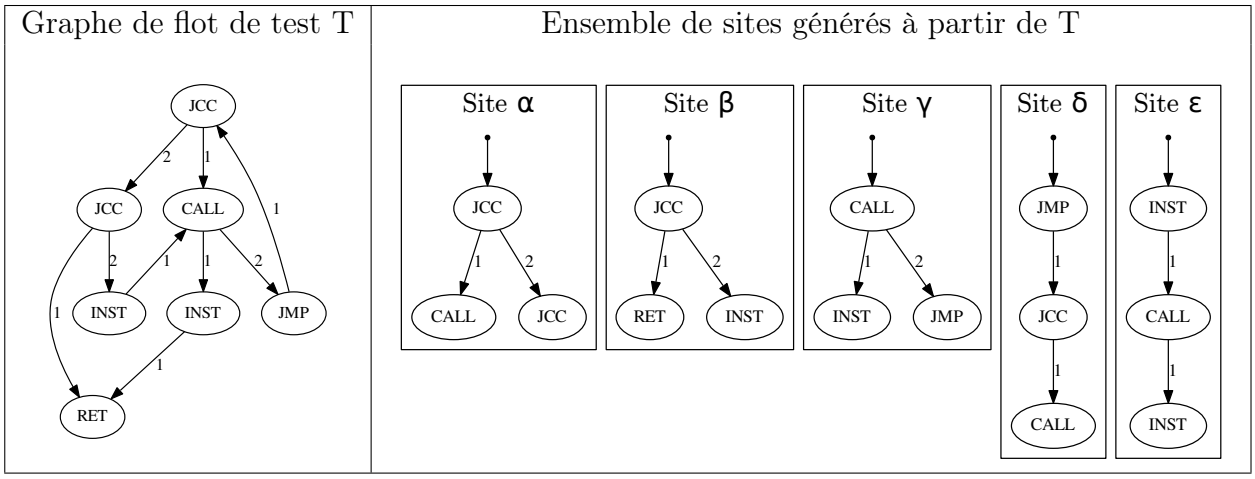


FIGURE 8.15 – Graphe de flot d'un programme de test et son découpage en sites

Arbre de décision pour la détection de sites

Chaque site est généré par un parcours numérotant ses sommets et chaque sommet a un nombre de fils bornés. On représente alors les sites sous forme matricielle : chaque ligne représente un sommet, le premier élément de la colonne est l'étiquette du sommet et les éléments suivants sont les numéros des fils de ce sommet (0 indique qu'il n'y a pas de sommet). La représentation des sites de la figure 8.12 est donnée figure 8.16.

Un site généré par un parcours a une unique représentation sous cette forme parce que, d'une part le parcours a numéroté ses sommets qui sont donc ordonnés, d'autre part les fils de chaque sommet sont numérotés, enlevant toute ambiguïté sur l'ordre à donner aux fils dans la matrice générée.

On peut ensuite agréger tous les sites générés dans un arbre de décision (figure 8.13). L'apprentissage d'un site nécessite le parcours d'au plus W éléments de l'arbre de décision. Chaque sommet de l'arbre de décision a au plus $l.W.W$ fils où l est le nombre d'étiquettes possibles. L'apprentissage d'un site se fait donc en $O(W^3)$ où W est le nombre de sommets des sites à apprendre (proposition 8.12).

Proposition 8.12. *La complexité de l'ajout d'un site de taille W à une base, pour SIDT, se fait en $O(W^3)$.*

Sommet	*Etiquette*	*Fils*1	*Fils*2
1	*JCC*	2	6
2	*CALL*	3	5
3	*INST*	4	0
4	*RET*	0	0
5	*RET*	0	0
6	*JCC*	0	7
7	*INST*	0	0

(a)

Sommet	*Etiquette*	*Fils*1	*Fils*2
1	*JCC*	2	6
2	*CALL*	3	5
3	*INST*	4	0
4	*RET*	0	0
5	*RET*	0	0
6	*JCC*	0	7
7	*INST*	2	0

(b)

Sommet	*Etiquette*	*Fils*1	*Fils*2
1	*JCC*	2	6
2	*CALL*	3	5
3	*INST*	4	0
4	*RET*	0	0
5	*RET*	0	0
6	*JCC*	0	7
7	*INST*	3	0

(c)

Sommet	*Etiquette*	*Fils*1	*Fils*2
1	*JCC*	2	3
2	*RET*	0	0
3	*JCC*	0	4
4	*INST*	5	0
5	*CALL*	6	7
6	*INST*	2	0
7	*RET*	0	0

(d)

FIGURE 8.16 – Représentation matricielle des sites de la figure 8.12

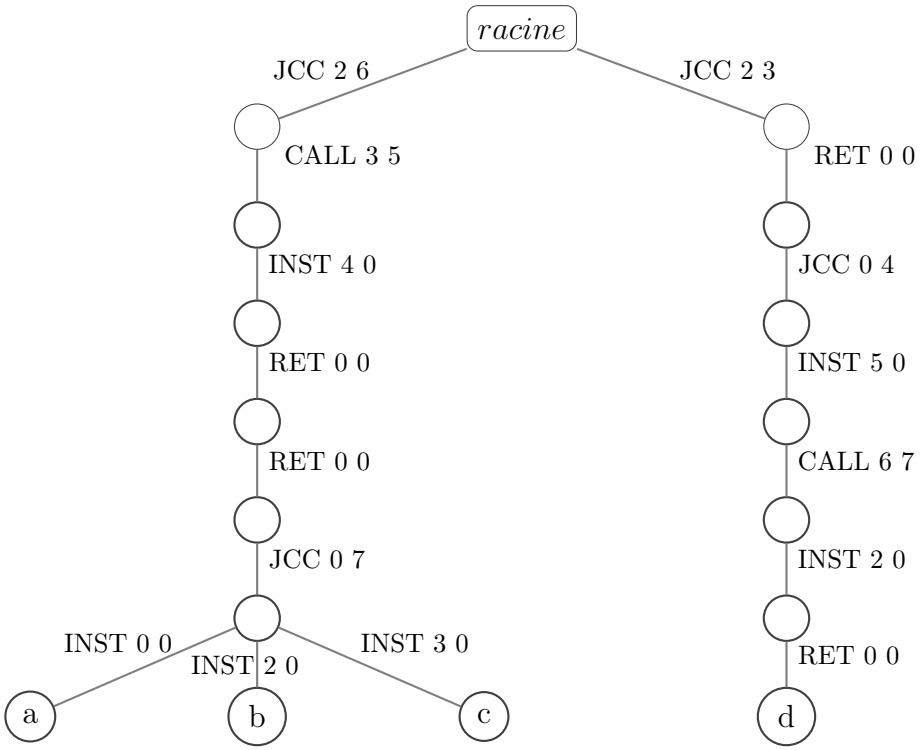


FIGURE 8.17 – Arbre de décision contenant les matrices de la figure 8.16

Recherche d'isomorphismes de sous-sites

Rechercher un isomorphisme de sous-sites à partir de l'arbre de décision ainsi créé revient à écrire le site sous forme matricielle et à faire une recherche dans l'arbre de décision. L'algorithme 8.12 renvoie, à partir d'une liste de sites provenant d'un graphe à tester, une liste contenant les feuilles de l'arbre de décision atteintes lors de la recherche. Chaque feuille est associée à un unique site et donc l'algorithme permet de résoudre, de manière incomplète, le problème 8.3 de recherche d'isomorphismes de sous-sites.

Algorithme 8.12 : Liste les sites d'un arbre de décision qui sont présents dans un graphe

Données : Un arbre A contenant les sites appris, de racine r, un ensemble M de sites sous forme matricielle

Résultat : Une liste des feuilles de l'arbre de décision atteignables à partir d'un des sites de M

```
decompteSIDT(A, r, M)
    feuilles ← ∅
    pour site m ∈ M sous forme matricielle faire
        feuilles ← feuilles ∪ trouver(A, r, m, 1)
    fin
    retourner feuilles
trouver(A, a, m, i)
    si a n'a aucun fils dans A alors
        // Si a n'a aucun fils, on a trouvé un site identique
        retourner {a}
    sinon
        pour f fils de mot m de a dans A faire
            si la i^e ligne de m est égale à l'étiquette de f alors
                retourner trouver(A, f, m, i + 1)
            fin
        fin
        retourner ∅
    fin
```

La complexité de la recherche d'un site dans l'arbre de décision est la même que pour l'ajout dans d'un site dans l'arbre et se fait en $O(W^3)$. Ainsi l'analyse de n sites se fait en $O(n.W^3)$ (proposition 8.13).

Proposition 8.13. *La complexité dans le pire des cas de la recherche de n sites (de taille W) de la base contenant S sites, construite avec algorithme par parcours se fait en $O(n.W^3)$.*

8.4 Comparaison des performances

8.4.1 Complexité

On dispose donc de quatre algorithmes à comparer : celui par recherche exhaustive avec les matrices d'adjacence, l'algorithme d'Ullmann, pris pour référence, l'algorithme par parcours avec arbre de décision et l'algorithme incomplet par arbre de décision (SIDT). Nous calculons les complexités dans le pire des cas de l'ajout d'un graphe de flot dans la base et d'analyse d'un graphe de flot inconnu. On dispose donc d'une part d'un corpus de programmes dont les graphes de flot ont été déterminés. Nous notons :

- S le nombre de sites du corpus.
- n le nombre de sommets du graphe de flot que l'on cherche à ajouter ou analyser.
- W le nombre de sommets des sites de la base (constante fixée typiquement entre 12 et 24).
- h la constante introduite à la section 8.2.2 sur l'algorithme d'Ullmann, comprise entre 0 et 1, dépendant de la répartition des étiquettes, et évaluée à 0.34.
- k la valence maximale dans les graphes de flot de contrôle, c'est à dire le nombre maximum de fils qu'un sommet peut avoir ; $k = 2$.

D'autre part nous étudions deux opérations sur cette base. La première est l'ajout d'un graphe de flot, à n sommets, à cette base de reconnaissance. La seconde est l'analyse d'un graphe de flot inconnu, comprenant n sommets. Il y a donc S sites de taille W dans la base.

Algorithme par parcours. La complexité pour déterminer tous les sites d'un graphe de flot à n sommets est $O(n.(k+1).W)$. Ainsi la complexité de l'ajout pour l'algorithme par parcours est $O(n.(k+1).W + n.W^2)$, soit $O(n.W^2)$. De même l'analyse d'un graphe de flot a pour complexité $O(n.(k+1).W + S.n.W)$, soit $O(S.n.W)$

SIDT. La complexité de l'ajout pour SIDT est $O(n.(k+1).W + n.W^3)$, soit $O(n.W^3)$ et celle pour l'analyse est $O(n.(k+1).W + n.W^3)$, soit $O(n.W^3)$.

Liste de sites et unicité des résultats. Tous les algorithmes étudiés précédemment ainsi que les complexités calculées permettent d'obtenir une liste contenant tous les sites de la base (sauf SIDT qui est incomplet) isomorphes à un sous-site de T. Afin de pouvoir détecter des programmes malveillants nous préférons en général connaître l'ensemble des sites uniques qui sont isomorphes à un sous-site de T et sa taille. Par exemple si les sommets finaux atteints dans le graphe de décision de l'algorithme par parcours sont $[c, a, c, b]$, on cherche à récupérer l'ensemble $\{a, b, c\}$ contenant 3 éléments.

Obtenir ces sites uniques nécessite d'ajouter les éléments récupérés à un ensemble que l'on peut par exemple implémenter à l'aide d'un arbre bicolore. L'ajout d'un élément dans un tel arbre contenant m éléments se fait en $O(log_2(m))$.

Les algorithmes par recherche exhaustive et d'Ullmann cherchent pour chaque élément de S un isomorphisme de sous-graphe entre l'élément de S et T puis ajoutent, si le résultat

est positif, cet élément à l'ensemble : la complexité totale dans le pire des cas ajoutée par la récupération des sites uniques est celle d'un tri, c'est à dire $O(S.log_2(S))$.

L'algorithme par parcours détermine pour chaque sommet de T une liste de parcours possibles de taille maximale S. Le nombre maximal d'éléments à insérer dans l'ensemble est donc $n.S$ avec au plus S éléments uniques : la complexité totale ajoutée est $O(n.S.log_2(S))$.

SIDT détermine pour chaque sommet de T au plus un site possible. On doit donc insérer n éléments dans un ensemble de taille n au maximum : la complexité totale ajoutée est $n.log_2(n)$.

La complexité totale des opérations de détection permettant de récupérer les solutions uniques est donnée dans le tableau 8.18.

Algorithme	Analyse d'un graphe de flot à n sommets
Recherche exhaustive	$O(S.(n^3.n! + log_2(S)))$
Ullmann	$O(S.(W.(n.h)^W + log_2(S)))$
Parcours	$O(n.S.(W + log_2(S)))$
SIDT	$O(n.(W^2 + log_2(n)))$

FIGURE 8.18 – Complexité de la détection dans le pire des cas en récupérant les éléments uniques

En pratique un programme contient typiquement 10000 sommets dans son graphe de flot réduit et on peut donc en extraire au maximum 10000 sites. Si la base contient un million de programmes, ce qui est une borne largement supérieure au nombre de programmes que l'on a en général dans nos bases de programmes malveillants, il y a au plus 10 milliards de sites et le logarithme en base 2 de 10 milliards vaut 43. Même dans le cas de cette borne supérieure le terme $log_2(S)$ est négligeable pour les algorithmes par recherche exhaustive et celui d'Ullmann. Il est de l'ordre de W et nous considérons qu'il n'a pas d'influence significative sur le temps de calcul pour l'algorithme par parcours. Le facteur logarithmique pour SIDT est $log_2(n)$ et donc même pour un programme ayant des millons de sommets dans son graphe de flot réduit, $log(n)$ serait de l'ordre de 20, ce qui est négligeable par rapport au terme en W^3. Dans la suite nous présentons les calculs de complexité sans ce terme logarithmique et évaluons notre hypothèse de non influence pour l'algorithme par parcours lors des expériences.

Complexité totale

Si l'on ne prend plus en compte le temps pour déterminer les solutions uniques, la complexité totale de chacun des algorithmes pour ces opérations est donnée par le tableau donné en figure 8.19. Les deux premiers algorithmes ne rangent pas les sites dans une base spécifique ; on dispose simplement d'une liste de sites que l'on doit générer à partir du graphe de flot.

Algorithme	Ajout d'un graphe de flot	Analyse d'un graphe de flot
Recherche exhaustive	$O(n.W)$	$O(S.n^3.n!)$
Ullmann	$O(n.W)$	$O(S.(n.h)^W)$
Parcours	$O(n.W^2)$	$O(S.n.W)$
SIDT	$O(n.W^3)$	$O(n.W^2)$

FIGURE 8.19 – Complexité dans le pire des cas

Par la suite nous exploiterons le fait que W et h sont des constantes.

Complexité à nombre de sites dans la base constant

Nous fixons le nombre de site dans le base, S. La complexité de l'ajout et de l'analyse devient celle donnée en figure 8.20.

Algorithme	Ajout d'un graphe de flot	Analyse d'un graphe de flot
Recherche exhaustive	$O(n)$	$O(n^3.n!)$
Ullmann	$O(n)$	$O(n^W)$
Parcours	$O(n)$	$O(n)$
SIDT	$O(n)$	$O(n)$

FIGURE 8.20 – Complexité à W, P, S et h fixés

Ce tableau nous indique que, bien que l'on ait grandement réduit la complexité initiale de l'algorithme d'Ullmann, l'utilisation de l'un des deux premiers algorithmes augmente en complexité d'un facteur d'au moins n^{12} avec la taille du graphe à analyser.

Les deux autres algorithmes ont eux une complexité augmentant linéairement avec la taille du graphe à ajouter ou à analyser.

Influence de la taille de la base

Les trois premiers algorithmes ont, pour l'analyse de graphe de flot inconnu, une complexité linéaire en le nombre de sites de la base, S. L'algorithme SIDT est le seul pour lequel le temps de l'analyse ne dépend pas du nombre de sites dans la base, ce qui le rend à même de mieux gérer un corpus de grande taille.

8.4.2 Détail des implémentations

Nous avons implémenté différents algorithmes de la manière suivante. Ils prennent en entrée un ou plusieurs graphes de flot à insérer dans la base de programmes connus et un ou plusieurs graphes de flot à tester. Le programme donne en sortie, pour chaque graphe de flot à tester, les correspondances trouvées avec ceux ajoutés à la base. Les graphes de flot sont stockés dans des fichiers de format binaire ou des fichiers de graphes (`.dot`).

Algorithme d'Ullmann. Développé en 2000 lignes C [Thib], l'implémentation offre les différentes options d'optimisation dont celles discutées précédemment à la section 8.2.2. Elle est principalement constituée des deux fonctions *backtrack* et *forwardChecking*.

Arbre de décision sur l'algorithme par parcours. Développée en 1000 lignes de C++ [Thia], l'implémentation permet de créer une base à partir d'une liste de graphes de flot et de tester une liste de graphes de flot. Elle ne permet pas d'enregistrer la base dans un fichier.

SIDT. Développée en 1200 lignes de C [Thic], l'implémentation permet également de créer une base et de tester une liste de graphes de flot. Elle ne permet pas non plus d'enregistrer la base dans un fichier.

8.4.3 Performances des implémentations

Machine de test. Nous avons réalisé nos tests sur un ordinateur portable DELL Latitude E6510 doté d'un processeur Intel i5-560M à double cœur fonctionnant à 2.66 GHz.

Programmes d'empaquetage. Nous avons utilisé l'analyse morphologique lors de l'expérience sur les programmes de protection du chapitre 6 dont les résultats sont donnés en annexe A. Nous avons cherché à détecter les sites extraits du GFC de `hostname.exe` que l'on retrouve dans des versions protégées de ce même binaire.

Conformément à nos attentes les algorithmes d'Ullmann et l'algorithme par parcours nous donnent les mêmes résultats en termes de détection sur deux exemples choisis (tElock99 et UPX). L'algorithme d'Ullmann étant lent, nous n'avons pas mené à terme la détection sur tous les échantillons. SIDT donne des résultats rapides mais incomplets. Nous résumons ces résultats pour l'ensemble des échantillons protégés dans le tableau suivant.

Algorithme	Apprentissage	Détection	Sites uniques détectés	Complet
Ullmann	∞	∞	(6739)	Oui
Parcours	$<$ 1 seconde	70 secondes	6739	Oui
SIDT	$<$ 1 seconde	1.6 seconde	6045	Non

Complétude. Sur l'exemple de `hostname.exe` empaqueté 694 sites sur 6739 ne sont pas détectés par SIDT, c'est à dire environ 10% de faux négatifs. Pourtant, avec l'empaqueteur pelock, on passe de 214 sites détectés à seulement 123 : dans certains cas il y a une différence flagrante de précision dans la détection.

Apprentissage et détection de sites. Nous voulons évaluer plus généralement les performances pour l'apprentissage et la détection de l'algorithme par parcours et de l'algorithme SIDT. Le but est principalement de valider les résultats théoriques indiquant que l'apprentissage de n sites se fait dans les deux cas en $O(n)$ et que la détection d'un nombre fixé de sites dépend du nombre S de sites uniques dans la base et se fait en $O(S)$ pour l'algorithme par parcours mais ne dépend pas de S dans le cas SIDT.

La base d'apprentissage contient 100 programmes totalisant 167404 sites dont 72482 sont uniques. Lors de l'apprentissage on s'intéresse indifféremment à l'apprentissage de sites déjà présents comme de sites inédits. Par contre pour l'expérience sur la détection, on considère que S est le nombre de sites uniques dans la base.

La figure 8.21 donne le temps d'apprentissage en fonction du nombre de sites que l'on apprend. On observe bien la croissance linéaire attendue et on constate que SIDT est beaucoup plus rapide lors de l'apprentissage.

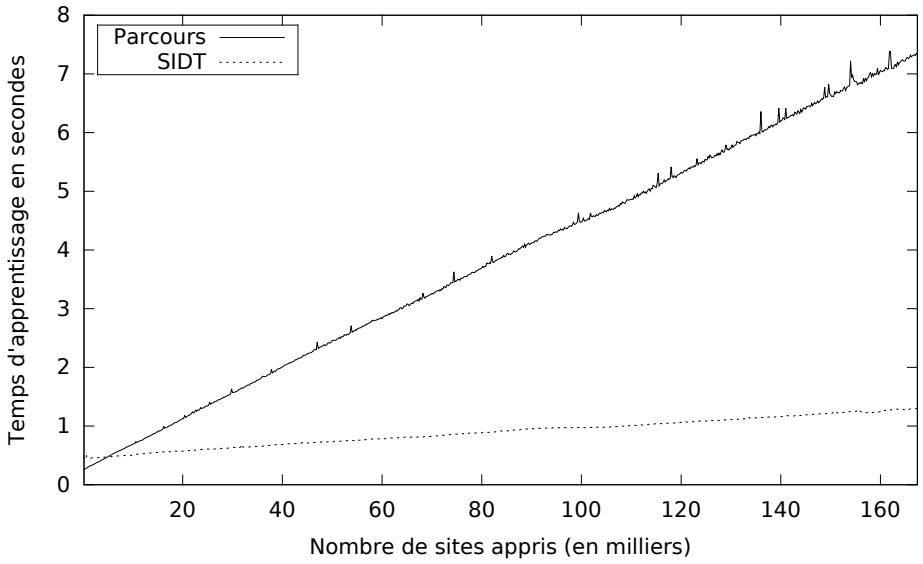


FIGURE 8.21 – Temps d'apprentissage en fonction du nombre de sites

La figure 8.22 donne le temps de détection de 5 programmes (qui sont composés de 18160 sites en tout) en fonction du nombre de sites dans la base. Sur cette figure le temps de détection pour SIDT semble ne pas dépendre du nombre de sites dans la base tandis que pour l'algorithme par parcours il semble suivre une courbe à croissance plus lente qu'une courbe linéaire en fonction du nombre de sites dans la base. Cette seconde observation peut sembler raisonnable si l'on prend en compte que la dépendance linéaire calculée dans la section précédente est la complexité dans le pire des cas et que l'ajout des parcours à un arbre de décision factorise en général certains parcours.

Influence de la détermination des sites uniques. Nous avons cherché à évaluer l'influence de l'étape de tri des résultats afin d'obtenir un ensemble de sites uniques lors de la détection avec l'algorithme par parcours. Nous avons calculé le temps de détection pour l'algorithme par parcours cette fois-ci en ne récupérant pas les éléments uniques : nous obtenons alors une liste de sites dans laquelle il peut y avoir des doublons. Il y a au maximum 2896 sites uniques détectés ce qui nécessite l'insertion dans un ensemble contenant au plus 2896 éléments : le retrait de cette étape ne crée pas une différence perceptible dans le temps de détection. Nous considérons donc que cette distinction n'a pas d'influence en pratique lors de l'étape de détection.

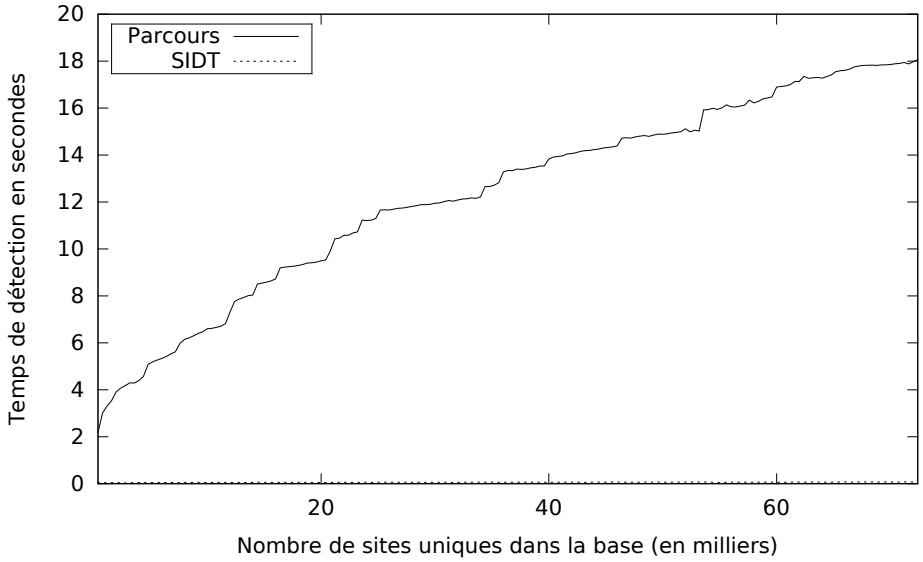


FIGURE 8.22 – Temps de détection de 5 programmes (18160 sites en tout) en fonction du nombre de sites dans la base

Apprentissage et détection sur un grand nombre de sites. Forts de cette première expérience, pour valider la dépendance temporelle plus amortie qu'une dépendance linéaire en fonction du nombre de sites de la base lors de la détection avec l'algorithme par parcours, nous avons utilisé un second jeu de données. Nous avons 17044 programmes pour l'apprentissage, contenant 4452219 sites uniques. Notre jeu de données pour la détection contient 5 programmes totalisant 29145 sites uniques.

Avec l'algorithme SIDT le temps d'apprentissage de l'ensemble de ces programmes est de 335 secondes, soit 5 minutes et 30 secondes et le temps de détection de 5 programmes est toujours inférieur à une seconde.

Une partie des résultats de l'algorithme par parcours en termes de vitesse de détection est donnée à la figure 8.23. Nous observons que la courbe a une croissance intermédiaire entre une croissance linéaire (donnée par le pire des cas) et ce qui semble être une croissante logarithmique. Le temps de détection dépend fortement du nombre de sites uniques appris dans la base.

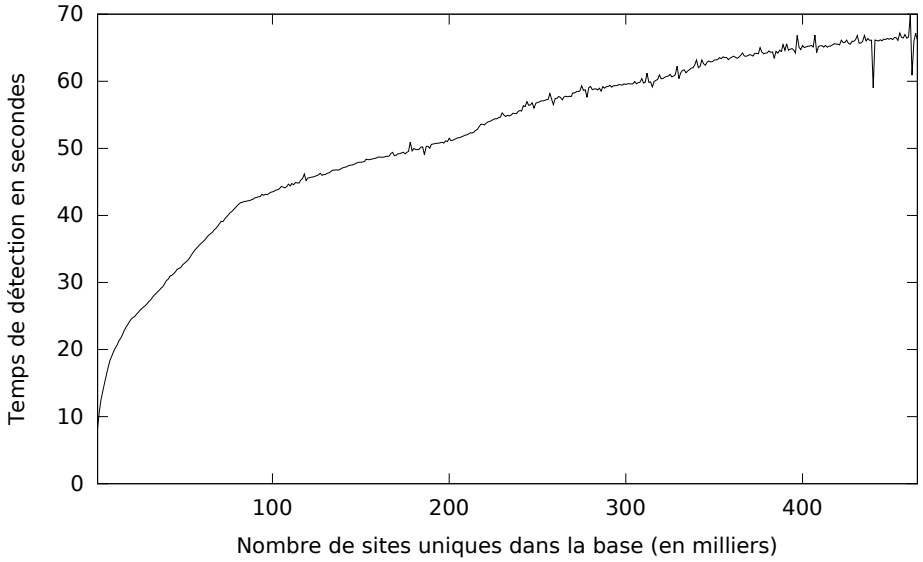


FIGURE 8.23 – Temps de détection, avec l'algorithme par parcours, de 5 programmes (29145 sites en tout) en fonction du nombre de sites dans la base

Comparaison des temps d'exécution. Le temps d'apprentissage des deux algorithmes varient linéairement en fonction du nombre de sites à ajouter à la base. En revanche le temps de détection pour l'algorithme par parcours dépend du nombre de sites dans la base tandis que le temps de détection pour SIDT n'en dépend pas.

Notre analyse est la suivante. Dans un contexte de détection pour implémenter un détecteur de programmes malveillants les deux algorithmes peuvent être utilisés pour apprendre l'ensemble des programmes du corpus en un temps raisonnable à condition d'optimiser les algorithmes pour améliorer le facteur linéaire. Il est par contre important que le temps de détection ne dépende pas trop du nombre de programmes dans le corpus, ce qui n'est pas le cas pour l'algorithme par parcours. Une approche incomplète comme SIDT remplit cette condition sur le temps de détection.

8.5 Conclusion

Nous avons formalisé le problème de la détection d'isomorphismes de graphes et de sous-graphes adapté à l'analyse morphologique et proposé deux approches algorithmiques tirant partie des spécificités des graphes de flot et des sites. La première, cherchant à reproduire des parcours de sites au sein d'un graphe de flot, donne des résultats exacts tandis que la seconde, fonctionnant par simple comparaison des sites, n'est pas exacte mais est beaucoup plus rapide : le temps de détection avec l'algorithme SIDT ne dépend pas du nombre de programmes dans le corpus.

Chapitre 9

Application à la détection de similarités logicielles

L'analyse morphologique appliquées aux graphes de flot de contrôle de plusieurs binaires peut également être utilisée pour détecter l'utilisation de fonctionnalités similaires entre ces programmes et faire correspondre précisément des morceaux de code assembleur de l'un et de l'autre.

Dans ce chapitre nous présentons des travaux réalisés en ce sens sur un logiciel malveillant, Waledac, qui utilise une bibliothèque connue de chiffrement, OpenSSL. Ces travaux ont fait l'objet d'une communication orale à REcon en 2012 [Bon+12a] et d'une publication à Malware [Bon+12b].

9.1 Contexte d'analyse de code

L'analyse manuelle de codes binaires inconnus est à la base de tout travail sur la détection de logiciels malveillants. L'analyste cherche d'une part à déterminer et comprendre l'ensemble du code du programme afin d'en connaître les fonctionnalités et le classer comme logiciel malveillant ou légitime. D'autre part, s'il s'agit d'un logiciel malveillant, il cherche à établir une signature du binaire permettant de détecter sa présence sur une machine infectée et éventuellement de le supprimer. L'analyse se fait à l'aide de différents outils, dont un désassembleur interactif tel IDA [Hexb] ou Radare [rad] pour l'analyse statique et d'un outil pour l'analyse dynamique comme un débogueur, un émulateur ou un logiciel d'instrumentation.

Les logiciels analysés ne présentent en général pas d'informations de compilation permettant d'identifier les bibliothèques logicielles utilisées ni leur version. Nous cherchons à automatiser la recherche de bibliothèques connues dans un logiciel à analyser et à marquer son emplacement dans un désassembleur interactif (IDA) afin d'éviter l'analyse inutile de code documenté et d'accélérer l'analyse manuelle.

9.1.1 Waledac et OpenSSL

Waledac [Cal+10], apparu en 2008, est un botnet, c'est à dire un programme malveillant transformant les machines infectées en esclaves (ou *zombies*) recevant des ordres d'un serveur de commande et de contrôle (C&C). Les machines esclaves communiquent entre elles sous la forme d'un réseau pair à pair et la charge finale principale du réseau est l'envoi de courrier électronique non sollicité (*spam*). Il s'agit d'un programme dont les fonctionnalités étaient déjà bien connues et documentées lorsque nous avons commencé à nous y intéresser. Notre objectif était alors de voir s'il était possible d'automatiser une partie de l'analyse, sachant que nous serions capables de vérifier nos résultats à l'aide d'analyses manuelles existantes.

Afin de savoir quelles méthodes de chiffrement le programme utilise, nous avons cherché des chaînes de caractères dans le binaire correspondant à quelques bibliothèques standard à l'aide du l'outil *strings*. Par chance il a été facile de trouver qu'il utilise la version 0.9.8e d'OpenSSL :

```
$ strings "Waledac v48 unpacked.exe" | grep OpenSSL
   EC part of OpenSSL 0.9.8e 23 Feb 2007
   ECDSA part of OpenSSL 0.9.8e 23 Feb 2007
```

OpenSSL [Ope] est une bibliothèque libre et documentée, elle ne devrait pas nécessiter une analyse de son code binaire ici inclus dans Waledac. Nous voulons aller plus loin afin de connaître précisément les fonctions utilisées ainsi que les parties de code communes.

9.2 Analyse morphologique

La technique d'analyse morphologique détaillée aux deux chapitres précédents a été directement utilisée : on détermine les graphes de flot de contrôle de chacun des deux binaires, on leur applique des réductions puis on les découpe en sites. On cherche ensuite des sites communs entre les Waledac et OpenSSL.

Taille des graphes de flot. Les binaires sur lesquels nous avons travaillé ont des graphes de flot de contrôle initiaux allant jusqu'à quelques centaines de milliers de sommets avant réduction et environ quinze mille sommets après réduction. Nous avons initialement trouvé 53 sites communs entre la version 0.9.8x d'OpenSSL et Waledac. Nous avons initialement pris la version 0.9.8x, la version à jour d'OpenSSL, parce que les versions antérieures n'étaient pas disponibles sous forme binaire pour Windows et nécessitaient d'être compilées.

Binaire	Taille du GFC	Taille du GFC réduit	Nombre de sites de taille 24
Waledac	38236	14626	11141
OpenSSL 0.9.8x	174754	28313	22171

Influence des options de compilation. Nous avons ensuite compilé la version 0.9.8e en utilisant différentes options de compilation utilisées avec le compilateur pour Windows Visual Studio. Nous avons pu remarquer que l'option donnant le plus sites en commun était celle compilée pour optimiser la taille du binaire. Le tableau suivant donne le nombre de sous-sites communs entre Waledac et chacune des versions d'OpenSSL.

Version	Remarque	Sites communs
0.9.8x	Version de mai 2012	53
0.9.8e	Optimise les performances temporelles (/0x /02)	53
0.9.8e	Optimise la taille du fichier (/01)	1264

9.2.1 Correspondance fine entre instructions

Dans un second temps nous avons voulu être capables de marquer, dans le désassembleur interactif utilisé, les instructions ainsi que les fonctions assembleur qui ont été reconnues. Nous avons utilisé les techniques précédentes pour qu'en plus de retourner une correspondance entre un site du graphe de motif P et un site du graphe de test T, nous ayons aussi l'information précise donnant la correspondance entre un sommet du sous-site de motif et un sommet du sous-site de test. Tous les algorithmes présentés au chapitre précédent peuvent fournir cette information.

Nous supposons donc que l'on dispose de la fonction *match* qui, à partir d'un sous-site S_P de P et d'un sous-site S_T de T, renvoie une liste de couples de correspondance entre un sommet de S_P et un sommet de S_T. Dans le cas où les deux sous-sites ne correspondent pas, elle renvoie $\emptyset$.

Comme décrit dans l'algorithme 9.1, nous considérons que plus la taille des sous-sites correspondants est grande, plus la correspondance entre les sommets sera précise. Nous cherchons en premier lieu le plus grand sous-site que l'on retrouve dans P et T et nous associons tous les sommets de P et de T correspondants dans ce sous-site. Nous récupérons l'ensemble des sous-sites d'une certaine taille de P et T à l'aide de l'algorithme 8.1 du chapitre précédent. Puis nous continuons avec un sous-site commun plus petit, en associant les sommets qui n'ont pas encore été associés, et ainsi de suite jusqu'à atteindre la taille initiale des sites choisis pour la détection, soit W.

9.2.2 Implémentation et résultats

Correspondance entre sommets. La fonction *match* donnant les correspondances a été implémentée pour une variante de l'algorithme d'Ullmann présentée au chapitre précédent, variante adaptée à la comparaison de sites de même taille. Elle a également été implémentée dans la version d'origine du détecteur fonctionnant par automates d'arbres. Nous avons ajouté dans le détecteur une possibilité d'export de ces correspondances.

Greffon IDA. Un greffon (ou *plugin*) a été développé pour le désassembleur interactif IDA, permettant, lorsque que deux instances du désassembleur sont lancées, chacune

Algorithme 9.1 : Association des sommets de deux graphes de flot de contrôle

Entrées : Deux graphes de flot, P et T de taille respective n_P et n_T, la taille minimale des sites recherchés, W

Résultat : Une liste de correspondances sommet à sommet entre P et T

```
association(P, n_P, T, n_T, W)
    L ← ∅
    A_P, A_T ← (∅, ∅)          /* ensembles des sites de T et P associés */
    m ← min(n_P, n_T)
    /* On extrait tous les sites :                                        */
    pour W ≤ i ≤ m faire
        S_{P,i} ← sites(P, i)
        S_{T,i} ← sites(T, i)
    fin
    k ← m
    tant que k ≥ W faire
        pour S_P ∈ S_{P,k} faire
            pour S_T ∈ S_{T,k} faire
                /* On place dans C les couples de sommets
                   correspondants :                                         */
                C ← match(S_P, S_T) pour (s_P, s_T) ∈ C faire
                    si s_P ∉ A_P et s_T ∉ A_T alors
                        L ← L ∪ {(s_P, s_T)}
                        A_P ← A_P ∪ {s_P}
                        A_T ← A_T ∪ {s_T}
                    fin
                fin
            fin
        fin
        k ← k − 1
    fin
    retourner L
```

analysant un des deux programmes comparés, d'aligner le code comparé selon les correspondances trouvées au préalable par analyse morphologique. Le greffon surligne les instructions ayant trouvé un correspondant dans l'autre graphe de flot et donne également la correspondance entre les fonctions assembleur du premier programme et du second.

Exemple avec Waledac et OpenSSL. La figure 9.1 montre un extrait de la page de correspondances entre les fonctions assembleur identifiées par IDA entre la DLL principale d'OpenSSL (*libeay32-098e.dll*) et Waledac (*Waledac48.int*). La première colonne indique les adresses des instructions d'OpenSSL correspondant aux instructions présentes aux adresses de Waledac situées à la troisième colonne. Les deuxième et quatrième colonnes donnent le nom des fonctions assembleur auxquelles ces instructions appartiennent. OpenSSL ayant été compilé avec des options de débogage, le nom des fonctions est re-

trouvé par IDA, ce qui n'est pas le cas pour Waledac. Il est à noter que, puisqu'on a effectué des réductions avant de lancer la comparaison, la plupart des instructions, dont les instructions séquentielles, ne sont pas comparées : ce sont principalement les instructions modifiant le flot de contrôle qui ont été considérées.

OpenSSL (libeay32-098e.dll)		Waledac (Waledac48.int)	
Adresse	Fonction	Adresse	Fonction
10002CDE	CRYPTO_new_ex_data	00455B5C	sub_455B55
10002CD0	CRYPTO_new_ex_data	00455B5E	sub_455B55
10002D0A	CRYPTO_new_ex_data	00455B72	sub_455B55
10002D0C	CRYPTO_new_ex_data	00455B74	sub_455B55
10021F6D	AES_set_encrypt_key	00452C1E	sub_452C1B
10021F7C	AES_set_encrypt_key	00452C2D	sub_452C1B
10021F88	AES_set_encrypt_key	00452C39	sub_452C1B
10021F99	AES_set_encrypt_key	00452C4A	sub_452C1B
10021FA0	AES_set_encrypt_key	00452C51	sub_452C1B
10021FA7	AES_set_encrypt_key	00452C58	sub_452C1B
10021FB3	AES_set_encrypt_key	00452C64	sub_452C1B
10021FD9	AES_set_encrypt_key	00452C8A	sub_452C1B
10021FEF	AES_set_encrypt_key	00452CA0	sub_452C1B
100224D9	AES_set_encrypt_key	0045317D	sub_452C1B
100224E0	AES_set_decrypt_key	00453184	sub_45317E
100224F0	AES_set_decrypt_key	00453194	sub_45317E
100224FA	AES_set_decrypt_key	0045319E	sub_45317E

FIGURE 9.1 – Fonctions correspondantes entre OpenSSL et Waledac

Nous avons ensuite vérifié dans IDA, grâce à la fonction d'alignement de code du greffon, que le code des instructions correspondantes était bien suffisamment similaire pour valider la méthode. Une telle correspondance est donnée en exemple à la figure 9.2 pour la fonction *AES_set_encrypt_key*. L'image est constituée d'un bout du graphe de flot de contrôle d'OpenSSL à gauche et d'un bout du GFC de Waledac à droite. Les instructions surlignées en orange et alignées sont celles pour lesquelles on a trouvé une correspondance.

Sur cet exemple nous n'avons pas trouvé de correspondance sommet à sommet qui ne soit pas cohérente mais certaines fonctions n'étaient pas couvertes par suffisamment de sommets pour que l'on soit assuré d'une correspondance. La figure 9.3 donne un extrait des fonctions d'OpenSSL retrouvées dans Waledac et les fonctionnalités qu'elles fournissent : on y trouve par exemple des méthodes préparant au chiffrement AES, RSA et DSA.

Il n'est pas surprenant que l'on retrouve ces fonctionnalités puisque Waledac utilise du chiffrement AES et RSA et gère des certificats X509. Notre contribution est l'emploi de la technique d'analyse morphologique pour déterminer automatiquement ces informations sans analyser manuellement le code assembleur.

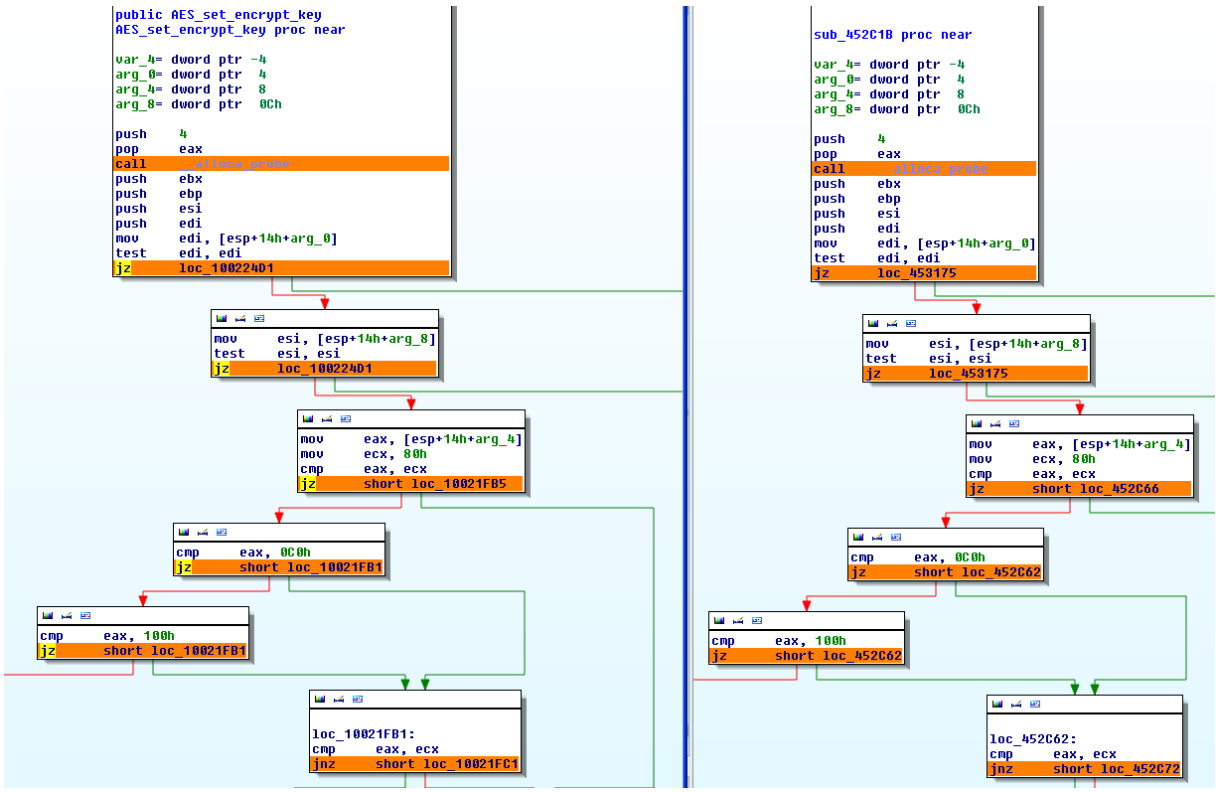


FIGURE 9.2 – Capture d'écran d'IDA : Code correspondant entre OpenSSL à gauche et Waledac à droite

Fonctions	Fonctionnalité
AES_set_encrypt_key, AES_set_decrypt_key	Chiffrement AES
RSA_free, DSA_size, DSA_new_method	Chiffrement RSA / DSA
X509_PUBKEY_set, X509_PUBKEY_get	Certificats X509
BN_is_prime_fasttest_ex, BN_ctx_new, BN_mod_inverse	Gestion des grands entiers (BN)
CRYPTO_lock, CRYPTO_malloc	Fonctions génériques d'OpenSSL

FIGURE 9.3 – Fonctions d'OpenSSL retrouvées dans Waledac et fonctionnalités associées

9.3 Limites

On a vu qu'on retrouve certaines fonctions utilisées lors d'un chiffrement AES. Pourtant nous n'avons pas retrouvé les fonctions principales servant à chiffrer et déchiffrer à l'aide de l'algorithme AES. La raison est que notre approche par graphes de flot réduits s'applique très mal aux fonctions de chiffrement qui sont en général très simples du point de vue de leur graphe de flot de contrôle. Une vue simplifiée du GFC de la fonction *AES_encrypt* est donnée en figure 9.4.

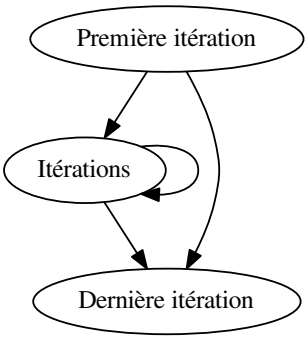


FIGURE 9.4 – GFC simplifié de la fonction AES_encrypt

Le GFC réduit est trop petit pour pouvoir être détecté par l'analyse morphologique qui prend des graphes d'une taille 24 au minimum. En fait beaucoup d'algorithmes de chiffrement ont cette structure une fois réduits et elle n'est pas non plus spécifique aux algorithmes de chiffrement. De ce fait l'analyse morphologique est inadaptée à la détection de ce genre de fonctions.

9.4 Application à Duqu et Stuxnet

Dans cette section nous présentons nos travaux sur deux programmes malveillants particuliers, Duqu et Stuxnet. Lorsque nous avons commencé ces travaux Stuxnet était déjà documenté et détecté et Duqu venait d'être découvert. Il a rapidement été dit qu'ils étaient semblables et nous avons donc cherché à détecter Duqu connaissant Stuxnet.

Stuxnet, découvert en juin 2010, est capable de cibler et de reprogrammer des systèmes industriels. Sans que cela ait été prouvé, il a été avancé qu'il visait le programme nucléaire iranien. Symantec [Sym11b] indique que la plupart des systèmes infectés sont en Iran et que la cible pourrait être un système de contrôle de centrifugeuses.

Duqu, découvert en premier par Crysys [Cry11] en septembre 2011, laboratoire de sécurité et de cryptographie de l'université de Budapest, a directement été détecté comme apparenté à Stuxnet parce que ces deux programmes utilisent des techniques d'infection et de propagation similaires. Duqu est un outil offensif utilisé pour le vol d'informations. Symantec [Sym11a] a identifié parmi ses fonctionnalités des enregistreurs de frappes (*keyloggers*), de l'écran, de l'activité réseau ainsi que des outils de détection de machines sur le

réseau. Il est maintenu à jour via des serveurs de commande et de contrôle (C&C) et dispose d'une fonctionnalité d'auto-destruction après, typiquement, 36 jours sans nouvelles du C&C.

Les attaques semblent réussies puisque le programme malveillant n'a pas été détecté à chaud alors que certaines opérations ont duré plusieurs mois, mais seulement post-mortem. De nombreuses souches de Duqu ont été trouvées dans la natures, chacune avec des binaires différents mais similaires. Kaspersky a publié un historique des versions [Kas12], la dernière souche détectée date de février 2012, bien après que les première attaques n'ont été détectées et documentées.

Analyse des similarités. Nous avons effectué une analyse similaire à celle décrite au chapitre précédent pour trouver les similarités entre Stuxnet et Duqu afin de trouver les parties de Stuxnet que l'on retrouve dans Duqu. Pour ces deux programmes malveillants, nous avons analysé la DLL (bibliothèque logicielle au format Windows, ou *Dynamic Link Library*) principale une fois que celle-ci a été déchiffrée. Nous avons extrait les graphes de flot réduits de Duqu et Stuxnet. Nous avons trouvé 846 sites communs entre Duqu et Stuxnet : 26.5% des sites de Duqu proviennent de Stuxnet. Lorsque l'on regarde les sommets correspondants dans les graphes réduits, on s'aperçoit que ces sites contiennent 2215 sommets dans les graphes de flot réduits : 60.3% des sommets du graphe de flot réduit de Duqu correspondent à des sommets présents dans Stuxnet.

Forts de ces résultats indiquant que Duqu et Stuxnet partagent beaucoup de code, nous considérons donc qu'un détecteur de programmes malveillants fonctionnant avec la technique d'analyse morphologique connaissant Stuxnet serait capable de détecter Duqu.

La principale difficulté que doit résoudre un détecteur est que le programme provocant l'infection de Duqu ne ressemble pas à Stuxnet ni à aucun autre programme malveillant connu. Ce n'est que lorsque certains composants de Duqu sont déchiffrés et installés que l'on peut le détecter. Nous détaillons dans le chapitre suivant la méthode d'infection de Duqu et le travail nécessaire afin de détecter une attaque.

9.5 Perspectives

Un approfondissement de cette approche à l'analyse de similarités logicielles serait intéressant. D'une part nous voudrions effectuer des expériences sur plus d'échantillons de code malveillant pour confirmer les résultats prometteurs mis en lumière dans ce chapitre. D'autre part nous souhaiterions comparer en détail notre approche à celles existantes, notamment par rapport à une comparaison directe des instructions des binaires.

9.6 Conclusion

Nous avons identifié des similarités entre Waledac et une version spécifique d'OpenSSL, nous avons pu reconstruire automatiquement des correspondances fines afin de déterminer quelles fonctions d'OpenSSL ont été utilisées par Waledac. L'obstacle principal à une généralisation de cette méthode est qu'elle est sensible aux options de compilation et

nécessite de déterminer au préalable les binaires à comparer. Cette technique n'est pas infaillible puisqu'elle n'est pas adaptée pour détecter certaines structures ou fonctions, telles des fonctions de chiffrement.

Nous avons ensuite développé un greffon pour IDA permettant d'analyser les similarités trouvées afin de permettre à l'analyste de ne pas s'attarder sur du code déjà documenté. Cette approche, appliquée à Duqu et Stuxnet nous a permis de mettre en lumière de nombreuses portions de code partagées entre ces deux logiciels malveillants.

Chapitre 10

Cas d'étude : détection de Duqu connaissant Stuxnet

Dans ce chapitre nous continuons nos travaux sur deux programmes malveillants particuliers, Duqu et Stuxnet, dont nous avons montré qu'ils partageaient du code au chapitre précédent. Notre objectif est de pouvoir détecter une attaque par Duqu connaissant Stuxnet.

Nous cherchons donc à détecter Duqu avant qu'il ne puisse infecter une machine ciblée. Nous étudions pour cela un composant spécifique de Duqu, son pilote (*driver*) qui permet de charger discrètement le code malveillant en mémoire. Notre contribution consiste en la rétroingénierie de ce composant, en la reconstitution de son code source et en une analyse de son fonctionnement. Nous avons détournons ensuite le pilote pour en faire une version défensive capable de détecter d'éventuelles attaques similaires. En particulier nous montrons comment le pilote de Duqu modifié permet de détecter et d'empêcher une attaque par Duqu. Ces travaux ont fait l'objet d'une publication à SSTIC [Bon+13b] ainsi qu'à Malware [Bon+13a].

10.1 Détection d'une infection par Duqu

10.1.1 Déroulement d'une infection

L'infection détectée par Crysys utilise un document Microsoft Word piégé, incluant Duqu. Dans un premier temps il exploite une faille jusque là inconnue (*0-day* sur les polices d'écriture TrueType [CVE11]) du noyau Windows afin d'installer trois composants sur le système :

— Un pilote : `nfrd965.sys`
— Une DLL chiffrée : `NETP191.PNF`
— Un fichier de configuration chiffré : `netp192.PNF`

Au redémarrage de la machine, le pilote surveille le chargement des processus par le système d'exploitation et injecte la DLL principale de Duqu, une fois déchiffrée, dans un processus spécifié par le fichier de configuration, typiquement `services.exe`. Enfin

le pilote modifie `services.exe` afin qu'il exécute la charge finale, qui est incluse dans la DLL.

Une fois installé sur une première cible Duqu reçoit des ordres d'attaque et de propagation d'un serveur C&C. Chaque machine infectée peut être configurée pour se connecter à l'attaquant, pas directement mais par la machine qui l'a infectée, créant une sorte de tunnel de routage pour des machines non accessibles directement depuis l'extérieur. Une illustration de ce mécanisme est donnée en figure 10.1.

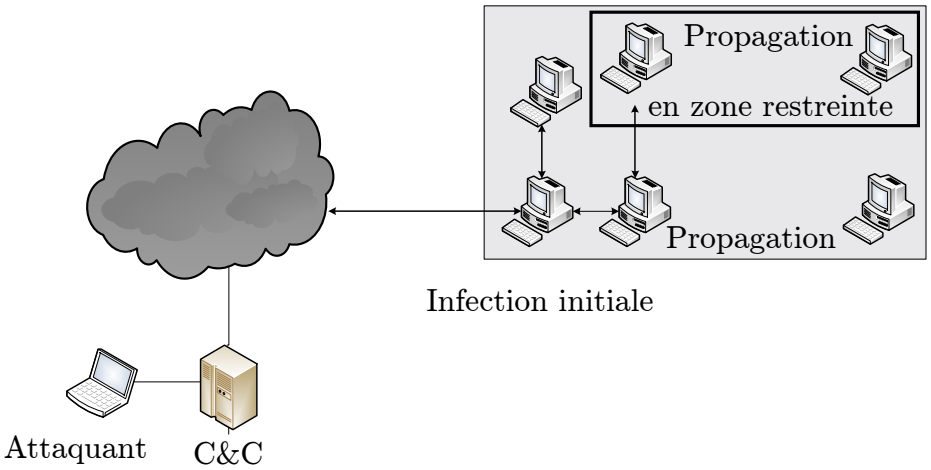


FIGURE 10.1 – Schéma de propagation en profondeur de Duqu

Difficulté de la détection. L'obstacle principal à la détection est que seul le pilote est présent déchiffré sur le disque. La DLL est chiffrée et empaquetée avec UPX, elle n'apparaît déchiffrée qu'en mémoire, lorsqu'elle est injectée dans `services.exe`.

Angle d'attaque pour une détection. L'attaque peut être détectée au moment du déchiffrement de la DLL principale et de son injection. Elle doit prendre place après le déchiffrement mais avant que la charge finale ne soit exécutée. Nous aurons pour cela besoin de suivre les processus lancés et de pouvoir analyser les DLL qu'ils exécutent. Étant donné que le pilote de Duqu réalise cette opération, nous avons choisi de le modifier de telle sorte qu'il puisse s'interfacer avec le détecteur par analyse morphologique. Pour mettre ce plan en œuvre, nous avons

— reconstitué, par rétroingénierie, le code source du pilote de Duqu à partir de son binaire.
— modifié son code pour qu'il surveille le chargement des processus sans provoquer d'injection.

10.1.2 Reconstruction du code du pilote

Nous savions donc que les DLL principales de Duqu et Stuxnet partageaient du code. Le pilote de Stuxnet a été décompilé par Amr Thabet [Tha11]. Nous avons voulu suivre

la même route et désassembler le pilote de Duqu afin de le documenter. Nous avons eu à notre disposition la souche du pilote découverte en octobre 2011 en Europe : `nfrd965.sys`.

Nous avons donc travaillé sur la rétroingénierie de cette version spécifique du pilote afin d'en documenter les fonctionnalités. Notre objectif est d'obtenir un code compréhensible que l'on peut compiler et dont la version compilée soit au plus proche du binaire d'origine.

Décompilation avec IDA

Nous avons utilisé le module de décompilation "Hex-Rays Decompiler", intégré à IDA sous la forme d'un greffon [Hexa]. Il permet de générer un pseudo-code C à partir du fichier binaire en cours d'analyse. Il produit non seulement du code source mais facilite également sa réécriture directement à l'intérieur de l'interface graphique du greffon. Malheureusement le code en sortie n'est, dans notre cas, pas exploitable directement. D'une part le code n'est pas compilable parce que des types de variables n'ont pas été correctement reconnus et certaines conventions d'appel ne sont pas standard (non reconnues par le décompilateur). De plus le code généré est difficilement lisible en partie parce que certaines structures n'ont pas été identifiées. Nous détaillons dans les paragraphes suivants ces difficultés et des moyens de résolution.

Nous avons procédé de manière incrémentale afin de reconstruire le code petit à petit en vérifiant à chaque étape que le code compile et qu'une fois compilé il est équivalent à celui du binaire `nfrd965.sys` original. Cela a consisté à :

- commenter tout le pseudo-code sauf la fonction du point d'entrée du pilote (*DriverEntry*) et les variables globales s'y rapportant,
- régler chaque erreur une par une,
- comparer le code compilé au binaire original, modifier le code pour s'en rapprocher,
- ajouter du code auparavant commenté et revenir à l'étape de correction d'erreurs.

Identification des structures et des types

La figure 10.2 montre les premières lignes du code C reconstruit par le décompilateur pour une des fonctions du pilote. Beaucoup d'informations manquent. Par exemple la plupart des types sont décrits comme des entiers (ou des pointeurs) : il n'est pas possible de savoir quel type de données cette fonction manipule. Nous allons détailler sur cet exemple quelques techniques permettant de récupérer ces informations.

Certaines constantes peuvent nous aider : par exemple `0xF750F284 XOR 0xF750B7D4 = 0x00004550` , qui représente la chaîne de caractères 'PE\0\0' en ASCII. Le texte est obscurci à l'aide d'une opération de ou exclusif (*XOR*).

Nous soupçonnons alors que cette fonction est utilisée pour le traitement de fichiers binaires au format PE. La documentation officielle de Microsoft Visual C++ détaille la structure PIMAGE_NT_HEADERS dont le premier champ, *Signature*, vaut PE\0\0 pour les binaires Windows. Ainsi la variable `v4`, qui est comparée à PE\0\0, est probablement du type PIMAGE_NT_HEADERS. Nous forçons ce type pour cette variable au sein d'IDA à la place du type `int` et IDA trouve automatiquement le nom des champs de ce type de variable à partir de leur décalage (*offset*) en mémoire. Lorsque les structures

```
signed int __cdecl sub_12F36(int a1, int a2, int a3)
{
  int v4; // eax@3
  unsigned __int16 v5; // cx@4
  int v6; // ecx@7

  v4 = a2 + *(_DWORD *)(a2 + 60);
  if (*(_DWORD *)v4 ^ 0xF750F284 != 0xF750B7D4)
    return 1;
```

FIGURE 10.2 – Premières lignes de la fonction ParsePE décompilée par IDA

sont spécifiques au binaire analysé, il est possible de les définir manuellement dans le décompilateur. Nous avons retrouvé les types des autres variables de manière similaire.

La figure 10.3 donne les premières lignes de la fonction retouchée. Elle est lisible par un développeur C : on peut voir que la fonction vérifie si un fichier est binaire PE. Le reste de la fonction parcourt le binaire PE passé en entrée et remplit une structure spécifique avec quelques informations (son point d'entrée, ses sections, etc.). De plus le code compilé de cette fonction est très similaire au binaire d'origine.

```
NTSTATUS __cdecl ParsePE(__out PEDataPtr pPEData,
    __in PIMAGE_DOS_HEADER BaseAddress, __in int flag){
PVOID infosPE;
PIMAGE_DOS_HEADER pDosHeader;
PIMAGE_NT_HEADERS pNtHeader;

pNtHeader = (DWORD)infosPE + infosPE->e_lfanew;
if ((pNtHeader->Signature ^ 0xF750F284)
      != (IMAGE_NT_SIGNATURE ^ 0xF750F284))
    return STATUS_WAIT_1;
```

FIGURE 10.3 – Premières lignes de la fonction ParsePE reconstruite

Conventions d'appel

Pour chaque routine IDA cherche à déterminer la convention d'appel utilisée à partir des registres qui sont lus avant d'être écrits (paramètres) et ceux écrits sans être lus après (valeur de retour). Si ces registres correspondent à un appel classique, IDA l'annote dans le code C pour que le compilateur respecte la convention. Les conventions d'appel de Microsoft Visual C++ sont données Figure 10.4, l'appel par défaut étant *thiscall*. Dans le cas où il ne détermine pas la convention, il annote les registres d'entrée et de sortie en notant qu'il s'agit d'un appel non conventionnel (*usercall*) et met la définition de la fonction en commentaire (ici les arguments sont passés dans les registres `edi` et `esi`) :

```
// int __usercall SearchForCode<eax>(int *a1<edi>, int a2<esi>);
```

Une convention d'appel non standard est détectée dans le cas où une partie de la fonction a été écrite directement en assembleur ou à la suite d'une optimisation faite par le compilateur. On doit alors réécrire, en partie en assembleur, la fonction sans passer par le décompilateur ou choisir une convention d'appel soi-même.

Convention	Arguments	*this* (C++)	Retour	Nettoie la pile
C (__cdelcl)	pile	(argument)	eax	appelant
Standard (__stdcall)	pile	(argument)	eax	appelé
Thiscall (__thiscall)	pile	ecx	eax	appelé
Fastcall (__fastcall)	ecx, edx, pile	(argument)	eax	appelé

FIGURE 10.4 – Conventions d'appel dans leur version Visual C++

Nous avons réalisé ce type d'analyse sur l'ensemble du pilote afin d'en reconstruire une version compréhensible et cohérente du code source du pilote.

10.2 Analyse fonctionnelle du pilote de Duqu à partir du code source

Une fois le code du pilote reconstitué, nous l'avons donc analysé. Il y a deux phases principales, la première consiste en la mise en place du pilote : il demande au système à être notifié en cas de chargement de binaires et initialise ses mécanismes de furtivité. La seconde phase est lancée lorsqu'une notification est signalée au chargement d'un des binaires ciblés : le pilote infecte alors le binaire en y injectant la DLL du Duqu puis celle-ci active la charge finale.

10.2.1 Initialisation du pilote lors du démarrage du système

Sous Windows l'ordre de démarrage des pilotes est déterminé par leur clé de registre `Group`. Le pilote `nfrd965.sys` de Duqu, appartenant au groupe "network", est activé avant même que la couche d'abstraction matérielle (*HAL*) ne soit chargée en mémoire.

Le pilote, une fois démarré, commence par allouer un emplacement mémoire de 512 octets destiné à contenir un tableau de pointeurs de fonctions partagées entre les différentes routines de rappel (*callback*) qui seront définies par la suite. Il passe ensuite au déchiffrement de quelques paramètres internes, révélant le nom et l'emplacement de la clé de registre utilisée pour la configuration de l'injection.

Si le déchiffrement s'est correctement déroulé, vient alors la vérification du mode d'exécution : soit le système s'avère être en mode sans échec ou en mode débogage, dans ce cas le pilote termine prématurément son exécution ; soit il est en mode normal, le pilote crée alors un *device*, `{624409B3-4CEF-41c0-8B81-7634279A41E5}`, et définit la liste des commandes de contrôle qu'il sera amené à traiter.

Cette étape réalisée, le pilote enregistre deux fonctions de rappel auprès du gestionnaire d'événements interne du noyau. La première est requise par le système : elle est utilisée

pour créer un point d'accès (`\Device\Gpd0`) et un lien (`\DosDevices\GpdDev`) vers le pilote ainsi que pour définir une pile mémoire pour le *device*. La seconde fonction sera appelée lorsque le pilote sera initialisé ou ré-initialisé.

Cette seconde fonction attend que le noyau Windows soit complètement chargé en vérifiant si la DLL `hal.dll` est chargée en mémoire. Lorsque le système est prêt, un point d'accès, `\Device\Gpd1`, est créé et lié à une routine de traitement des requêtes. À ce stade le pilote est prêt à réaliser l'injection.

Techniques de furtivité

Le pilote agit désormais furtivement (on parle de *rootkit*) et évite d'utiliser directement des appels systèmes connus pour être sensibles, utilisés par des logiciels malveillants, et probablement surveillés par un éventuel antivirus. La fonction *ZwAllocateVirtualMemory* peut être utilisée pour allouer de la mémoire au sein d'un processus au choix, pour y injecter du code arbitraire par exemple. De plus, afin de détourner le point d'entrée d'un binaire (*hook*), Duqu veut également utiliser la fonction *ZwProtectVirtualMemory* que Microsoft a délibérément omise de la liste des fonctions accessibles en dehors du noyau. Cette fonction permet de modifier les permissions d'une page mémoire et peut être utilisée pour rendre une partie de code accessible en écriture ou rendre une section de données exécutable.

Ces deux fonctions sont implémentées dans le noyau Windows, dans les fichiers `Ntoskrnl.exe` ou `ntkrnlpa.exe`, selon les versions. Le pilote inspecte chaque module, DLL et exécutables, chargés par le système lors du démarrage à la recherche d'un de ces deux fichiers.

Une fois le fichier cible trouvé, le pilote utilise la fonction *ParsePE* pour l'examiner et y retrouver l'adresse de *ZwProtectVirtualMemory*. Pour cela il dispose d'un motif à reconnaître. Il est à la recherche d'un appel vers *ZwAllocateVirtualMemory*, dont l'adresse est connue parce qu'elle est présente dans la table d'exports du noyau, suivi par l'instruction `push 0x104` et par une instruction `call`. Si ce motif, représenté en figure 10.5, est retrouvé alors l'adresse cible de ce `call` est considérée comme étant *ZwProtectVirtualMemory*. À partir de cet instant, le pilote connaît les adresses mémoires de ces deux fonctions.

Vérification d'intégrité. Le pilote cherche à détecter si les fonctions *ZwAllocateVirtualMemory* et *ZwProtectVirtualMemory* ont été la cible d'un détournement défensif par un antivirus cherchant à les surveiller. Il vérifie dans un premier temps que les deux fonctions sont présentes dans l'espace mémoire du noyau et non en espace utilisateur. Dans un second temps il leur applique un masque d'intégrité vérifiant la valeur d'une partie des 20 premières adresses mémoires sur lesquelles les deux fonctions sont codées. Le masque est le même pour les deux fonctions et est donné en figure 10.6. Si les fonctions passent le test, leurs adresses sont considérées valides et sont conservées pour une future utilisation discrète.

```
(01) PAGE:004ED1AD                    loc_4ED1AD: [...]
(02) PAGE:004ED1BC 50                 push    eax             ; BaseAddress
(03) PAGE:004ED1BD 57                 push    edi             ; ProcessHandle
(04) PAGE:004ED1BE E8 19 8C F1 FF     call DS:ZwAllocateVirtualMemory
(05) PAGE:004ED1C3 3B C3              cmp     eax, ebx
(06) PAGE:004ED1C5 8B 4D FC           mov     ecx, [ebp+BaseAddress]
(07) PAGE:004ED1C8 89 4E 0C           mov     [esi+0Ch], ecx
(08) PAGE:004ED1CB 7C 2E              jl      short loc_4ED1FB
(09) PAGE:004ED1CD 38 5D 0B           cmp     byte ptr [ebp+ProcessHandle+3], bl
(10) PAGE:004ED1D0 74 27              jz      short loc_4ED1F9
(11) PAGE:004ED1D2 8B 45 D0           mov     eax, [ebp+var_30]
(12) PAGE:004ED1D5 89 45 F8           mov     [ebp+ProtectSize], eax
(13) PAGE:004ED1D8 8D 45 F4           lea     eax, [ebp+OldProtect]
(14) PAGE:004ED1DB 50                 push    eax             ; OldProtect
(15) PAGE:004ED1DC 68 04 01 00 00     push 104h
(16) PAGE:004ED1E1 8D 45 F8           lea     eax, [ebp+ProtectSize]
(17) PAGE:004ED1E4 50                 push    eax             ; ProtectSize
(18) PAGE:004ED1E5 8D 45 FC           lea     eax, [ebp+BaseAddress]
(19) PAGE:004ED1E8 50                 push    eax             ; BaseAddress
(20) PAGE:004ED1E9 57                 push    edi             ; ProcessHandle
(21) PAGE:004ED1EA E8 93 96 F1 FF     call loc_406882 ; ZwProtectVirtualMemory
(22) PAGE:004ED1EF 3B C3              cmp     eax, ebx
```

FIGURE 10.5 – Fonction faisant appel à *ZwProtectVirtualMemory*

b8					8d	54	24	04	9c
6a	08	e8					c2	14	

ZwAllocateVirtualMemory :

0x405ddc	b8	11	00	00	00	mov eax, 0x11
0x405de1	8d	54	24	04		lea edx, [esp+ProcessHandle]
0x405de5	9c					pushf
0x405de6	6a	08				push 8
0x405de8	e8	b9	20	00	00	call +0x20be (sub_407ea6)
0x405ded	c2	14	00			ret 0x14

ZwProtectVirtualMemory :

0x406882	b8	89	00	00	00	mov eax, 0x89
0x406887	8d	54	24	04		lea edx, [esp+ProcessHandle]
0x40688b	9c					pushf
0x40688c	6a	08				push 8
0x40688e	e8	13	16	00	00	call +0x1618 (sub_407ea6)
0x406893	c2	14	00			ret 0x14

FIGURE 10.6 – Masque d'intégrité appliqué à *ZwAllocateVirtualMemory* et *ZwProtectVirtualMemory*. Les valeurs grisées sont celles qui sont vérifiées.

Initialisation de la mémoire partagée

Une mémoire partagée est allouée et utilisée comme lien entre les routines de rappel du pilote et le noyau. Elle contiendra, entre autres, les paramètres pour l'infection déchiffrés depuis les données d'une clé de registre et une table d'imports donnant accès à la DLL `kernel.dll` et aux fonctions du noyau. Cette table d'imports sera utilisée à la fois par le code que Duqu va injecter dans `services.exe` et par la charge finale.

La phase d'initialisation prend fin en mettant en place une notification système dès qu'un module (DLL ou exécutable) est chargé en mémoire, via l'appel système **PsSetLoadImageNotifyRoutine**.

10.2.2 Injection de code

Traitement de la première notification

Préparation à l'injection. Le pilote est notifié à chaque fois qu'un module (DLL ou exécutable) est chargé en mémoire. À chaque fois le pilote tente de localiser l'emplacement mémoire du module. Pour cela il utilise l'identifiant du processus que lui fournit le système d'exploitation lors de la notification. Il lit l'adresse de base du fichier directement à partir des informations accessibles dans la structure PEB (*Process Environment Block*) et la compare à celle passée en paramètre par le système. Il vérifie que le fichier de configuration est bien déchiffré dans la mémoire partagée et lit le champ donnant la cible de l'injection. Comme expliqué dans le document de Crysys [Cry11], la cible est `services.exe` donc nous nous focalisons sur ce processus et l'injection dont il sera victime.

Injection de la charge finale. Le pilote de Duqu va maintenant injecter du code malveillant dans `services.exe` de telle manière que la charge finale soit exécutée par `services.exe` avant que son code légitime ne soit à son tour exécuté.

Une fois que `services.exe` est chargé, le pilote détermine son point d'entrée et alloue de la mémoire dans sa section `.data` à l'aide de la fonction *ZwAllocateVirtualMemory*. Deux fichiers PE dont les entêtes ont été altérés à des fins de furtivité sont injectés. Ensuite certaines constantes ('`MZ`', '`IMAGE_NT_SIGNATURE`', '`IMAGE_PE_i386_MACHINE`, et '`IMAGE_PE32_MAGIC`') du premier code injecté sont restaurées. Certaines adresses sont recalculées : les adresses cibles de sauts qui étaient codées en dur doivent être recalculées. Enfin le pilote modifie les permissions du point d'entrée de `services.exe` de `RX` (`PAGE_EXECUTE_READ`) à `RWX` (`PAGE_EXECUTE_WRITECOPY`) en utilisant la fonction *ZwProtectVirtualMemory*.

Le pilote `nfrd965.sys` alloue alors de la mémoire dans le processus `services.exe` de la taille de la DLL déchiffrée `NETP191.PNF` augmentée de 57 octets. Ensuite un gestionnaire d'événements (*handler*) est ouvert sur le pilote et est sauvegardé dans la mémoire partagée afin de pouvoir être utilisé par le code injecté.

Traitement de la seconde notification

Le pilote n'est pas uniquement notifié quand le module principal (`services.exe`) est chargé mais également lorsque des DLL liées à ce module sont également chargées. En

particulier lorsque la DLL `kernel32.dll` est chargée, le pilote cherche les adresses de 10 de ses fonctions exportées qui seront utilisées par la charge finale. Toujours dans une optique de furtivité la recherche consiste à comparer un haché cryptographique au nom de chacune des fonctions exportées par la DLL. Cette étape se termine par une sauvegarde des 12 premiers octets présents au point d'entrée de `services.exe` et leur remplacement par un saut vers le premier code injecté et restauré. Les premières instructions du point d'entrée sont changées en l'instruction `mov eax, @AdresseInjection` suivie de `call eax`.

Le processus `services.exe` a ainsi été altéré et est prêt à lancer la charge finale.

Lancement de la charge finale

Le système d'exploitation termine l'initialisation de `services.exe` et procède à son exécution en passant le contrôle au point d'entrée modifié, c'est à dire au premier code injecté par Duqu.

Sa première tâche consiste à déterminer sa propre adresse en mémoire afin de pouvoir recalculer les adresses de certaines cibles de saut. Cette opération peut être effectuée à l'aide de deux instructions : un `call +5` vers l'instruction suivante suivi d'un `pop eax` a pour effet de placer l'adresse de retour (celle de `pop eax`) en haut de la pile puis de la dépiler dans `eax` qui contient alors cette même adresse. Il modifie alors les adresses à partir du nouveau point d'entrée déterminé.

Il restaure ensuite les entêtes du second PE injecté afin de le rendre valide et remplit, dans une structure partagée, une table d'import à partir des 10 fonctions trouvées précédemment de `kernel32.dll`. Il crée ensuite un gestionnaire d'événements sur la DLL `ntdll.dll` qui est enregistré dans une structure partagée. Il transfert ensuite le contrôle au point d'entrée sur second code injecté.

Ce module additionnel ajoute les données de son propre entête (adresse du module, nombre de sections, adresse de la table d'exports) dans la mémoire partagée. Enfin ces informations sont utilisées pour charger ce PE manuellement en mémoire : les espaces mémoires sont alloués, l'entête est copié, les sections et les DLL liées sont chargées en mémoire, une table d'imports est créée, les adresses sont recalculées à partir de son point d'entrée. Puis la DLL principale de Duqu, `NETP191.PNF`, est chargée et liée à ce PE et son point d'entrée est appelé. La figure 10.7 donne l'état du processus `services.exe` et de la mémoire à cette étape de l'injection.

La charge finale contenue dans la DLL est maintenant en place et exécutée. Une fois qu'elle a fini, elle envoie une requête au pilote via le point d'accès créé précédemment, `{624409B3-4CEF-41c0-8B81-7634279A41E5}`, afin qu'il restaure les 12 premiers octets du point d'entrée de `services.exe`. Une seconde requête est envoyée pour restaurer les droits d'accès d'origine du point d'entrée de `services.exe`.

L'attaque ayant été réalisée, le contrôle est maintenant passé à `services.exe`, qui a été restauré et s'exécute cette fois normalement.

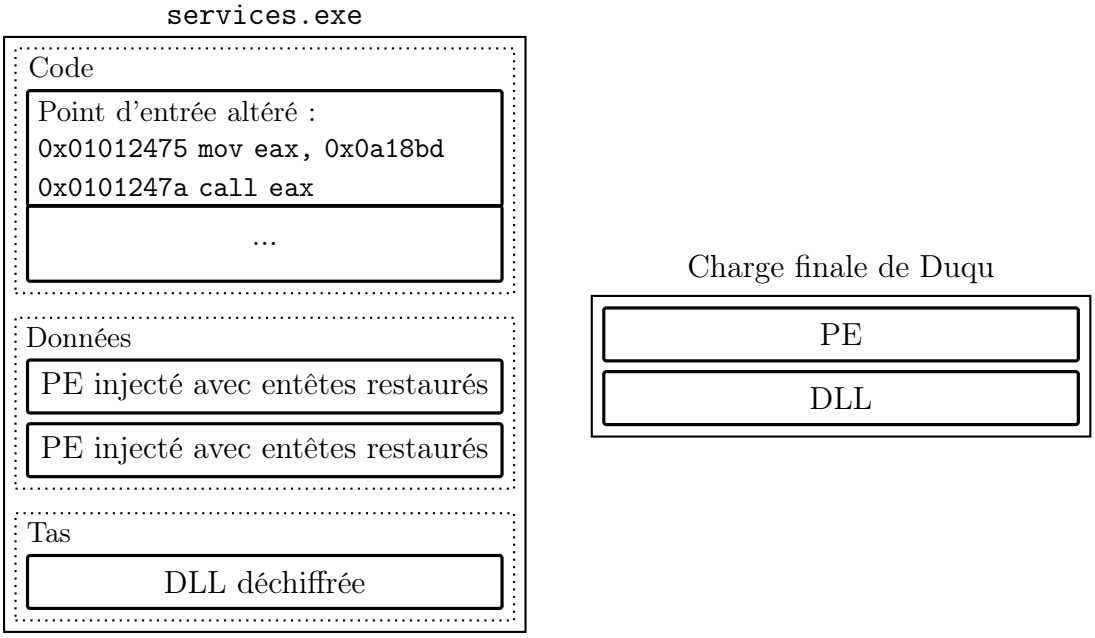


FIGURE 10.7 – Mémoire de `services.exe` et de Duqu une fois que l'injection est complète

10.3 Réalisation d'une version défensive

Nous avons précédemment décrit comment la DLL de Duqu est injectée dans `services.exe`. Certaines des techniques décrites, telle la mise en place de notifications au lancement de chaque module, peuvent être utilisées à des fin défensives.

Schématiquement, le pilote modifié va calculer des signatures sur les binaires chargés en mémoire. Lorsqu'un binaire est lancé, dans le cas où sa signature n'est pas reconnue, il sera considéré suspect et stoppé. Nous détaillons les phases d'initialisation, de mémorisation puis de détection implémentées dans le pilote modifié. Nous terminerons ce chapitre par une présentation détaillée d'une exécution du binaire modifié en situation d'attaque par Duqu.

10.3.1 Phase d'initialisation

La phase d'initialisation d'origine a été grandement allégée pour notre pilote modifié. Nous avons gardé la création des points d'accès, enlevé la recherche de la fonction *ZwProtectVirtualMemory*, que nous n'utiliserons pas. Nous avons conservé le système de notification des chargements de modules en mémoire et avons également demandé de recevoir une notification lorsque le système termine la création d'un processus (à l'aide de la fonction *PsSetCreateProcessNotifyRoutine*).

10.3.2 Phase de mémorisation

Nous avons vu que le point d'entrée n'est pas altéré lors de la première notification. Ainsi si une somme de contrôle est calculée pour le point d'entrée de chaque module,

la modification du point d'entrée peut être détectée lorsque la seconde notification sera déclenchée. Nous avons intégralement repris la fonction de hachage implémentée dans Duqu qui était à l'origine utilisée pour obscurcir le nom des fonctions appelées.

Une première notification est reçue lorsque le processus est créé. Nous récupérons la structure PEB (*Process Environment Block*) associée à ce processus et l'utilisons pour récupérer l'adresse mémoire du module chargé.

Afin de détecter Duqu nous nous focalisons uniquement sur le processus `services.exe` ciblé. Si le nom du processus chargé est "`services.exe`", nous recherchons son point d'entrée, nous calculons une somme de contrôle sur ses huit premiers octets et l'enregistrons comme signature initiale. Le pilote défensif est maintenant prêt à détecter l'injection effectuée par Duqu.

10.3.3 Phase de détection

Lorsqu'un module est chargé, le système passe le contrôle au pilote défensif qui vérifie le point d'entrée du module. Si le module est une DLL, le point d'entrée recherché est celui de l'exécutable auquel la DLL est associée. Nous avons donc ajouté une vérification de la somme de contrôle.

Cette somme de contrôle est comparée à celle calculée en premier lieu. Si elles diffèrent, nous considérons qu'une injection a eu lieu sur `services.exe` entre les deux notifications et, puisque son point d'entrée a été altéré, le processus `services.exe` est considéré comme suspect.

10.3.4 Démonstration

Pour faciliter la mise au point du pilote de détection, nous l'avons installé, ainsi que celui de Duqu, sur une machine de test en suivant la démarche proposée par Sergei Shevchenko [She]. Nous renommons la calculatrice Windows (`calc.exe`) en `services.exe` et observons comment deux pilotes réagissent à son lancement.

Pour cette démonstration, nous avons utilisé deux machines virtuelles sous Windows XP SP3 connectées par un lien série. Une instance de WinDbg [Mic], débogueur fonctionnant sur le noyau Windows, tourne sur la première machine. La seconde machine est lancée en mode "débogage du noyau" l'autorisant à dialoguer avec la première machine afin que celle-ci puisse déboguer les pilotes du noyau.

Lors de ces tests nous avons observé que le pilote `nfrd965.sys` de Duqu vérifie si le système est en mode débogage ou en mode sans échec : nous l'avons alors modifié pour qu'il se lance tout de même en mode débogage. Nous avons également configuré les deux pilotes afin que l'on puisse les lancer sur demande (et non automatiquement au lancement de la machine). Cette configuration nous permet de choisir l'ordre de lancement des pilotes ainsi que de `services.exe`.

Lancement du pilote défensif en premier. Nous lançons en premier le pilote défensif, puis celui de Duqu et enfin `services.exe`. La sortie du débogueur est donnée en figure 10.8 : on voit le chargement de `services.exe`. Le système notifie le pilote défensif

qui enregistre l'identifiant, l'adresse du point d'entrée et la signature des premiers octets du point d'entrée de `services.exe`.

```
-----------------------+* Create process 0x914 *+-----------------------
ProcessImageInformation: PEB=0x7ffd6000, ImageBaseAddress=0x01000000,
                         UniqueProcessId=0x914
Entrypoint bytes at 0x01012475: 0x6a 0x70 0x68 0xe0 0x15 0x00 0x01 0xe8
ProcessImageName: Desktop\services.exe
ProcessImageName: save processID=0x914
CreateProcessNotify: ImageBaseAddress=0x01000000, EntryPoint=0x01012475,
                     EntrypointChecksum=0x49af1bf2
```

FIGURE 10.8 – Sortie de WinDbg. Le processus `services.exe` est chargé : le pilote défensif enregistre son identifiant (`0x914`), son point d'entrée (`0x01012475`) et sa somme de contrôle (`0x49af1bf2`).

Lorsque la notification pour `kernel32.dll` est déclenchée, aucune modification n'a encore été faite puisque Duqu reçoit la notification après le pilote défensif, car nous avons lancé le pilote défensif en premier. La comparaison des sommes de contrôle ne détecte alors pas de différence. Lorsque d'autres DLL liées à `services.exe` sont chargées, le pilote défensif vérifie à nouveau le point d'entrée qui a cette fois été altéré. Le traitement des notifications liées aux DLL `kernel32.dll` et `shell32.dll` est montré en figure 10.9. Ainsi l'altération du point d'entrée est détecté et le pilote défensif prend une décision pour protéger le système : le processus `services.exe` est stoppé, arrêtant la tentative d'infection de la machine.

```
----------* Loaded module \WINDOWS\system32\kernel32.dll *----------
LoadImageNotifyRoutine: ImageBaseAddress=0x7c800000 ProcessId=0x914
-> Verify services.exe process:
   Entrypoint at 0x01012475: 0x6a 0x70 0x68 0xe0 0x15 0x00 0x01 0xe8
-> OK!

----------* Loaded module \WINDOWS\system32\shell32.dll *----------
LoadImageNotifyRoutine: ImageBaseAddress=0x7c9d0000 ProcessId=0x914
-> Verify services.exe process:
   Entrypoint at 0x01012475: 0xb8 0xbd 0x18 0x0a 0x00 0xff 0xd0 0xe8
-> Checksum error !
-> Terminating services.exe
```

FIGURE 10.9 – Détection de l'altération du point d'entrée (`0x01012475`) de `services.exe`.

Lancement du pilote de Duqu en premier. Dans le cas où on lance le pilote du Duqu puis le pilote défensif, la première notification ne provocant pas de modification par le pilote de Duqu, la version défensive calcule toujours la somme de contrôle d'origine de `services.exe`. Lors de la seconde notification, pour `kernel32.dll`, Duqu est injecté et le point d'entrée est modifié. Puis le pilote défensif est également notifié du chargement de `kernel32.dll` et détecte l'altération du point d'entrée, terminant cette fois aussi `services.exe` avant que la charge finale ne soit activée.

Au final, quel que soit l'ordre de chargement des pilotes, le pilote défensif permet d'éviter l'infection. Il est à noter que ce comportement vient de l'utilisation par le pilote

de Duqu de deux notifications pour faire son injection : s'il l'effectuait dès la réception de la première notification, l'ordre de chargement des pilotes deviendrait crucial et, si le pilote de Duqu était lancé en premier, nous ne serions pas capables d'empêcher l'infection.

10.4 Perspectives

Nous sommes capables de détecter l'injection de d'empêcher le chargement de `services.exe` afin d'éviter l'attaque. Cela dit Windows ne peut pas fonctionner normalement sans `services.exe`. La machine ne sera donc pas infectée mais nécessitera une intervention humaine pouvant éventuellement détecter que le programme malveillant injecté est une variante de Stuxnet. Deux contributions supplémentaires pourraient être effectuées. D'une part nous voudrions analyser automatiquement la mémoire du processus infecté, `services.exe`, à la recherche de binaires connus pour être malveillants. Cette étape permettrait de trouver la DLL `NETP191.PNF` de Duqu et de la relier à Stuxnet puisque l'analyseur morphologique est capable de détecter des similarités. Nous n'avons pas implémenté cette analyse. D'autre part nous voudrions être capables de restaurer la version d'origine de `services.exe` afin que le système puisse non seulement éviter l'attaque mais également fonctionner dans un état normal.

10.5 Conclusion

Des similarités entre Duqu et Stuxnet nous ont poussés à nous intéresser à une technique de détection permettant de stopper une attaque par Duqu. Nous avons décrit la technique d'infection ainsi que le fonctionnement du pilote de Duqu et ses méthodes pour rester furtif. Nous avons ensuite reconstruit le code source du pilote de Duqu et en avons fait une version défensive capable de détecter l'injection faite par le logiciel malveillant et de stopper le processus infecté.

Ce travail de décompilation, d'analyse et de construction d'une version défensive de Duqu a principalement été orchestré et réalisé par Fabrice Sabatier que je tiens à remercier.

Conclusion

Synthèse des travaux réalisés

Notre contribution à l'analyse de programmes obscurcis s'est concentrée sur l'analyse de deux techniques de protection : l'auto-modification et le chevauchement de code. Nous avons décrit un langage assembleur disposant d'une sémantique concrète compatible avec l'exécution de programmes auto-modifiants. Nous avons étendu BAP, une plateforme d'analyse de binaires existante, pour lui permettre d'évaluer des programmes auto-modifiants et de séparer des programmes simples en différentes vagues d'exécution. Nous avons formalisé le problème du chevauchement de code et étudié l'usage que les programmes obscurcis font de cette méthode de protection. Enfin nous avons proposé une technique de désassemblage consistant à effectuer une analyse dynamique que l'on complète à l'aide de techniques d'analyse statique. Cette technique a pour objectif de reconstruire un graphe de flot de contrôle dont nous avons défini la forme idéale. Nous avons implémenté un détecteur réalisant l'analyse dynamique à l'aide de l'outil d'instrumentation Pin et reconstruisant le graphe de flot de contrôle paramétré par cette exécution. La pertinence de cet outil a été démontrée lors de l'analyse de programmes obscurcis.

Notre contribution à l'analyse morphologique consiste en la formalisant du sous-problème d'isomorphisme de sous-graphes qu'elle cherche à résoudre. Nous avons proposé une alternative à l'algorithme d'Ullmann, fonctionnant par recherche de parcours au sein d'un graphe de flot. Cette approche est aussi précise que les approches existantes mais est beaucoup plus rapide. Nous l'avons implémentée et avons également décrit et implémenté un second algorithme, incorrect, mais dont le temps d'exécution constant ne dépend pas du nombre de programmes présents dans la base de détection. Nous avons enfin proposé une application de la technique d'analyse morphologique au domaine de la détection de similarités logicielles [Bon+12a ; Bon+12b] et appliqué cette méthode pour analyser quelques logiciels malveillants spécifiques dont Duqu et Stuxnet [Bon+13b ; Bon+13a].

Perspectives

L'analyse hybride que nous avons développée n'utilise que des techniques élémentaires d'analyse statique : elle ne cherche pas à interpréter les instructions trouvées à chaque vague. Notre approche ne permet donc pas de gérer les sauts dynamiques qui n'ont pas été parcourus par l'exécution suivie lors de la phase d'analyse dynamique. Nous voudrions

utiliser des techniques d'interprétation abstraite pour reconstruire plus précisément les vagues et éventuellement être capables de déterminer les potentielles vagues suivantes à partir de la vague courante. D'autre part nous nous sommes concentrés sur l'analyse d'une unique trace d'exécution pour approximer le graphe de flot de contrôle parfait : nous voudrions être capables d'incorporer plusieurs traces différentes dans l'analyse afin de couvrir plus de code potentiellement exécuté.

L'analyse morphologique produit des résultats prometteurs pour la détection de programmes malveillants et de similarités logicielles mais ne prend en compte dans les graphes de flot de contrôle que l'enchaînement des instructions. D'autres critères pourraient être intégrés dans ces graphes, traduisant l'utilisation de certaines méthodes d'obscurcissement : on pourrait ajouter les arcs entre deux sommets illustrant un chevauchement de code. D'autre part nous avons vu que notre méthode réduit la complexité du problème à résoudre, le rendant polynomial, au prix d'une perte de flexibilité dans les résultats qu'elle produit. En particulier, l'inversion de certaines branches dans un graphe conduit à une non détection, ce qui n'aurait pas été le cas si l'on avait gardé le problème d'isomorphisme de sous-graphes. Nous souhaiterions développer et analyser des techniques de réduction et de détection des graphes de flot de contrôle plus résistantes à ces modifications, afin d'être capables de détecter plus de programmes similaires sans trop perdre en vitesse de détection. Il serait alors crucial d'étudier de manière détaillée la précision de notre technique de détection dans des réels cas d'utilisation ainsi que sa résistance à des techniques d'obscurcissement spécifiquement élaborées pour la déjouer. Nous souhaiterions donc déterminer si notre modèle, avec ces modifications, permet de construire un détecteur de logiciels malveillants efficace, précis et rapide lors de la détection, capable de fonctionner sur l'ordinateur d'un utilisateur final.

Bibliographie

[Adl88] L. Adleman. "An abstract theory of computer viruses". *Advances in Cryptology*. Tome 403. Lecture Notes in Computer Science. New York : Springer, 1988, pages 354–374 (cité page 2).

[AMB06] Bertrand Anckaert, Matias Madou et Koen De Bosschere. "A Model for Self-Modifying Code." *Information Hiding*. Tome 4437. Lecture Notes in Computer Science. Springer, 2006, pages 232–248 (cité page 72).

[ANR] ANR. *BinSec*. `http://binsec.gforge.inria.fr/` (cité page 40).

[Bar+11] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary et Aymeric Vincent. "The BINCOA Framework for Binary Code Analysis." *CAV*. Tome 6806. Lecture Notes in Computer Science. Springer, 2011, pages 165–170 (cité page 40).

[Bar+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan et Ke Yang. "On the (Im)Possibility of Obfuscating Programs". *J. ACM* 59.2 (mai 2012), 6 :1–6 :48. doi : `10.1145/2160158.2160159` (cité page 22).

[Bel07] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 7 mai 2007, pages 41–46 (cité page 54).

[BHV11] Sébastien Bardin, Philippe Herrmann et Franck Védrine. "Refinement-Based CFG Reconstruction from Unstructured Programs." *VMCAI*. Tome 6538. Lecture Notes in Computer Science. Springer, 2011, pages 54–69 (cité page 78).

[BKM09] Guillaume Bonfante, Matthieu Kaczmarek et Jean-Yves Marion. "Architecture of a Morphological Malware Detector". *Journal in Computer Virology* 5 (2009), pages 263–270. doi : `10.1007/s11416-008-0102-4` (cité pages 2, 3, 88, 90, 96).

[BMM06] D. Bruschi, L. Martignoni et M. Monga. "Detecting self-mutating malware using control-flow graph matching". *Detection of Intrusions and Malware & Vulnerability Assessment* (2006), pages 129–143 (cité page 92).

[Bon+12a] Guillaume Bonfante, Joan Calvet, Jean-Yves Marion, Fabrice Sabatier et Aurelien Thierry. "Recognition of binary patterns by Morphological analysis". *REcon* (octobre 2012) (cité pages 4, 143, 167).

[Bon+12b] Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier et Aurélien Thierry. "Code synchronization by morphological analysis". *7th International Conference on Malicious and Unwanted Software (Malware)* (octobre 2012) (cité pages 4, 143, 167).

[Bon+13a] Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier et Aurélien Thierry. "Analysis and Diversion of Duqu's Driver". *Malware 2013 - 8th International Conference on Malicious and Unwanted Software* (octobre 2013) (cité pages 4, 153, 167).

[Bon+13b] Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier et Aurélien Thierry. "Duqu contre Duqu : Analyse et détournement du driver de Duqu". *SSTIC - Symposium sur la sécurité des technologies de l'information et des communications*. Rennes, France, juin 2013 (cité pages 4, 153, 167).

[BR13] Zvika Brakerski et Guy N. Rothblum. "Virtual Black-Box Obfuscation for All Circuits via Generic Graded Encoding." *IACR Cryptology ePrint Archive* 2013 (2013), page 563 (cité page 22).

[Bru+11] David Brumley, Ivan Jager, Thanassis Avgerinos et Edward J. Schwartz. "BAP : A Binary Analysis Platform." *Computer Aided Verification (CAV)*. Tome 6806. Lecture Notes in Computer Science. Springer, 2011, pages 463–469 (cité page 40).

[Cal+10] Joan Calvet, Carlton R. Davis, José M. Fernandez, Jean-Yves Marion, Pier-Luc St-Onge, Wadie Guizani, Pierre-Marc Bureau et Somayaji Anil. "The case for in-the-lab botnet experimentation : creating and taking down a 3000-node botnet". *Annual Computer Security Applications Conference*. Austin, Texas, United States, décembre 2010 (cité page 144).

[Cal13] Joan Calvet. "Analyse dynamique de logiciels malveillants". Thèse de doctorat. 2013 (cité pages 1, 3, 19, 35, 47, 54, 75, 80).

[CC77] Patrick Cousot et Radhia Cousot. "Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." *POPL '77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, 1977, pages 238–252 (cité page 78).

[Chr+05] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser et Helmut Veith. *Malware Normalization*. Rapport technique 1539. Wisconsin, USA : University of Wisconsin, Madison, novembre 2005 (cité page 92).

[CJ04] Mihai Christodorescu et Somesh Jha. "Testing Malware Detectors". *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (juillet 2004) (cité page 87).

[Coh86] F. Cohen. "Computer Viruses". Thèse de doctorat. University of Southern California, 1986 (cité page 2).

[Cor+99] L.P. Cordella, P. Foggia, C. Sansone et M. Vento. "Performance Evaluation of the VF Graph Matching Algorithm". *http ://amalfi.dis.unina.it* (1999) (cité page 113).

[Cry11] Laboratory of Cryptography of Systems Security (CrySyS). *Duqu : A Stuxnet-like malware found in the wild*. Octobre 2011 (cité pages 149, 160).

[CVE11] CVE. *CVE-2011-3402*. CVE. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3402`. 2011 (cité page 153).

[CX10] Silvio Cesare et Yang Xiang. "A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost." *AINA*. IEEE Computer Society, 21 mai 2010, pages 721–728 (cité page 91).

[DP10] Saumya K. Debray et Jay Patel. "Reverse Engineering Self-Modifying Code : Unpacker Extraction." *WCRE*. IEEE Computer Society, 2010, pages 131–140 (cité page 54).

[Fer07] P. Ferrie. "Attacks on more virtual machine emulators". *Symantec Technology Exchange* (2007) (cité page 55).

[FSV01] Pasquale Foggia, Carlo Sansone et Mario Vento. "A performance comparison of five algorithms for graph isomorphism". *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*. 2001, pages 188–199 (cité page 113).

[Gar+13] Sanjam Garg, Craig Gentry, Shai Halevi, Raykova Mariana, Amit Sahai et Brent Waters. "Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits." *FOCS*. IEEE Computer Society, 2013, pages 40–49 (cité page 22).

[Hen+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar et Grégoire Sutre. "Lazy Abstraction". *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. 2002. doi : `10.1145/503272.503279` (cité page 72).

[Hexa] Hex-Rays. *Hex-Rays Decompiler*. `http://www.hex-rays.com/products/decompiler/index.shtml` (cité page 155).

[Hexb] Hex-Rays. *IDA*. `https://www.hex-rays.com/products/ida/index.shtml` (cité pages 17, 58, 143).

[IBM] IBM. *IBM Archives : ASCC Reference room*. `http://www-03.ibm.com/ibm/history/exhibits/markI/markI_reference.html` (cité page 10).

[Inta] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual (Volume 2)*. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf` (cité pages 12, 27, 44).

[Intb] INTEL. *XED : X86 Encoder Decoder*. `https://software.intel.com/sites/landingpage/pintool/docs/58423/Xed/html/` (cité pages 44, 78).

[JBV11] Jiyong JANG, David BRUMLEY et Shobha VENKATARAMAN. "BitShred : feature hashing malware for scalable triage and semantic analysis." *ACM Conference on Computer and Communications Security*. ACM, 2011, pages 309–320 (cité page 88).

[JDF08] Grégoire JACOB, Hervé DEBAR et Eric FILIOL. "Behavioral detection of malware : from a survey towards an established taxonomy". *Journal in Computer Virology* 4 (2008), pages 251–266 (cité page 87).

[JLH13] Christopher JÄMTHAGEN, Patrik LANTZ et Martin HELL. "A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries ". 2013 (cité pages 29, 59, 62).

[Kac08] Matthieu KACZMAREK. "Des fondements de la virologie informatique vers une immunologie formelle". Thèse de doctorat. 2008 (cité page 88).

[Kas12] KASPERSKY. *The mystery of Duqu : Part Ten*. `https://www.securelist.com/en/blog/208193425/The_mystery_of_Duqu_Part_Ten`. Mars 2012 (cité page 150).

[KDF09] Nithya KRISHNAMOORTHY, Saumya K. DEBRAY et Keith FLIGG. "Static Detection of Disassembly Errors." *WCRE*. IEEE Computer Society, 2009, pages 259–268 (cité page 57).

[Kin10] Johannes KINDER. "Static analysis of x86 executables." http ://d-nb.info/1008875570. Thèse de doctorat. Darmstadt University of Technology, 2010, pages 1–199 (cité page 59).

[Kir+06] E. KIRDA, C. KRUEGEL, G. BANKS, G. VIGNA et R. KEMMERER. "Behavior-based spyware detection". *Usenix Security Symposium*. 2006 (cité page 87).

[KK12] Johannes KINDER et Dmitry KRAVCHENKO. "Alternating Control Flow Reconstruction." *VMCAI*. Tome 7148. Lecture Notes in Computer Science. Springer, 2012, pages 267–282 (cité page 78).

[KPY07] Min Gyung KANG, Pongsin POOSANKAM et Heng YIN. "Renovo : A Hidden Code Extractor for Packed Executables". *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM)* (2007) (cité pages 77, 80).

[Krü+06] Christopher KRÜGEL, William K. ROBERTSON, Fredrik VALEUR et Giovanni VIGNA. "Static Disassembly of Obfuscated Binaries." *USENIX Security Symposium*. USENIX, 18 septembre 2006, pages 255–270 (cité pages 3, 57, 58).

[KST93] Johannes KÖBLER, Uwe SCHÖNING et Jacobo TORÁN. *The graph isomorphism problem : its structural complexity*. Basel, Switzerland, Switzerland : Birkhauser Verlag, 1993 (cité page 100).

[KV08] Johannes KINDER et Helmut VEITH. "Jakstab : A Static Analysis Platform for Binaries." *CAV*. Tome 5123. Lecture Notes in Computer Science. Springer, 8 juillet 2008, pages 423–427 (cité page 40).

[KV10] Johannes KINDER et Helmut VEITH. "Precise static analysis of untrusted driver binaries." *FMCAD*. IEEE, 2010, pages 43–50 (cité page 40).

[LD03] Cullen LINN et Saumya DEBRAY. "Obfuscation of Executable Code to Improve Resistance to Static Disassembly". *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS '03. Washington D.C., USA : ACM, 2003, pages 290–299. DOI : `10.1145/948109.948149` (cité page 24).

[LPS04] Ben LYNN, Manoj PRABHAKARAN et Amit SAHAI. "Positive Results and Techniques for Obfuscation." *EUROCRYPT*. Tome 3027. Lecture Notes in Computer Science. Springer, 20 avril 2004, pages 20–39 (cité page 22).

[Luk+06] Chi-Keung LUK, Robert S. COHN, Robert MUTH, Harish PATIL, Artur KLAUSER, P. Geoffrey LOWNEY, Steven WALLACE, Vijay Janapa REDDI et Kim M. HAZELWOOD. "Pin : building customized program analysis tools with dynamic instrumentation." *Programming Language Design and Implementation (PLDI)*. ACM, 10 juillet 2006, pages 190–200 (cité page 54).

[MB95] Bruno T. MESSMER et Horst BUNKE. "Subgraph Isomorphism Detection in Polynominal Time on Preprocessed Model Graphs." *ACCV*. Tome 1035. Lecture Notes in Computer Science. Springer, 1995, pages 373–382 (cité page 113).

[Mes95] B.T. MESSMER. "Efficient Graph Matching Algorithms for Preprocessed Model Graph". Thèse de doctorat. Institut für Informatik und angewandte Mathematik, Universität Bern, Switzerland, 1995 (cité pages 107, 113).

[Mic] MICROSOFT. *Standalone Debugging Tools for Windows*. `http://www.microsoft.com/whdc/devtools/debugging/default.mspx` (cité page 163).

[MKK08] Andreas MOSER, Christopher KRUEGEL et Engin KIRDA. "Limits of Static Analysis for Malware Detection." *ACSAC*. IEEE Computer Society, 3 mars 2008, pages 421–430 (cité page 24).

[NC09] J. NAGRA et C. COLLBERG. *Surreptitious Software : Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009 (cité page 21).

[Obj] OBJDUMP. *objdump*. `https://en.wikipedia.org/wiki/Objdump` (cité page 16).

[Ope] OPENSSL. *OpenSSL*. `https://www.openssl.org/` (cité page 144).

[PC03] Manish PRASAD et Tzi cker CHIUEH. "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks." *USENIX Annual Technical Conference, General Track*. USENIX, 3 septembre 2003, pages 211–224 (cité page 58).

[Pre+10] Mila Dalla PREDA, Roberto GIACOBAZZI, Saumya K. DEBRAY, Kevin COOGAN et Gregg M. TOWNSEND. "Modelling Metamorphism by Abstract Interpretation." *SAS*. Tome 6337. Lecture Notes in Computer Science. Springer, 2010, pages 218–235 (cité page 54).

[rad] RADARE. *radare, the reverse engineering framework*. `http://radare.org` (cité pages 16, 59, 143).

[Rey10] Daniel REYNAUD. "Analyse de codes auto-modifiants pour la sécurité logicielle". Thèse de doctorat. 2010 (cité pages 1, 3, 47, 54).

[Roy+07] Paul ROYAL, Mitch HALPIN, David DAGON, Robert EDMONDS et Wenke LEE. "PolyUnpack : Automating the Hidden-Code Extraction of Unpack-Executing Malware." *ACSAC*. IEEE Computer Society, 2 octobre 2007, pages 289–300 (cité pages 77, 80).

[SDA03] Benjamin SCHWARZ, Saumya K. DEBRAY et Gregory R. ANDREWS. "Disassembly of Executable Code Revisited." *WCRE*. IEEE Computer Society, 6 mars 2003, pages 45–54 (cité page 58).

[SH12] Michael SIKORSKI et Andrew HONIG. *Practical Malware Analysis : The Hands-On Guide to Dissecting Malicious Software*. 1st. San Francisco, CA, USA : No Starch Press, 2012 (cité pages 24, 27, 57).

[She] Sergei SHEVCHENKO. *Actually, my name is Duqu - Stuxnet is my middle name*. `http://baesystemsdetica.blogspot.fr/2012/03/actually-my-name-is-duqu-stuxnet-is-my_4108.html` (cité page 163).

[Son+08] Dawn SONG, David BRUMLEY, Heng YIN, Juan CABALLERO, Ivan JAGER, Min Gyung KANG, Zhenkai LIANG, James NEWSOME, Pongsin POOSANKAM et Prateek SAXENA. "BitBlaze : A New Approach to Computer Security via Binary Analysis". *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*. 2008 (cité pages 40, 54).

[Sym11a] SYMANTEC. *W32.Duqu : The Precursor to the Next Stuxnet*. Octobre 2011 (cité page 149).

[Sym11b] SYMANTEC. *W32.Stuxnet Dossier*. Février 2011 (cité page 149).

[Szo05] P. SZOR. *The Art of Computer Virus Defense and Research*. Symantec Press, 2005 (cité page 87).

[Tar71] Robert TARJAN. "Depth-first search and linear graph algorithms". *Foundations of Computer Science, Annual IEEE Symposium on* (1971), pages 114–121. DOI : `10.1109/SWAT.1971.10` (cité page 113).

[Tha11] Amr THABET. *Reversing Stuxnet's Rootkit (MRxNet) Into C++*. `http://amrthabet.blogspot.fr/2011/01/reversing-stuxnets-rootkit-mrxnet-into.html`. Janvier 2011 (cité page 154).

[Thia] Aurélien THIERRY. *Implémentation de l'algorithme par parcours (C++)*. `https://gforge.inria.fr/projects/gtsi/` (cité page 138).

[Thib] Aurélien THIERRY. *Isomorphisme de graphes : implémentation (C)*. `https://gforge.inria.fr/projects/graphbinalgo/` (cité page 138).

[Thic] Aurélien THIERRY. *Site Isomorphism Decision Tree (SIDT) : implémentation (C)*. `https://gforge.inria.fr/projects/sidt/` (cité page 138).

[Tim] Claude TIMSIT. *Du transistor à l'ordinateur* (cité page 10).

[TR05] Dullien THOMAS et Rolf ROLLES. "Graph-based comparison of Executable Objects". *SSTIC - Symposium sur la sécurité des technologies de l'information et des communications*. 2005 (cité page 91).

[TS09] Dullien THOMAS et Porst SEBASTIEN. "REIL : A platform-independent intermediate representation of disassembled code for static code analysis". *CanSecWest*. 2009 (cité page 41).

[Ull76] J.R. ULLMANN. "An algorithm for Subgraph Isomorphism". *Journal of the Association for Computing Machinery, Vol 23, No 1, pages 31-42* (1976) (cité page 104).

[UPX] UPX. *Ultimate Packer for eXecutables*. `http://upx.sourceforge.net/` (cité pages 28, 32).

[UZ13] Jason UPCHURCH et Xiaobo ZHOU. "First byte : Force-based clustering of filtered block N-grams to detect code reuse in malicious software." *MALWARE*. IEEE, 2013, pages 68–76 (cité page 87).

[Wan+00] Chenxi WANG, Jonathan HILL, John KNIGHT et Jack DAVIDSON. *Software tamper resistance : Obstructing static analysis of programs*. Rapport technique. 2000 (cité page 24).

[Weg05] I. WEGENER. *Complexity Theory : Exploring the Limits of Efficient Algorithms*. Springer, 2005 (cité page 100).

[Zyna] ZYNAMICS. *BinDiff*. `http://www.zynamics.com/bindiff.html` (cité pages 41, 91).

[Zynb] ZYNAMICS. *BinNavi*. `http://www.zynamics.com/binnavi.html` (cité page 41).

Annexe A

Désassemblage et détection de `hostname.exe` empaqueté

Nous donnons dans cette annexe les résultats de l'analyse du binaire `hostname.exe` empaqueté par différents logiciels de protection. L'expérience sur `hostname.exe` et ses 34 variantes protégées est décrite à la section 6.6 du chapitre 6 décrivant notre méthode de désassemblage. Pour chaque logiciel de protection la première ligne (en gris) donne des informations globales sur le résultat et chaque ligne supplémentaire détaille une vague de code. Par exemple `hostname.exe` protégé par fsg contient 2 vagues de code. Dans la seconde vague nous avons désassemblé 354 instructions et on y retrouve 100% des sites de `hostname.exe` appris. La trace totale de ce programme contient 240 instructions uniques.

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
(aucun)	/	154	354	/	/	/	/	/	/	254	254	✓
	0	154	354	0	0	1	1	0	0	254	254	✓
armadillo378	(14)	2473	5817	/	/	/	/	/	/	0 (0%)	0 (0%)	✗
Aspack	/	1005	1686	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	681	1286	0	79	1	3	0	31	0 (0%)	0 (0%)	✓
	1	794	1251	0	57	1	3	0	22	0 (0%)	0 (0%)	✓
	2	156	356	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
epprotector03	/	186	386	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	32	32	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
expressor	/	1217	1703	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	1063	1349	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
fsg	/	240	440	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	86	86	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
jdpack2	/	7313	11247	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	558	872	0	458	1	3	0	188	0 (0%)	0 (0%)	✓
	1	6601	10021	0	5014	1	4	0	1817	0 (0%)	0 (0%)	✓
	2	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
molebox	/	7172	16121	/	/	/	/	/	/	254 (100%)	191 (75%)	✓
	0	435	1552	0	799	1	3	0	29	0 (0%)	0 (0%)	✓
	1	6199	13470	0	0	1	1	0	0	5 (2%)	5 (2%)	✓
	2	2129	7491	0	0	1	1	0	0	254 (100%)	191 (75%)	✓
Mystic	/	2465	4221	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	1741	3086	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	192	232	0	16	1	2	0	6	0 (0%)	0 (0%)	✓
	2	378	549	0	9	1	3	0	8	0 (0%)	0 (0%)	✓
	3	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
neolite2	/	1386	2280	/	/	/	/	/	/	254 (100%)	254 (100%)	✓

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	0	1232	1926	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
npack	/	818	1291	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	664	937	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
packman	/	351	561	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	196	206	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	155	355	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
pec2	/	1139	1488	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	17	35	0	3	1	2	0	2	0 (0%)	0 (0%)	✓
	1	136	136	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	2	832	963	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	3	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
pelock	/	4733	8457	/	/	/	/	/	/	211 (83%)	123 (48%)	✓
	0	164	233	0	64	1	3	0	27	0 (0%)	0 (0%)	✓
	1	162	203	0	56	1	2	0	25	0 (0%)	0 (0%)	✓
	2	16	51	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	3	108	108	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	4	261	785	0	291	1	3	0	127	0 (0%)	0 (0%)	✓
	5	394	662	0	196	1	3	0	83	0 (0%)	0 (0%)	✓
	6	235	566	0	193	1	3	0	96	0 (0%)	0 (0%)	✓
	7	215	546	0	285	1	4	0	103	0 (0%)	0 (0%)	✓
	8	385	911	0	409	1	3	0	152	0 (0%)	0 (0%)	✓
	9	242	422	0	216	1	3	0	75	0 (0%)	0 (0%)	✓
	10	385	639	0	265	1	3	0	111	0 (0%)	0 (0%)	✓
	11	1104	1732	0	714	1	3	0	262	0 (0%)	0 (0%)	✓
	12	1360	2842	0	1049	1	3	0	373	0 (0%)	0 (0%)	✓
	13	326	498	0	198	1	3	0	67	0 (0%)	0 (0%)	✓
	14	351	586	0	26	1	2	0	9	211 (83%)	123 (48%)	✓
pespin	/	3104	4693	/	/	/	/	/	/	72 (28%)	67 (26%)	partiel

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	0	7	7	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	6	95	0	25	1	2	0	9	0 (0%)	0 (0%)	✓
	2	97	113	0	29	1	3	0	11	0 (0%)	0 (0%)	✓
	3	70	128	0	23	1	2	0	9	0 (0%)	0 (0%)	✓
	4	139	181	0	27	1	3	0	7	0 (0%)	0 (0%)	✓
	5	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	6	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	7	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	8	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	9	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	10	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	11	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	12	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	13	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	14	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	15	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	16	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	17	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	18	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	19	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	20	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	21	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	22	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	23	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	24	148	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	25	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	26	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	27	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	28	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	29	116	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	30	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	31	113	162	0	17	1	3	0	6	0 (0%)	0 (0%)	✓
	32	171	246	0	24	1	3	0	8	0 (0%)	0 (0%)	✓
	33	23	27	0	6	1	2	0	3	0 (0%)	0 (0%)	✓
	34	4	4	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	35	286	507	0	122	1	3	0	36	0 (0%)	0 (0%)	✓
	36	38	42	0	4	1	2	0	2	0 (0%)	0 (0%)	✓
	37	123	133	0	15	1	2	0	6	0 (0%)	0 (0%)	✓
	38	254	296	12	21	2	2	12	16	0 (0%)	0 (0%)	✓
	39	80	201	0	69	1	3	0	29	0 (0%)	0 (0%)	✓
	40	17	875	0	49	1	2	0	19	0 (0%)	0 (0%)	✓
	41	134	158	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	42	137	141	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	43	81	87	0	10	1	2	0	4	0 (0%)	0 (0%)	✓
	44	2	41	0	3	1	2	0	1	0 (0%)	0 (0%)	partiel
	45	1	1	0	0	1	1	0	0	0 (0%)	0 (0%)	partiel
	46	77	172	0	16	1	2	0	3	0 (0%)	0 (0%)	partiel
	47	25	30	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	48	199	256	8	47	2	3	8	18	0 (0%)	0 (0%)	✓
	49	21	63	0	25	1	3	0	8	0 (0%)	0 (0%)	✓
	50	217	564	0	167	1	3	0	41	0 (0%)	0 (0%)	✓
	51	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	52	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	53	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	54	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	55	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	56	303	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	57	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	58	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	59	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	60	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	61	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	62	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	63	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	64	303	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	65	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	66	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	67	300	426	0	72	1	3	0	26	0 (0%)	0 (0%)	✓
	68	439	594	0	100	1	3	0	38	0 (0%)	0 (0%)	✓
	69	538	685	0	67	1	3	0	22	0 (0%)	0 (0%)	✓
	70	533	685	0	67	1	3	0	22	0 (0%)	0 (0%)	✓
	71	532	685	0	67	1	3	0	22	0 (0%)	0 (0%)	✓
	72	717	853	0	95	1	3	0	34	0 (0%)	0 (0%)	✓
	73	645	842	0	95	1	3	0	34	0 (0%)	0 (0%)	✓
	74	349	481	0	82	1	3	0	31	0 (0%)	0 (0%)	✓
	75	297	603	0	166	1	3	0	35	0 (0%)	0 (0%)	✓
	76	12	132	0	34	1	3	0	14	0 (0%)	0 (0%)	✓
	77	26	54	0	18	1	2	0	3	0 (0%)	0 (0%)	✓
	78	130	135	0	11	1	3	0	6	0 (0%)	0 (0%)	✓
	79	193	449	0	128	1	3	0	26	72 (28%)	67 (26%)	partiel
petite22	/	1589	4185	/	/	/	/	/	/	254 (100%)	215 (85%)	✓
	0	1197	1838	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	228	2129	0	4	1	2	0	4	0 (0%)	0 (0%)	✓
	2	164	2056	0	0	1	1	0	0	254 (100%)	215 (85%)	✓
rlpack	/	464	804	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	310	450	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
Setisoft-271	(0)	0	0	/	/	/	/	/	/	0 (0%)	0 (0%)	✗
svk143f	(27)	9421	14090	/	/	/	/	/	/	0 (0%)	0 (0%)	✗
tElock51	/	1285	1847	/	/	/	/	/	/	254 (100%)	225 (89%)	✓

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	0	40	41	0	3	1	2	0	2	0 (0%)	0 (0%)	✓
	1	11	11	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	2	40	40	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	3	54	54	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	4	984	1345	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	5	156	356	0	0	1	1	0	0	254 (100%)	225 (89%)	✓
tElock92a	/	1717	2455	/	/	/	/	/	/	254 (100%)	225 (89%)	✓
	0	24	29	0	6	1	2	0	4	0 (0%)	0 (0%)	✓
	1	121	192	0	48	1	3	0	14	0 (0%)	0 (0%)	✓
	2	19	20	2	2	2	2	2	2	0 (0%)	0 (0%)	✓
	3	25	25	8	8	2	2	8	8	0 (0%)	0 (0%)	✓
	4	127	149	0	16	1	2	0	6	0 (0%)	0 (0%)	✓
	5	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	6	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	7	13	13	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	8	16	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	9	18	63	0	5	1	2	0	4	0 (0%)	0 (0%)	✓
	10	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	11	15	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	12	14	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	13	56	61	0	2	1	2	0	2	0 (0%)	0 (0%)	✓
	14	154	288	0	60	1	3	0	27	0 (0%)	0 (0%)	✓
	15	180	274	0	32	1	2	0	17	0 (0%)	0 (0%)	✓
	16	509	656	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	17	190	267	0	60	1	2	0	23	0 (0%)	0 (0%)	✓
	18	191	395	0	0	1	1	0	0	254 (100%)	225 (89%)	✓
tElock90	/	1521	2312	/	/	/	/	/	/	254 (100%)	225 (89%)	✓
	0	22	27	0	6	1	2	0	4	0 (0%)	0 (0%)	✓
	1	99	150	0	48	1	2	0	18	0 (0%)	0 (0%)	✓
	2	19	20	2	2	2	2	2	2	0 (0%)	0 (0%)	✓

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	3	25	25	8	8	2	2	8	8	0 (0%)	0 (0%)	✓
	4	56	74	0	11	1	2	0	7	0 (0%)	0 (0%)	✓
	5	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	6	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	7	13	13	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	8	16	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	9	18	19	0	2	1	2	0	1	0 (0%)	0 (0%)	✓
	10	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	11	15	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	12	14	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	13	30	33	0	2	1	2	0	2	0 (0%)	0 (0%)	✓
	14	30	30	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	15	322	1168	0	138	1	3	0	58	0 (0%)	0 (0%)	✓
	16	564	701	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	17	233	480	0	37	1	3	0	14	254 (100%)	225 (89%)	✓
tElock95	(16)	1197	2099	/	/	/	/	/	/	0 (0%)	0 (0%)	✗
tElock96	/	1971	2722	/	/	/	/	/	/	254 (100%)	225 (89%)	✓
	0	24	29	0	6	1	2	0	4	0 (0%)	0 (0%)	✓
	1	140	176	0	53	1	2	0	11	0 (0%)	0 (0%)	✓
	2	19	20	2	2	2	2	2	2	0 (0%)	0 (0%)	✓
	3	25	25	8	8	2	2	8	8	0 (0%)	0 (0%)	✓
	4	274	425	0	26	1	3	0	11	0 (0%)	0 (0%)	✓
	5	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	6	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	7	13	13	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	8	16	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	9	18	19	0	2	1	2	0	1	0 (0%)	0 (0%)	✓
	10	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	11	15	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	12	76	81	0	5	1	2	0	1	0 (0%)	0 (0%)	✓

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	13	196	283	0	6	1	2	0	2	0 (0%)	0 (0%)	✓
	14	222	400	0	28	1	3	0	17	0 (0%)	0 (0%)	✓
	15	513	665	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	16	175	195	0	18	1	2	0	6	0 (0%)	0 (0%)	✓
	17	200	407	0	5	1	2	0	1	254 (100%)	225 (89%)	✓
tElock98	/	1935	2592	/	/	/	/	/	/	254 (100%)	225 (89%)	✓
	0	24	29	0	6	1	2	0	4	0 (0%)	0 (0%)	✓
	1	109	121	0	12	1	2	0	10	0 (0%)	0 (0%)	✓
	2	19	20	2	2	2	2	2	2	0 (0%)	0 (0%)	✓
	3	25	25	8	8	2	2	8	8	0 (0%)	0 (0%)	✓
	4	274	315	0	9	1	2	0	3	0 (0%)	0 (0%)	✓
	5	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	6	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	7	13	13	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	8	16	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	9	18	43	0	4	1	2	0	4	0 (0%)	0 (0%)	✓
	10	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	11	15	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	12	76	81	0	5	1	2	0	1	0 (0%)	0 (0%)	✓
	13	196	283	0	6	1	2	0	2	0 (0%)	0 (0%)	✓
	14	211	364	0	43	1	2	0	15	0 (0%)	0 (0%)	✓
	15	533	691	0	14	1	2	0	2	0 (0%)	0 (0%)	✓
	16	161	219	0	22	1	2	0	12	0 (0%)	0 (0%)	✓
	17	200	404	0	0	1	1	0	0	254 (100%)	225 (89%)	✓
tElock99	/	2044	3022	/	/	/	/	/	/	254 (100%)	225 (89%)	✓
	0	46	130	0	42	1	3	0	16	0 (0%)	0 (0%)	✓
	1	62	106	0	7	1	2	0	2	0 (0%)	0 (0%)	✓
	2	19	20	2	2	2	2	2	2	0 (0%)	0 (0%)	✓
	3	25	25	8	8	2	2	8	8	0 (0%)	0 (0%)	✓
	4	402	624	0	173	1	3	0	73	0 (0%)	0 (0%)	✓

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	5	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	6	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	7	13	13	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	8	16	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	9	18	34	0	2	1	2	0	2	0 (0%)	0 (0%)	✓
	10	15	15	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	11	15	16	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	12	76	80	0	5	1	2	0	2	0 (0%)	0 (0%)	✓
	13	196	283	0	6	1	2	0	2	0 (0%)	0 (0%)	✓
	14	183	401	0	37	1	2	0	17	0 (0%)	0 (0%)	✓
	15	564	727	0	16	1	2	0	2	0 (0%)	0 (0%)	✓
	16	138	178	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	17	226	430	0	0	1	1	0	0	254 (100%)	225 (89%)	✓
Themida18	(12)	75154	88259	/	/	/	/	/	/	0 (0%)	0 (0%)	✗
Themida-203	(6)	5302	6133	/	/	/	/	/	/	0 (0%)	0 (0%)	✗
Upack039	/	437	15249	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	87	14797	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	255	255	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	2	152	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
upx	/	322	523	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	168	169	4	4	2	2	2	2	0 (0%)	0 (0%)	✓
	1	154	354	0	0	1	1	0	0	254 (100%)	254 (100%)	✓
vmprotect	/	264	1021	/	/	/	/	/	/	148 (58%)	148 (58%)	✓
	0	264	1021	0	73	1	3	0	34	148 (58%)	148 (58%)	✓
winupack	/	451	652	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	256	257	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	47	47	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	2	152	352	0	0	1	1	0	0	252 (99%)	252 (99%)	✓
YodaC13	/	857	1549	/	/	/	/	/	/	212 (83%)	141 (56%)	✓
	0	41	41	0	0	1	1	0	0	0 (0%)	0 (0%)	✓

Empaqueteur	Vague	Instructions		Instructions en chevauchement		Couches		Sauts entre couches		Sites de `hostname.exe` détectés		Statut
		Trace	Désas	Trace	Désas	Trace	Désas	Trace	Désas	Parcours	SIDT	
	1	590	670	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	2	45	78	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	3	181	760	0	0	1	1	0	0	212 (83%)	141 (56%)	✓
yp-1.02	/	2165	13315	/	/	/	/	/	/	254 (100%)	254 (100%)	✓
	0	92	117	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	1	1144	1620	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	2	274	541	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	3	688	1269	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	4	113	164	0	0	1	1	0	0	0 (0%)	0 (0%)	✓
	5	171	10511	0	9525	1	2	0	0	254 (100%)	254 (100%)	✓

Résumé

Cette thèse porte en premier lieu sur l'analyse et le désassemblage de programmes malveillants utilisant certaines techniques d'obscurcissement telles que l'auto-modification et le chevauchement de code. Les programmes malveillants trouvés dans la pratique utilisent massivement l'auto-modification pour cacher leur code utile à un analyste. Nous proposons une technique d'analyse hybride qui utilise une trace d'exécution déterminée par analyse dynamique. Cette analyse découpe le programme auto-modifiant en plusieurs sous-parties non auto-modifiantes que nous pouvons alors étudier par analyse statique en utilisant la trace comme guide. Cette seconde analyse contourne d'autres techniques de protection comme le chevauchement de code afin de reconstruire le graphe de flot de contrôle du binaire analysé.

Nous étudions également un détecteur de programmes malveillants, fonctionnant par analyse morphologique : il compare les graphes de flot de contrôle d'un programme à analyser à ceux de programmes connus comme malveillants. Nous proposons une formalisation de ce problème de comparaison de graphes, des algorithmes permettant de le résoudre efficacement et détaillons des cas concrets d'application à la détection de similarités logicielles.

Mots-clés : programmes malveillants, auto-modification, graphe de flot de contrôle, comparaison de graphes

Abstract

This dissertation explores tactics for analysis and disassembly of malwares using some obfuscation techniques such as self-modification and code overlapping. Most malwares found in the wild use self-modification in order to hide their payload from an analyst. We propose an hybrid analysis which uses an execution trace derived from a dynamic analysis. This analysis cuts the self-modifying binary into several non self-modifying parts that we can examine through a static analysis using the trace as a guide. This second analysis circumvents more protection techniques such as code overlapping in order to recover the control flow graph of the studied binary.

Moreover we review a morphological malware detector which compares the control flow graph of the studied binary against those of known malwares. We provide a formalization of this graph comparison problem along with efficient algorithms that solve it and a use case in the software similarity field.

Keywords: malwares, self-modification, control flow graph, graph comparison