Université de Lorraine
Laboratoire Lorrain de Recherche en Informatique et ses Applications - UMR 7503
École Doctorale IAEM Lorraine

Analyse Dynamique de Logiciels Malveillants

Thèse présentée et soutenue publiquement le 23 août 2013 pour l'obtention du
Doctorat de l'Université de Lorraine
(spécialité informatique)

par
Joan Calvet

Rapporteurs :
Marc Dacier, Ph.D., Symantec Recherche
Jean Goubault-Larrecq, Professeur, ENS de Cachan

Examinateurs :
José M. Fernandez, Professeur, École Polytechnique de Montréal
Jean-Yves Marion, Professeur, École des Mines de Nancy
Ettore Merlo, Professeur, École Polytechnique de Montréal

REMERCIEMENTS

J'ai eu la chance de partager mes pauses café avec de nombreuses personnes durant ce doctorat, qui ont toutes eu un impact positif sur mon travail, que cela soit en France ou au Canada. Sans prétendre à l'exhaustivité, merci à Aurélien, Daniel, Fabrice, Guillaume, Hugo, Isabelle, Matthieu, Philippe, Thanh Dinh et Wadie, qui ont rendu mes séjours à Nancy toujours stimulants et agréables. Du côté de Montréal, un grand merci à Alireza, Antoine, Antonin, Axel, Bernard, Erwann, Étienne, Fanny, Gabriel, Jean-François, Mahshid, Pierre, Pier-Luc, Philippe, Simon et Xavier.

Je remercie en particulier Antonin et Erwann pour leur support dans la mise en place des expériences.

Je remercie Pierre-Marc et Nicolas, pour avoir toujours pris le temps de donner leur avis sur mes projets et pour m'avoir constamment encouragé durant ces quatre années.

Merci à Jean-Yves et José de m'avoir donné la chance de réaliser ce doctorat, et pour m'avoir guidé à travers les méandres académiques.

Un grand merci à mes parents, mon frère et toute ma famille, pour leur soutien durant toutes mes années d'étude.

Finalement, un grand merci à Claire pour m'avoir supporté, dans tous les sens du terme, tout le long de ce doctorat.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES ET ABRÉVIATIONS

AES	Advanced Encryption Standard
BB	Bloc Basique
BFD	Boucles avec Flux de Données
CISC	Complex Instruction Set Computing
CC	Command & Control
GFC	Graphe de Flot de Contrôle
MVS	Microsoft Visual Studio
RI	Représentation Intermédiaire
RTEA	Russian Tiny Encryption Algorithm
SIMD	Single Instruction Multiple Data
SR	Super Répéteur
SSA	Single Static Assignment
TEA	Tiny Encryption Algorithm
xCAT	Extreme Cloud Administration Toolkit

CHAPITRE 1

INTRODUCTION

1.1 Contexte

Définitions. Un logiciel malveillant est un programme informatique qui réalise volontairement une action allant à l'encontre de l'intérêt de son utilisateur. Conceptuellement, cela se traduit par la combinaison d'un *moyen de propagation* et d'une *charge utile*. Cette dernière est la partie du logiciel malveillant qui réalise lors de l'exécution son objectif final, comme récupérer des informations bancaires, ou encore envoyer des pourriels. Par exemple, *Melissa*, un logiciel malveillant rendu célèbre par son grand nombre de victimes à la fin des années 90, commençait par effacer des fichiers aléatoirement choisis sur le disque de la victime – la charge utile – puis s'envoyait en pièce jointe d'un courriel aux 40 premières personnes du carnet d'adresses – le moyen de propagation.

De nos jours il est devenu commun pour les machines sous le contrôle d'un logiciel malveillant de recevoir des ordres par Internet, et de former alors ce qu'on appelle un *botnet*, un réseau de machines zombies. L'opérateur de ce réseau peut ainsi coordonner les actions des machines infectées, par exemple pour déclencher des attaques de déni de service contre des sites Internet, en les sollicitant plus qu'ils ne peuvent le supporter.

Prolifération. La dernière décennie a vu une augmentation substantielle du nombre de logiciels malveillants propagés sur Internet. La compagnie AV-TEST déclare par exemple avoir collecté *plus de 30 millions de nouveaux fichiers malveillants pour l'année 2012* [5], alors qu'ils en avaient moins d'un million en 2005. De son côté, l'éditeur antivirus Kaspersky annonce avoir détecté plus de 200 000 fichiers malveillants *par jour* durant la fin de l'année 2012 [83].

Il convient toutefois de relativiser ces nombres astronomiques par le fait qu'ils ne représentent pas des logiques malveillantes différentes, mais des fichiers. En particulier, ceux-ci peuvent en fait être membres d'une même *famille*, c'est-à-dire dériver d'une base de code commune, et donc provenir d'un même auteur. Certaines familles ont notamment la capacité d'infecter des programmes bénins, afin que ceux-ci transportent ensuite leur charge utile malveillante, créant alors des milliers de fichiers différents. De plus, il est devenu commun pour les logiciels

malveillants de ne pas être distribués sous leur forme d'origine, mais d'être d'abord transformés par l'ajout de protections contre l'analyse humaine et la détection par les antivirus. Cette transformation a habituellement la propriété de pouvoir produire à partir d'un même fichier initial un ensemble de fichiers syntaxiquement différents ayant tous le même comportement.

Afin de rendre compte de la production de logiciels malveillants il est donc préférable de raisonner à partir de la notion de famille, plutôt que celle de fichier. Par exemple, l'outil de suppression de logiciels malveillants de Microsoft s'attaquait à 61 familles en 2006 [17], alors qu'en mai 2013 il en cible désormais 165 [102]. Ces familles sont sélectionnées à cause de leur forte prévalence sur les ordinateurs avec un système d'exploitation Windows. Au moment de l'écriture de cette thèse, l'éditeur antivirus ESET référence environ 1 500 familles de logiciel malveillants dans son encyclopédie [59], dont 38 nouvelles ont été ajoutées de mars à mai 2013. Ainsi, des logiciels malveillants continuent d'être créés, et cette production ne donne pas de signe d'essoufflement.

Motivations. Ces logiciels malveillants sont des créations humaines, et leur prolifération est donc due à des raisons propres aux humains. S'il fut une époque où l'ego des auteurs de logiciels malveillants était leur principale motivation, ce sont désormais des considérations plus matérialistes qui les occupent. Par exemple, des motivations géopolitiques sont récemment apparues au grand jour, telles que l'espionnage [98], ou encore l'attaque d'infrastructures critiques [65]. Mais c'est le gain financier aux dépens de simples utilisateurs, ou d'entreprises, qui demeure l'objectif de la majorité des logiciels malveillants rencontrés sur Internet. C'est sur ces logiciels malveillants « grand public » – qui ne ciblent pas de victimes particulières – que nous nous focaliserons dans cette thèse.

Professionnalisation. Comme toute organisation dont l'objectif est le profit, la production de logiciels malveillants a subi diverses transformations dans le but de maximiser ses gains. Ainsi, il est devenu courant qu'un tel logiciel soit conçu, distribué et opéré par des groupes différents, qui se spécialisent pour augmenter leur efficacité, et vendent alors leur savoir-faire comme un service. Par exemple, la Figure 1.1 présente un extrait d'un ensemble de règles rédigé par les opérateurs du réseau de machines zombies *Waledac*, que nous étudierons dans cette thèse.

Ces règles dénotent une relation quasi contractuelle entre les opérateurs du *botnet* et les personnes chargées de l'infection de nouvelles machines, celles-ci n'appartenant donc pas au même groupe. En particulier, la rémunération dépend de la qualité des machines amenées

Règles du système *FairMoney*	
1. Toute personne qui a commencé à distribuer le programme accepte tacitement les présentes règles. Toute personne qui enfreint ces règles peut se voir refuser le paiement pour les téléchargements.	**2.** La politique des prix est basée sur la durée de vie moyenne de 1 000 robots : *(i)* moins de 2 heures : non payés, *(ii)* plus de 2 heures : 25$, *(iii)* plus de 4 heures : 50$, *(iv)* plus de 8 heures : 100$, *(v)* plus de 24 heures : 200$
3. Ne jamais décharger notre programme de la mémoire de l'ordinateur, supprimer le fichier ou les clés de registre. Les comptes qui seront pris en train de le faire seront bloqués sans paiement.	**4.** Le montant versé doit être un multiple de 1 000 téléchargements, c'est-à-dire que si vous avez 10 567 téléchargements et que vous demandez le paiement, nous allons vous payer pour 10 000, et les 567 téléchargements restants seront payés la prochaine fois.
5. Le programme peut être déployé dans le monde entier (bien que quand 99% des machines sont en Chine, cela nous rend nerveux).	**6.** Nous nous réservons le droit, si nécessaire, de modifier les règles du système *FairMoney*.

Figure 1.1 Extrait des règles rédigées par les opérateurs de *Waledac* à destination de ceux chargés de sa propagation (traduites approximativement du russe par nos soins)

dans le réseau (règle 2).

Recherche en sécurité. En réponse à cette sophistication des logiciels malveillants, la recherche s'est développée ; la base de référence Google Scholar dénombre environ 4 000 articles contenant les mots « *malicious software* » ou « *computer virus* » publiés avant l'année 2000, alors qu'il y en a eu plus de 17 000 depuis. Cette productivité scientifique découle en partie du fait que de nombreux groupes de recherche ont trouvé dans les logiciels malveillants un nouveau champ d'application pour leurs outils. Par exemple, les *botnets* peuvent être étudiés comme des réseaux informatiques classiques, et les logiciels malveillants eux-mêmes peuvent profiter des méthodes d'analyse de programmes développées pour les applications légitimes.

1.2 Problèmes de recherche

Dans ce contexte, l'objectif principal de cette thèse est de **développer des méthodes aidant à la compréhension des logiciels malveillants**. Il s'agit donc de permettre à un être humain – que nous appellerons « l'analyste » – d'appréhender plus facilement les logiciels malveillants.

Cette compréhension prend d'abord place au niveau des programmes eux-mêmes, ce qui est un problème abordé de longue date par la recherche académique, et définie par Biggerstaff *et al.* [13] de la façon suivante : « *A person understands a program when he or she is able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program* ». Mais les méthodes habituelles de compréhension de programmes présupposent l'accès au code source, c'est-à-dire à la forme du programme qui a servi à le créer, tandis que les logiciels malveillants sont seulement disponibles dans une forme exécutable par un ordinateur, c'est-à-dire écrite en langage *machine*. Cette dernière a été produite par un compilateur à partir du code source, dans l'objectif d'être efficacement exécutée sur un ordinateur, et non dans celui d'être comprise par un être humain. Ainsi, de nombreuses abstractions n'existent plus et doivent être reconstruites pour permettre à l'humain de comprendre le programme. Ce processus porte le nom de *rétro-ingénierie*.

Au-delà des difficultés inhérentes à l'étude des programmes exécutables, les logiciels malveillants sont souvent rendus délibérément complexes à comprendre par leurs auteurs, notamment par l'utilisation de *protections*. Celles-ci transforment le logiciel malveillant en une forme « hors-normes » qui mettra alors en difficulté les méthodes d'analyse standards. L'étude de ces protections sera l'objet de la première partie de cette thèse, avec deux objectifs de recherche, tout d'abord réaliser une cartographie des protections de logiciels malveillants, et ensuite identifier les algorithmes cryptographiques employés par ces protections.

D'un autre côté, l'utilisation quasi systématique d'Internet pour envoyer des ordres aux ordinateurs infectés, c'est-à-dire la création de *botnets*, rend nécessaire la construction d'une approche globale qui ne se contente pas de considérer *une* machine infectée par un logiciel malveillant, mais un *ensemble* de machines infectées évoluant dans un même réseau. Ces réseaux ont évolué sous différentes formes ces dernières années, qui vont d'architectures centralisées à des architectures complètement décentralisées, le tout avec des protocoles de communication qui sont souvent construits spécifiquement pour l'occasion. La création d'une méthodologie expérimentale d'étude des réseaux de machines zombies sera l'objet de la deuxième partie de ce mémoire, avec un cas pratique sur le *botnet* de la famille de logiciel malveillant *Waledac*.

Pour les deux problématiques décrites précédemment, nous avons fait le choix d'appliquer *l'analyse dynamique*, c'est-à-dire d'étudier une exécution réelle des logiciels malveillants concernés. Ceci permet de passer à travers de nombreuses difficultés inhérentes à l'analyse des lo-

giciels malveillants et permet d'atteindre un niveau de précision et de réalisme élevé. Nous allons maintenant présenter chacun des objectifs de recherche spécifiques de cette thèse, avec les contributions que nous avons réalisées.

1.2.1 Cartographie expérimentale des protections de logiciels malveillants

Problématique. Une des raisons du nombre conséquent de fichiers malveillants propagés sur Internet réside dans la façon dont ceux-ci sont protégés. En effet, l'ajout d'une protection permet de créer un nouveau fichier, tout en conservant la logique du programme original. De nombreuses applications légitimes emploient également des protections, dans le but de préserver leur propriété intellectuelle. Dans le cas des logiciels malveillants, l'objectif est surtout d'éviter leur détection, à la fois en produisant des fichiers différents – pour garder une longueur d'avance sur les antivirus –, et en introduisant des techniques agressives pour mettre en difficulté les outils d'analyse.

La compréhension de ces protections est donc d'un grand intérêt, que cela soit pour aider à la détection, ou pour accéder à la charge utile. Il convient alors de déterminer les caractéristiques des protections de logiciels malveillants, afin de développer des méthodes d'analyse adaptées.

État de l'art. De nombreuses techniques de protection existent : cela peut aller de la traduction du code à protéger dans un autre langage (la protection se chargeant alors de l'interprétation lors de l'exécution), à l'utilisation de chiffrement pour camoufler le code et les données. La connaissance des types de protections privilégiés par les logiciels malveillants est très limitée. En effet, que cela soit dans l'industrie [16, 106] ou dans le milieu académique [9, 121], le recensement de ces protections se fait essentiellement avec des outils tels que PEiD [130], SigBuster [85] ou Yara [154], qui parcourent *statiquement* un programme à la recherche de signatures présentes dans leur base de référence. Cette approche ne permet pas de saisir clairement les caractéristiques des protections, pour les raisons suivantes :

- Les protections identifiées sont celles pour lesquelles une signature a été ajoutée, ce qui privilégie naturellement les protections connues et anciennes, tandis que les plus confidentielles ou les nouvelles versions sont invisibles.
- De par leur nature statique, ces outils ne peuvent reconnaître que la couche extérieure d'un programme ; si une partie du code de la protection n'est visible que lors de l'exécution, elle ne sera pas analysée.
- De nombreux faux positifs peuvent avoir lieu, c'est-à-dire qu'une protection peut être identifiée à tort dans un programme. Il est même possible de créer ces faux positifs délibérément

et automatiquement, avec l'outil ExeForger [138].
Finalement, le résultat donné par ces outils est le nom d'une protection, ce qui ne fournit pas directement d'indications sur le fonctionnement de celle-ci (les protections étant souvent peu documentées). Ainsi, il n'existe pas à notre connaissance d'étude en profondeur des protections de logiciels malveillants.

Objectif. À la suite de plusieurs analyses de logiciels malveillants réalisées au cours de cette thèse [33, 34], nous avons pu dégager un modèle de protection qui semble être privilégié par ceux-ci. Ce modèle est essentiellement basé sur le *code auto modifiant*, c'est-à-dire la capacité d'un programme à créer ses propres instructions lors de l'exécution, celles-ci étant alors invisibles pour l'observateur qui se contenterait d'étudier le programme sans l'exécuter. L'ensemble d'instructions qui est créé dynamiquement est appelé une *couche de code*. Nous avons alors posé l'objectif de recherche suivant :

Objectif de recherche 1. *Développer une méthode expérimentale pour valider l'hypothèse selon laquelle le modèle de protection suivant est particulièrement prévalent pour les logiciels malveillants :*
- *Il y a au moins deux couches de code auto modifiant dans le programme,*
- *Le rôle des couches dépend de leur ordre de création :*
  - *Les premières couches, c'est-à-dire toutes sauf la dernière, servent à la protection de la charge utile,*
  - *La dernière couche implémente la charge utile du logiciel malveillant.*

Contributions. Dans le cadre de ce travail, nous avons d'abord construit un cadre semi-formel pour discuter la notion de protection. Ensuite, nous avons développé une expérience à grande échelle pour étudier en profondeur les protections de logiciels malveillants. En particulier, nous avons mesuré les caractéristiques de 600 exemplaires choisis grâce à des critères mesurant le succès de leurs familles. Cela nous a permis de mettre en avant les caractéristiques de leurs protections, et de valider en grande partie notre hypothèse. Ces résultats sont décrits au chapitre 4.

1.2.2 Identification de fonctions cryptographiques dans les protections de logiciels malveillants

Problématique. L'auto modification d'un programme, c'est-à-dire la création d'instructions lors de son exécution, est une technique habituelle des logiciels malveillants pour se protéger. Une façon de l'implémenter consiste à utiliser un *algorithme cryptographique* pour déchiffrer

lors de l'exécution une partie du programme à partir d'un texte chiffré et d'une clé. La cryptographie – la science du chiffrement – permet alors aux logiciels malveillants de se protéger en ne révélant leur logique interne qu'au moment de l'exécution.

Afin d'éviter le travail fastidieux de construction d'un algorithme cryptographique, les auteurs de logiciels malveillants choisissent souvent des algorithmes du domaine public. Et s'il leur arrive de faire preuve d'originalité afin de rendre l'analyse plus compliquée, ils ont alors tendance à réutiliser leurs créations de multiples fois. L'analyste peut ainsi se constituer une base de connaissances d'algorithmes susceptibles de se trouver dans les protections de logiciels malveillants, soit parce qu'ils appartiennent au domaine public, soit parce qu'ils ont déjà été utilisés à cet effet.

La capacité de reconnaître automatiquement un algorithme connu dans un programme, ce que nous appelons son *identification*, est alors d'une grande utilité. Premièrement, cela évite à l'analyste d'avoir à étudier le code concerné pour le comprendre, puisqu'il dispose déjà d'une description de haut niveau de l'algorithme. Ceci est particulièrement appréciable lorsque le programme est obfusqué afin de rendre sa compréhension difficile. Ensuite, la reconnaissance des paramètres cryptographiques devient plus facile et, une fois ceux-ci trouvés, le processus de déchiffrement peut être effectué sans exécuter le logiciel malveillant. Lorsque ce processus est inclus dans un antivirus, cela lui permet d'ignorer les couches de protections en se focalisant sur la partie déchiffrée, souvent plus facile à reconnaître, car moins propice au changement. Finalement, comme toute caractéristique de haut niveau d'un logiciel malveillant, la connaissance des algorithmes cryptographiques peut servir à la classification.

État de l'art. Il existe déjà un certain nombre d'outils pour identifier une implémentation d'un algorithme cryptographique dans un programme en langage machine [87, 70, 131, 4]. Ils recherchent des caractéristiques comme des constantes particulières, ou des successions d'instructions machines, de façon *statique*. De ce fait, ces outils sont habituellement inefficaces sur les protections de logiciels malveillants, puisque leur nature statique ne leur permet pas d'analyser au-delà de la première couche de code. De plus, l'obfuscation du code fait disparaître les signes habituels d'un algorithme comme les constantes.

Plus récemment, des approches dynamiques, c'est-à-dire basées sur une exécution du code, ont été développées [69, 157], mais seulement pour des programmes ayant été produits par des compilateurs standards. Ainsi, il n'existe actuellement pas de méthode adaptée à l'identification d'implémentations obfusquées d'algorithmes cryptographiques.

Objectif. Les algorithmes cryptographiques implémentés dans les protections de logiciels malveillants étant souvent documentés, il est intéressant de chercher des moyens de les identifier automatiquement. Comme le code de ces protections est habituellement obfusqué, cela nécessite une approche spécifique afin de se focaliser sur les caractéristiques qui sont conservées dans toutes les implémentations.

Objectif de recherche 2. *Développer une méthode pour identifier automatiquement une implémentation d'un algorithme cryptographique connu dans un programme en langage machine obfusqué.*

Contributions. Nous avons développé une méthode originale d'identification des algorithmes cryptographiques adaptée au contexte des programmes obfusqués. Ainsi, nous avons pu identifier avec succès diverses implémentations obfusquées de l'algorithme cryptographique *Tiny Encryption Algorithm (TEA)*, et nous avons montré que les auteurs des protections des logiciels malveillants *Storm Worm* et *Silent Banker* ont fait une erreur en l'implémentant (contrairement à ce qui est mentionné dans des analyses publiques). Nous avons également identifié de nombreuses implémentations de *RC4*, chose totalement impossible à faire avec les outils existants du fait de l'absence de signes particuliers dans cet algorithme. Encore une fois, notre méthode a été résistante face à des techniques d'obfuscation diverses, notamment celles rencontrées dans la protection du logiciel malveillant *Sality*. De plus, nous avons testé avec succès notre méthode sur des implémentations d'algorithmes tels que *AES* et *MD5*, rencontrées dans des logiciels malveillants, ou bien encore sur des exemples synthétiques protégés par des protections commerciales. Finalement, nous avons exploré la généralisation de notre approche en identifiant la multiplication modulaire à la base de *RSA*, mais également des cas tels que la combinaison ou la modification d'algorithmes cryptographiques.

Ces résultats ont été publiés en 2012 à la conférence *ACM Conference on Computer and Communications Security (CCS)* [38], ainsi que dans les conférences industrielles *Reverse-engineering Conference (REcon)* [30] et *Hackito Ergo Sum (HES)* [31]. Ils sont décrits dans cette thèse au chapitre 5.

1.2.3 Analyse des réseaux de machines zombies

Problématique. De nos jours, la très grande majorité des familles de logiciels malveillants font rentrer les machines qu'elles infectent dans un réseau – un *botnet* – par lequel une entité distante – un *Command & Control (CC)* – va envoyer des ordres et récupérer des informations. Ce contrôle à distance permet aux auteurs de logiciels malveillants de maximiser

l'utilisation des machines infectées, et donc leurs gains.

Ces réseaux sont constitués de machines hétérogènes et réparties dans différents pays qui communiquent souvent avec des protocoles construits pour l'occasion. Leur compréhension est donc un défi pour l'analyste, et ne peut se faire de façon complète qu'à un niveau *global*, c'est-à-dire en prenant en compte le réseau et pas seulement une machine infectée isolée.

État de l'art. Une première façon d'aborder le problème des *botnets* consiste à les étudier directement sur Internet, par exemple en les infiltrant [25, 135, 120], ou même en prenant le contrôle du CC [136]. Cela permet de collecter des informations en temps réel sur leurs actions et de mieux comprendre leurs objectifs. D'un autre côté, il est difficile de cacher ce type de recherche aux contrôleurs du *botnet*, qui pourraient alors se défendre. Ensuite, on ne peut pas reproduire facilement les expériences, puisqu'on ne contrôle pas le réseau. De ce fait, les observations sont peu fiables – on peut étudier un cas particulier sans s'en rendre compte –, et il est difficile d'explorer différentes configurations. De plus, toutes les variables expérimentales ne sont pas sous le contrôle de l'analyste. Finalement, cette approche pose des problèmes évidents d'éthique, comme l'interaction avec des utilisateurs infectés non consentants ou encore la participation à des activités criminelles.

Deuxièmement, d'autres travaux se focalisent sur la construction d'un modèle théorique d'un *botnet*, afin de le simuler [49, 141]. Cela permet d'estimer l'impact de divers paramètres expérimentaux, tout en ayant la possibilité de reproduire facilement les expériences à une grande échelle. Mais la construction d'un modèle est une tâche ardue, du fait de l'hétérogénéité des machines et des communications qui constituent un *botnet*. De plus, ces réseaux emploient souvent des protocoles de communication construits pour l'occasion [34, 63, 25], difficiles à modéliser en l'absence d'une description de haut niveau. Ainsi, il est difficile de faire des observations réalistes à l'aide de la simulation de *botnets*.

Finalement, une troisième approche consiste à émuler un *botnet*, c'est-à-dire à le reproduire au complet avec de *véritables machines infectées* dans un environnement contrôlé [8, 81]. Cela permet de répéter des expériences à volonté, tout en contrôlant ou mesurant un maximum de variables expérimentales. De plus, le fait de travailler en isolement permet d'explorer les différents paramètres expérimentaux sans risque de prévenir les contrôleurs du *botnet*. Ensuite, de nombreux problèmes de réalisme se trouvent résolus par l'utilisation de véritables machines infectées, car il n'est alors pas nécessaire de reproduire le comportement du logiciel malveillant. Finalement, il n'y a pas de problèmes éthiques, puisque l'expérience n'a pas lieu

sur Internet. D'un autre côté, l'émulation de *botnets* est une approche coûteuse en ressources, puisqu'elle nécessite d'avoir à disposition un laboratoire capable d'accueillir un réseau d'une taille importante.

Objectifs. De par la rigueur scientifique et le niveau d'éthique qu'elle permet, l'émulation apparaît comme une voie de recherche prometteuse pour comprendre les *botnets* de façon globale. Les premiers travaux dans ce domaine souffrent de limitations (échelle réduite, pas de véritables logiciels malveillants) qu'il faut dépasser pour pouvoir étudier les *botnets*.

Objectif de recherche 3. *Montrer la faisabilité de l'émulation de botnets à grande échelle.*

Dans des travaux précédents la thèse nous avions conjecturé la vulnérabilité du *botnet* pair-à-pair *Waledac* à une attaque permettant sa prise de contrôle [34]. La construction d'un environnement d'émulation de *botnets* était alors l'occasion d'explorer la faisabilité de cette attaque.

Objectif de recherche 4. *Valider la vulnérabilité du botnet Waledac à une attaque permettant sa prise de contrôle et explorer les paramètres qui influent sur sa réussite.*

Contributions. Nous avons développé une méthode complète pour mener de telles expériences, que cela soit au niveau matériel, logiciel ou des procédures. Nous avons appliqué notre approche expérimentale à un cas d'étude, le *botnet* pair-à-pair *Waledac*, que nous avons émulé avec 3 000 machines. Cela nous a permis d'implémenter une attaque contre le réseau afin d'en prendre le contrôle. Nous avons pu reproduire les expériences de nombreuses fois et ainsi nous assurer de leur validité. Finalement, nous avons découvert que le choix de distribuer une même clé cryptographique aux machines infectées n'était pas une grossière erreur de la part des opérateurs de *Waledac*, mais plutôt un compromis pour optimiser les performances du réseau.

Différentes parties de ces résultats ont été publiés en 2010 dans les conférences *Annual Computer Security Applications Conference (ACSAC)* [36], *Cyber Security Experimentation and Test (CSET)* [35] et *Virus Bulletin (VB)* [37]. Ils sont décrits dans cette thèse aux chapitres 6 et 7.

1.3 Organisation de la thèse

Cette thèse s'articule en deux parties distinctes, la première comporte quatre chapitres sur les protections de logiciels malveillants, tandis que la deuxième contient deux chapitres sur

les réseaux de machines zombies.

Dans la première partie, les chapitres 2 et 3 introduisent le contexte de cette recherche et la méthode d'analyse choisie. Ils sont donc avant tout tournés vers la pédagogie et posent des fondations. Les chapitres 4 et 5 présentent nos deux projets de recherche, le premier contient la cartographie expérimentale des protections, tandis que le second présente notre méthode d'identification des algorithmes cryptographiques.

Dans la deuxième partie, le chapitre 6 introduit la méthode d'étude choisie pour aborder les réseaux de machines zombies, l'émulation, tandis que le chapitre 7 présente le cas d'étude de *Waledac*.

Première partie

PROTECTIONS DE LOGICIELS MALVEILLANTS

CHAPITRE 2

PROGRAMMES EN LANGAGE MACHINE x86

Les logiciels malveillants étant des programmes informatiques, on pourrait être tenté de les définir formellement à partir d'un quelconque langage de programmation. Mais cela serait oublier qu'un logiciel malveillant est analysé avant tout dans sa forme exécutable, c'est-à-dire qu'il n'a pas été créé *dans cet état* par son concepteur, mais dans une représentation de plus haut niveau qui a ensuite été traduite par un compilateur pour obtenir la forme actuelle. Ainsi, caractériser les logiciels malveillants avec un langage autre que celui de la machine qui les exécute revient à construire une abstraction ; or cette construction n'est pas évidente en pratique, comme nous le verrons.
Nous commencerons donc par décrire l'environnement d'exécution des logiciels malveillants à bas niveau, pour ensuite insister sur les particularités de cet environnement par le biais d'un modèle simplifié. Finalement, nous présenterons une revue de littérature sur les notions abordées dans ce chapitre.

2.1 Fondements

Le langage machine est spécifique à une certaine architecture. C'est pourquoi nous présentons d'abord l'architecture x86, puis le langage machine associé, pour ensuite introduire une machine abstraite qui imite de façon simplifiée cette architecture. Finalement, nous définirons un langage assembleur pour cette machine abstraite, et nous montrerons comment il peut être traduit en langage machine.

2.1.1 Machine x86

Architecture

L'architecture x86 est à la base d'une série de processeurs apparus dans la continuité de l'Intel 8086 depuis les années 80, et elle constitue aujourd'hui encore l'architecture majoritaire des ordinateurs personnels. Son organisation suit « le modèle de von Neumann » [145], qui divise une machine en quatre parties :

1. **L'unité de calcul**, chargée des opérations arithmétiques et logiques. En particulier, un ensemble de drapeaux permettent de connaître l'état du dernier calcul, par exemple si le résultat produit était zéro.

2. **L'unité de contrôle**, qui dirige le flux d'exécution dans le programme par le biais d'un pointeur d'instruction désignant la prochaine instruction à exécuter.
3. **La mémoire**, qui contient à la fois les instructions et les données du programme, et dont l'unité d'indexation est l'octet. En pratique le mécanisme de mémoire virtuelle permet de considérer chaque programme s'exécutant sur la machine comme l'unique occupant de la mémoire. De plus, il y a différents niveaux de mémoire, et en particulier des registres qui sont des mémoires rapides de faible capacité.
4. **La gestion des entrées-sorties**.

Langage machine

Le jeu d'instructions exécutables sur une machine x86 contient plus de 700 éléments [80]. Ces instructions machine sont de taille variable et leur conception suit le principe du *Complex Instruction Set Computing (CISC)*, c'est-à-dire qu'elles tendent à avoir une sémantique complexe et à être déclinées en nombreuses variantes. Une instruction machine x86 peut prendre en mémoire jusqu'à 15 octets, qui suivent le format spécifié dans la Figure 2.1, c'est-à-dire :
– Un ou plusieurs octets de préfixes optionnels, qui permettent de préciser l'opération (accès partagés, répétitions...).
– Un ou plusieurs octets pour décrire l'opération elle-même (*opcode*).
– Plusieurs octets optionnels pour décrire les opérandes.

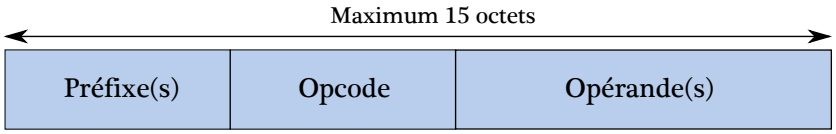


Figure 2.1 Format d'une instruction machine x86 [80]

Par la suite, nous noterons $\mathcal{X}86$ l'alphabet des instructions machine, et $\mathcal{X}86^*$ l'ensemble des mots construits à partir de cet alphabet, c'est-à-dire les séquences d'instructions machine.

2.1.2 Machine abstraite

À des fins pédagogiques, il serait délicat de présenter dans cette thèse des exemples de programmes directement dans le langage *machine* décrit précédemment. D'un autre côté, un langage de trop haut niveau nous ferait perdre de vue les spécificités de l'architecture. C'est pourquoi nous avons fait le choix de construire une machine abstraite – pour laquelle nous définirons ensuite un langage bas niveau mais humainement compréhensible –, qui est essentiellement une simplification de la machine x86. En d'autres termes, les caractéristiques

de cette machine abstraite sont celles d'une machine x86, mais, à l'inverse, toutes les spécificités de cette dernière ne sont pas présentes, seulement celles qui sont nécessaires à notre raisonnement. Plus précisément, la machine abstraite est constituée des éléments suivants :
– Un pointeur d'instruction désignant la prochaine instruction à exécuter,
– Un ensemble de six registres de 32 bits, $REG = \{eax, ebx, ecx, edx, esi, edi\}$,
– Un drapeau indiquant si le dernier calcul a donné zéro, ou non,
– Une mémoire adressable en 32 bits, où code et données sont mélangés.

Nous pouvons alors définir formellement l'état de la machine, ce qui nous servira à raisonner sur des exécutions de programmes. Nous notons $\mathbb{B}^i$ l'ensemble des valeurs de i bits.

Definition 1. *L'état d'une machine abstraite est un quadruplet* $(ip, \Delta, \mathcal{Z}, \Sigma)$*, où :*
– $ip \in \mathbb{B}^{32}$ *est la valeur du pointeur d'instruction,*
– $\Delta : REG \to \mathbb{B}^{32}$ *est l'état des registres,*
– $\mathcal{Z} \in \mathbb{B}^1$ *est l'état du drapeau,*
– $\Sigma : \mathbb{B}^{32} \to \mathbb{B}^8$ *est l'état de la mémoire.*

L'ensemble des états possibles d'une machine abstraite sera noté *MSTATE*. L'accès aux relations Σ et Δ se note avec des [.], par exemple $\Sigma[a]$ retourne la valeur stockée à l'adresse a en mémoire. La mise à jour de ces relations se note sous la forme $\Sigma[a \mapsto v]$, pour indiquer que la case mémoire située à l'adresse a contient désormais la valeur v, les autres cases restant inchangées. De plus, la mise à jour de n cases mémoire à partir d'une adresse a sera notée $\Sigma[a_{/n} \mapsto v]$ pour $v \in \mathbb{B}^{8\times n}$, tout comme l'accès qui sera noté $\Sigma[a_{/n}]$.

Remarque 1. Nous avons choisi de ne pas inclure de pile dans notre machine abstraite, car cela n'est pas nécessaire à notre raisonnement.

2.1.3 Langage assembleur réduit

Un langage assembleur est une traduction littérale du langage machine, c'est-à-dire qu'une instruction assembleur correspond à une instruction machine, rendant ce langage lui aussi spécifique à l'architecture. L'intérêt de travailler avec un tel langage est de pouvoir exhiber des exemples précis de programmes bas niveau, tout en restant humainement compréhensible. Un langage assembleur pour l'architecture x86 est défini informellement dans les manuels Intel [80], et il en existe en pratique plusieurs variantes. Nous définissons ici notre propre langage, qui – avec la même motivation que pour la définition de la machine abstraite – est simplement un sous ensemble du langage assembleur défini par Intel. Après avoir exhibé sa syntaxe, nous montrerons comment on peut le traduire dans un langage compréhensible par notre machine abstraite, puis comment celle-ci l'exécute, par le biais de sa sémantique opérationnelle.

Syntaxe

Nous utiliserons le langage assembleur *RASM* dont la syntaxe est donnée par la grammaire de la Figure 2.2.

```
          <program>  ::=  'DATA:' <data-decl> 'CODE:' <code>
        <data-decl>  ::=  val <data-decl> | ε
            <code>   ::=  <instruction> <code> | ε
     <instruction>   ::=  <data-transfer> | <arithmetic-logic> | <branch> |
                          'cmp r_i, r_j' | 'halt'
   <data-transfer>   ::=  'mov [val],r_i' | 'mov r_i,[val]' | 'mov r_i,val' |
                          'mov [r_i],r_j' | 'mov r_i,[r_j]'
<arithmetic-logic>   ::=  'xor r_i,r_j' | 'add r_i,val'
          <branch>   ::=  'jmp val' | 'jnz val'
```

Figure 2.2 Grammaire du langage *RASM* avec $val \in \mathbb{B}^{32}$, $r_i \in REG$ et $r_j \in REG$

Ainsi, un programme *RASM* commence par une section de données – déclarées par bloc de 4 octets – suivie d'une section de code. Tous les accès mémoires se font sur des valeurs de 4 octets. Les données et les instructions sont référencées par leurs adresses, et non par des noms de variables ou des labels. La sémantique de ces instructions sera donnée par la suite. Finalement, nous notons $RASM_{/Ins}$ la réduction du langage *RASM* à ses instructions.

Compilation sur la machine abstraite

Notre machine abstraite n'exécute pas directement le langage *RASM*, mais un langage machine. Pour pouvoir définir la sémantique opérationnelle de notre langage assembleur sur la machine, il est donc nécessaire d'exhiber d'abord la fonction de compilation des programmes *RASM* en programmes machines. Habituellement une telle fonction *traduit* le programme en langage machine, et c'est une seconde fonction qui s'occupe de le *charger* dans la machine. Afin de simplifier le discours, nous combinons ici les deux fonctions, c'est-à-dire que notre fonction de compilation traduit un programme *RASM* en langage machine et retourne l'état initial de la machine pour exécuter ce programme.

Traduction d'un programme RASM : Comme le langage *RASM* n'est qu'un sous-ensemble du langage assembleur Intel – et pour garder apparentes les spécificités du jeu d'instructions x86 – nous avons choisi de conserver l'encodage de la documentation Intel [80]. Ainsi, chaque instruction *RASM* a une traduction dans le langage machine décrit en §2.1.1. Les données

sont quant à elles stockées en *big-endian*, c'est-à-dire avec l'octet de poids fort en premier. La fonction partielle $INS(ip, \Sigma)$ retourne l'instruction *assembleur* commençant à l'adresse ip dans la mémoire Σ de la machine. De plus, l'opération $|\ .\ | : RASM_{/Ins} \to \mathbb{N}$ retourne la longueur en octets de l'encodage en langage machine d'une instruction assembleur.

Organisation dans la machine : La fonction de compilation $\mathscr{C} : RASM \to MSTATE$ transforme un programme $RASM$ en l'état initial de la machine abstraite qui va exécuter le programme.

Soit $P \in RASM$, alors $\mathscr{C}(P) = (ip_0,\ \Delta_0^P,\ 0,\ \Sigma_0^P)$, avec :
- $ip_0 = \texttt{0x100}$,
- $\Delta_0^P[r_i] = 0, \forall r_i \in REG$,
- Σ_0^P contient l'encodage des données de P à partir de l'adresse `0x0`, et l'encodage du code à partir de l'adresse `0x100`.

Remarque 2. Toutes les valeurs numériques mentionnées dans cette thèse sont représentées sous forme hexadécimale, ce qui est noté avec le préfixe « `0x` ».

Remarque 3. La valeur `0x100` a été choisie arbitrairement pour être le point d'entrée de nos programmes. Cela suppose que les données n'occupent pas plus de `0x100` octets. Dans un véritable programme exécutable, l'adresse du point d'entrée est déclarée dans son en-tête.

Sémantique

La sémantique opérationnelle [110] des instructions $RASM$ pour notre machine abstraite est définie dans la Figure 2.3. L'effet d'une instruction est décrit par une relation de transition $S \Rightarrow S'$ qui représente le fait que l'exécution à partir de l'état de la machine S engendre l'état S'. Nous notons $\perp$ l'arrêt de la machine, et nous utilisons les opérations arithmétiques et logiques usuelles sur $\mathbb{B}^{32}$.
La sémantique d'un programme $RASM$ se déduit alors de façon canonique par composition de la sémantique de ses instructions.

Exemple

Afin d'expliciter le processus de compilation décrit précédemment, la Figure 2.4 présente un exemple de programme $RASM$ et du résultat de sa compilation (seul l'état de la mémoire

$$
(ip, \Delta, \mathcal{Z}, \Sigma) \Rightarrow \begin{cases}
(ip',\ \Delta,\ \mathcal{Z},\ \Sigma[val_{/4} \mapsto \Delta[r_i]]) & \text{si } INS(ip,\Sigma) = \texttt{`mov [}val\texttt{],}r_i\texttt{'} \\
(ip',\ \Delta,\ \mathcal{Z},\ \Sigma[\Delta[r_i]_{/4} \mapsto \Delta[r_j]]) & \text{si } INS(ip,\Sigma) = \texttt{`mov [}r_i\texttt{],}r_j\texttt{'} \\
(ip',\ \Delta[r_i \mapsto \Sigma[val_{/4}]],\ \mathcal{Z},\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`mov }r_i\texttt{,[}val\texttt{]'} \\
(ip',\ \Delta[r_i \mapsto \Sigma[\Delta[r_j]_{/4}]],\ \mathcal{Z},\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`mov }r_i\texttt{,[}r_j\texttt{]'} \\
(ip',\ \Delta[r_i \mapsto val],\ \mathcal{Z},\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`mov }r_i\texttt{,}val\texttt{'} \\
 & \\
(ip',\ \Delta[r_i \mapsto \Delta[r_i] \oplus \Delta[r_j]],\ 0,\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`xor }r_i\texttt{,}r_j\texttt{'} \\
 & \text{et } \Delta[r_i] \oplus \Delta[r_j] \neq 0 \\
(ip',\ \Delta[r_i \mapsto \Delta[r_i] \oplus \Delta[r_j]],\ 1,\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`xor }r_i\texttt{,}r_j\texttt{'} \\
 & \text{et } \Delta[r_i] \oplus \Delta[r_j] = 0 \\
(ip',\ \Delta[r_i \mapsto \Delta[r_i] + val],\ 0,\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`add }r_i\texttt{,}val\texttt{'} \\
 & \text{et } \Delta[r_i] + val \neq 0 \\
(ip',\ \Delta[r_i \mapsto \Delta[r_i] + val],\ 1,\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`add }r_i\texttt{,}val\texttt{'} \\
 & \text{et } \Delta[r_i] + val = 0 \\
 & \\
(ip', \Delta, 0, \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`cmp }r_i\texttt{,}r_j\texttt{'} \\
 & \text{et } \Delta[r_i] - \Delta[r_j] \neq 0 \\
(ip', \Delta, 1, \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`cmp }r_i\texttt{,}r_j\texttt{'} \\
 & \text{et } \Delta[r_i] - \Delta[r_j] = 0 \\
 & \\
(val,\ \Delta,\ \mathcal{Z},\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`jnz }val\texttt{'} \\
 & \text{et } \mathcal{Z} = 0 \\
(ip',\ \Delta,\ \mathcal{Z},\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`jnz }val\texttt{'} \\
 & \text{et } \mathcal{Z} = 1 \\
(val,\ \Delta,\ \mathcal{Z},\ \Sigma) & \text{si } INS(ip,\Sigma) = \texttt{`jmp }val\texttt{'} \\
\bot & \text{si } INS(ip,\Sigma) = \texttt{`halt'}
\end{cases}
$$

Figure 2.3 Sémantique opérationnelle des instructions du langage *RASM* avec $ip' = ip + |INS(ip, \Sigma)|$

est montré). Le programme réalise une boucle qui applique quatre fois un ou-exclusif à deux valeurs, initialisées à partir de deux données.

$\mathscr{C}(P)$		P
a	$\Sigma[a_{/k}],\ k \in \mathbb{N}$	
		`DATA:`
`0x0`	`00 00 00 17`	`0x17`
`0x4`	`00 00 00 42`	`0x42`
`...`	`00 00 ...`	
		`CODE:`
`0x100`	`A1 00 00 00 00`	`mov eax, [0x0]`
`0x105`	`8B 1D 04 00 00 00`	`mov ebx, [0x4]`
`0x10A`	`33 C9`	`xor ecx, ecx`
`0x10C`	`03 C3`	`xor eax, ebx`
`0x10E`	`83 C1 01`	`add ecx, 0x1`
`0x111`	`83 F9 04`	`cmp ecx, 0x4`
`0x114`	`75 F6`	`jnz 0x10C`
`0x116`	`F4`	`halt`

Figure 2.4 Compilation de $P \in RASM$ sur la machine abstraite

2.2 Défis posés à l'analyse

Pour comprendre la complexité du domaine d'étude de cette thèse, il nous faut maintenant décrire trois défis particuliers de l'analyse des programmes en langage machine x86 : le code auto modifiant, les particularités du jeu d'instructions et l'absence d'informations de haut niveau.

2.2.1 Code auto modifiant

Comme indiqué en 2.1.1, le « modèle de von Neumann », qui a servi de base à l'architecture x86, ne fait pas de distinction entre les instructions et les données en mémoire. Cette absence de dichotomie permet la création de *programmes auto modifiants*, c'est-à-dire de programmes qui génèrent lors de l'exécution certaines de leurs instructions. Plus précisément, nous englobons dans le code auto modifiant toute création de nouvelles instructions, que celle-ci se fasse à la place d'anciennes instructions ou non. Par exemple, le programme de la Figure 2.5

écrit à l'adresse `0x150` la valeur `0xF4`, qui est l'encodage en langage machine de l'instruction « `halt` », puis va exécuter cette instruction nouvellement écrite.

$\mathscr{C}(P)$		P
a	$\Sigma[a_{/k}],\ k \in \mathbb{N}$	
		`DATA:`
`0x0`	`00 00 ...`	
		`CODE:`
`0x100`	`B8 00 00 00 F4`	`mov eax, 0xF4000000`
`0x105`	`A3 00 02 00 00`	`mov [0x150], eax`
`0x10A`	`EB 44`	`jmp 0x150`
`...`	`00 00 ...`	
`0x150`	`00 00 ...`	

Figure 2.5 Programme *RASM* auto modifiant

Le code auto modifiant n'est pas pris en compte par de nombreux modèles de calcul classiques. Ceux-ci considèrent en effet que les instructions d'un programme sont toutes définies initialement et que le code n'évoluera donc pas durant une exécution. Ainsi, dans notre exemple de la Figure 2.5, une analyse du programme qui se conterait de considérer les instructions assembleurs statiquement définies n'aurait pas conscience que l'instruction « `halt` » va être exécutée. La problématique peut être rapidement amplifiée lorsque les nouvelles instructions créent elles-mêmes d'autres instructions, comme dans les protections de logiciels malveillants que nous verrons au chapitre 4. Dans de telles conditions il devient difficile de reconstruire précisément le flot de contrôle d'un programme, c'est-à-dire de déterminer ses possibles chemins d'exécutions.

D'un autre côté, il existe des utilisations légitimes au code auto modifiant, comme :
- la compilation « juste à temps » [15, 92], qui est utilisée par des interpréteurs pour traduire un bloc d'instructions à interpréter en langage machine au moment de l'exécution. Ceci permet d'augmenter les performances en supprimant le coût de l'interprétation séparée de chaque instruction (tout en rajoutant un délai de compilation).
- la compression de programmes exécutables [113], par laquelle un programme est transformé en une version compressée contenant du code additionnel qui se chargera lors de l'exécution de la décompression (et qui créera donc à ce moment de nouvelles instructions).

L'absence de distinction code-données pourrait être corrigée, et le code auto modifiant être donc interdit, pour peu que des droits d'accès permettent de séparer les zones mémoires exécutables des zones écrivables. De tels droits ont en fait déjà été créés, que ça soit au niveau du système d'exploitation, ou du processeur lui-même. Mais leur utilisation reste limitée en pratique, notamment du fait de la prévalence de vieilles technologies, et de l'existence des utilisations légitimes précédemment mentionnées. Ainsi, il est nécessaire pour l'analyse de programmes en langage machine de prendre en compte le code auto modifiant.

Remarque 4. La possibilité d'avoir du code auto modifiant n'est pas limitée aux programmes en langage machine. Par exemple certains langages interprétés autorisent leurs programmes à faire appel à l'interpréteur dynamiquement (comme Python et Javascript avec la fonction `eval()`), et de ce fait leur permettent de créer des instructions à l'exécution [116].

Remarque 5. À propos de l'absence de distinction code-données, on trouve dans la description originelle de l'architecture par von Neumann [145] cette remarque : « *While it appeared that various parts of this memory have to perform functions which differ somewhat in their nature and considerably in their purpose, it is nevertheless tempting to treat the entire memory as one organ, and to have its parts even as interchangeable as possible [...]* ».

2.2.2 Jeu d'instructions

L'analyse de programmes en langage machine x86 est confrontée à plusieurs difficultés liées aux instructions machine elles-mêmes. Ces problèmes se reportent dans les instructions assembleurs, puisque ces dernières n'en sont qu'une simple traduction. Parmi ces difficultés, on peut citer :

- **Le nombre élevé d'instructions**, notamment à cause de la rétrocompatibilité, qui implique que toutes les instructions définies depuis la création de l'architecture ont été conservées, tandis que de nombreuses autres ont été ajoutées au fil des années. Il y a désormais plus de 700 instructions x86, avec des dizaines de combinaisons d'opérandes possibles pour chacune [80].
- **La complexité des instructions**, par exemple il en existe une pour calculer une valeur de contrôle par l'algorithme CRC32, et une autre pour faire une ronde de chiffrement de l'algorithme cryptographique AES [80]. Il est alors difficile de décrire précisément la sémantique de ces instructions, tel que nous avons pu le faire pour celles de *RASM* en Figure 2.3, et donc de raisonner sur les programmes qui les utilisent.
- **La multiplicité des opérandes**, en particulier car beaucoup d'instructions ont des opérandes implicites, c'est-à-dire non mentionnées comme telles dans l'encodage en langage

machine. Par exemple l'instruction assembleur x86 « `div` *`val`* » utilise implicitement les registres *eax* et *edx* pour stocker la valeur à diviser par *`val`* , et pour recevoir respectivement le quotient, et le reste, de la division.

– **Les accès indirects à la mémoire**, notamment par le biais d'un registre. Cela entraîne la possibilité d'avoir des branchements indirects, de la forme « `jmp [`r_i`]` », ou encore d'avoir la même adresse mémoire manipulée par différents registres. Pour analyser précisément un programme, il est alors nécessaire de connaître les valeurs exactes des registres.

– **L'absence d'alignement des instructions en mémoire**, ce qui permet leur chevauchement. Par exemple la Figure 2.6 présente un programme *RASM* dont la dernière instruction est un saut au milieu de l'instruction précédente, ce qui a pour effet d'exécuter le code machine `0xF4`, correspondant à l'instruction assembleur « `halt` ».

$\mathscr{C}(P)$		P
a	$\Sigma[a_{/k}],\ k \in \mathbb{N}$	
		`DATA:`
`0x0`	`00 00 ...`	
		`CODE:`
`0x100`	`B8 F4 00 00 00`	`mov eax, 0xF4`
`0x105`	`EB FA`	`jmp 0x101`

Figure 2.6 Exemple de programme *RASM* avec chevauchement d'instructions. L'octet souligné est celui qui sera exécuté après le branchement.

– **Les singularités du jeu d'instructions x86**, comme des instructions non officiellement documentées, mais qui existent bel et bien [43], ou encore des instructions assembleurs qui peuvent être encodées de plusieurs manières [61].

En pratique, une analyse pourrait limiter ces difficultés en se focalisant sur les instructions les plus courantes, par exemple celles produites par les compilateurs modernes. Mais nous verrons au chapitre 4 que les protections de logiciels malveillants utilisent justement un jeu d'instructions varié afin de mettre en difficulté l'analyse.

2.2.3 Perte d'informations de haut niveau

L'analyse de programmes sous la forme de code source de haut niveau peut s'appuyer sur un certain nombre d'informations qui, à l'inverse, sont absentes dans les programmes exécutables :

– **Le typage des données :** les données stockées en mémoire n'ont pas de types qui leur sont associés, elles sont simplement contenues dans un grand tableau (Σ dans notre machine abstraite). En particulier, les adresses stockées en mémoire ne sont pas distinguables des autres données.
– **La structure du code :** les langages de haut niveau définissent un ensemble de structures de contrôle (boucles, tests...) qui restreignent la façon dont le flot de contrôle évolue au cours de l'exécution. En particulier, ces langages n'autorisent habituellement pas d'instructions de branchement du type `goto`, ou alors seulement des versions restreintes. À l'inverse, en langage machine x86, ce type d'instruction est permis, par exemple sous la forme « `jmp` r_i ». Ainsi, il est possible de produire des formes de code non structurées en programmant directement en assembleur, par exemple du code « spaghetti » comme dans le programme *RASM* de la Figure 2.7. Dans cet exemple, il est difficile de trouver la valeur qu'aura `eax` à la fin de l'exécution (et même de voir comment termine l'exécution).

$\mathscr{C}(P)$		P
a	$\Sigma[a_{/k}],\ k \in \mathbb{N}$	
		`DATA:`
`0x0`	`00 00 ...`	
		`CODE:`
`0x100`	`B8 07 00 00 00`	`mov eax, 0x7`
`0x105`	`EB 06`	`jmp 0x10D`
`0x107`	`83 C0 12`	`add eax, 0x12`
`0x10A`	`EB 06`	`jmp 0x112`
`0x10C`	`F4`	`halt`
`0x10D`	`83 C0 03`	`add eax, 0x3`
`0x110`	`EB F5`	`jmp 0x107`
`0x112`	`83 C0 15`	`add eax, 0x15`
`0x115`	`EB F5`	`jmp 0x10C`

Figure 2.7 Exemple de programme *RASM* avec flot de contrôle « spaghetti ». Le code va sauter de l'adresse `0x105` à `0x10D`, puis de `0x110` à `0x107`, de `0x10A` à `0x112`, et finalement de `0x115` à `0x10C`.

– **Les fonctions :** cette notion très commune dans les langages haut niveau n'est pas définie en langage machine x86. Il existe simplement des instructions, comme « `call` » et « `ret` », qui facilitent l'implémentation d'un appel de fonction, en se chargeant notamment du retour au code appelant. Néanmoins, rien n'oblige à implémenter un appel de fonction avec

ces instructions, ou à suivre une quelconque norme.

Cette perte d'information est intrinsèque au processus de compilation dont l'objectif est de produire un programme exécutable par une machine.

Remarque 6. Il existe des mécanismes pour conserver de l'information de haut niveau dans des programmes exécutables, par exemple les symboles de débogage, qui permettent de faire le lien entre le code source et le code machine. Pour les logiciels malveillants, il serait bien entendu illusoire de supposer que leurs concepteurs fournissent ces informations.

2.3 Revue de littérature

Nous présentons ici les travaux sur la traduction du langage machine vers un langage assembleur et sur la définition formelle du langage machine.

2.3.1 Désassemblage

Avant de donner des références sur le désassemblage, nous allons définir le cadre théorique de ce problème.

Cadre théorique

Le désassemblage est la transformation d'un programme en langage machine en un programme en langage assembleur au comportement équivalent. Une condition nécessaire à toute transformation de programme non triviale est de pouvoir distinguer le code des données dans le programme d'entrée. Cela n'est pas un problème quand on compile un programme écrit dans un langage de haut niveau, car le code et les données sont alors différenciés *syntaxiquement*. À l'inverse, dans un programme en langage machine, le code et les données sont mélangés en mémoire sans être à priori distinguables.

Or, la distinction code-données est un problème indécidable dans un programme informatique, ce qui peut se montrer en se ramenant au problème de l'arrêt :

– Supposons que le problème de la distinction code-données est décidable, c'est-à-dire qu'il existe une routine $CODE(P)$ qui retourne pour tout programme P l'ensemble des adresses mémoires exécutables par P.

– Quelque soient P un programme et I une entrée de P, on peut alors construire le programme P' qui réalise un appel au programme P sur l'entrée I. Notons A l'adresse qui est située juste après cet appel dans P'.

– Si $A \in CODE(P')$, cela implique alors que le programme P termine nécessairement son exécution sur l'entrée I. Inversement, si $A \notin CODE(P')$, cela implique que P ne termine pas son exécution sur l'entrée I. Ce raisonnement étant valable pour tout P et tout I, nous pouvons décider le problème de l'arrêt, qui est notoirement indécidable. De ce fait, notre supposition de départ est fausse, c'est-à-dire que le problème de la distinction code-données est indécidable.

Comme le désassemblage nécessite de distinguer le code des données, on peut conclure à l'indécidabilité du désassemblage d'un programme en langage machine.

Solutions pratiques

Il existe en pratique deux algorithmes de désassemblage conventionnels :

– *le parcours linéaire*, qui commence au premier octet du programme et désassemble séquentiellement les instructions, et qui ne distingue donc pas les données qui peuvent se trouver au milieu du code.

– *le parcours récursif*, qui suit le flot de contrôle du programme pour trouver les instructions, mais qui de ce fait peut rapidement devenir imprécis à cause d'adresses de branchement calculées à partir des entrées du programme.

Des améliorations à ces algorithmes ont été étudiées, par exemple Cifuentes *et al.* [40] proposent un désassemblage « spéculatif » dans lequel les zones non reconnues comme du code lors d'une première passe récursive sont désassemblées jusqu'à rencontrer une instruction invalide, auquel cas la zone est considérée comme contenant des données. De même, Schwarz *et al.* [126] proposent une méthode hybride qui combine les deux algorithmes classiques en vérifiant qu'ils donnent des résultats compatibles et en remontant les erreurs à l'analyste lorsque ça n'est pas le cas. De leur côté, Wartell *et al.* [150] emploient l'apprentissage machine pour reconnaître les séquences d'octets qui sont habituellement du code, à partir d'un ensemble de binaires dont ils ont validé manuellement le désassemblage. Finalement, le désassembleur commercial le plus utilisé, IDA Pro [56], emploie un algorithme récursif et de nombreuses heuristiques basées sur la reconnaissance du compilateur. Ce qui fait particulièrement sa force est son interactivité, qui permet à l'analyste de corriger le désassemblage selon sa propre

connaissance.

À la base du désassemblage se trouve le décodage d'une instruction machine, c'est-à-dire sa traduction en assembleur. Or l'implémentation de cette – apparemment simple – opération est propice aux erreurs sur un jeu d'instructions aussi varié que celui de l'architecture x86, comme l'ont montrés Paleari *et al.* [114]. Les auteurs ont ainsi exhibé des erreurs de décodage pour *tous* les désassembleurs du marché, notamment en comparant leurs résultats avec le comportement du processeur.

2.3.2 Sémantique du langage machine x86

Le raisonnement formel sur un programme se base habituellement sur une sémantique de ses instructions. Or les manuels Intel [80] ne donnent celle des instructions machine x86 que dans un mélange d'anglais et de pseudo-code, qui est donc difficile à utiliser rigoureusement. Ainsi, des formes plus adaptées ont été créées par différents auteurs. Cifuentes *et al.* [39] ont défini un langage, nommé *Semantic Specification Language*, permettant de décrire la sémantique des instructions machine. Il s'agit d'une extension au langage Object-Z, qui permet la spécification formelle en combinant la logique de premier ordre et les ensembles mathématiques. Coogan *et al.* [47] raisonnent de leur côté sur les instructions machine x86 en les traduisant en équations ; chaque instruction correspond alors à une, ou plusieurs, équations qui font ressortir explicitement ses effets sur le système. Degenbaev [52] définit dans sa thèse un langage spécifique qui lui permet de construire la sémantique de plus d'une centaine d'instructions x86, en lien avec un modèle mémoire précis pour décrire les accès partagés. Dans leur étude sur l'exécution multiprocesseur, Sarkar *et al.* [124] définissent la sémantique d'une vingtaine d'instructions machine dans un langage fonctionnel leur permettant de préciser l'ordre (ou l'absence d'ordre) entre les différents effets d'une instruction. Finalement, Cai *et al.* [28] construisent une machine abstraite dans laquelle le code du programme n'est pas fixé et qui peut donc prendre en compte le code auto modifiant. Les auteurs présentent alors une sémantique opérationnelle pour un jeu d'instruction x86 réduit, sous une forme similaire à celle utilisée en 2.1.1, et raisonnent sur les programmes selon la logique de Hoare.

Une des utilisations principales de la sémantique est la traduction du langage machine en une *Représentation Intermédiaire (RI)*, plus adaptée à l'analyse. La sémantique se retrouve alors encodée dans le traducteur qui produit la RI. Ceci permet en particulier de développer des analyses pour plusieurs architectures à la fois. Ainsi, Lim *et al.* [89] ont construit un système qui permet de déduire, à partir d'une spécification de la sémantique opérationnelle d'un langage machine, une représentation intermédiaire dans un langage commun à différentes ar-

chitectures. Cette RI prend la forme de code C++ dont certains éléments sont paramétrables selon l'analyse qui sera ensuite lancée. Dans le même ordre d'idée, Dullien *et al.* [55] ont créé le langage REIL pour analyser des programmes en langage machine sur différentes architectures. Leur langage contient 17 instructions, qui sont de la forme 3-adresses et ont chacune un unique effet sur le système. Similairement, l'outil d'analyse dynamique Valgrind [108] traduit le code machine en un langage impératif avec la propriété *Single Static Assignment (SSA)*, qui explicite tous les effets des instructions. Le composant qui se charge de cette traduction, nommé VEX, a été réutilisé dans le projet d'analyse de programmes exécutables BitBlaze [132], et dans son successeur BAP [21], afin de produire une autre RI, dont la sémantique opérationnelle est fournie. Finalement, l'outil CodeSurfer/x86 [6] construit un ensemble de RI d'un binaire sous forme de graphes, comme le graphe de dépendances système, le graphe de flot de contrôle, ou encore l'arbre syntaxique abstrait.

Malgré cette diversité de formes, la sémantique est toujours définie manuellement, ce qui, du fait de la variété du jeu d'instruction x86 et de son évolution constante, entraîne des restrictions dans le nombre d'instructions modélisées. Par exemple, le *Semantic Specification Language* de Cifuentes *et al.* [39] est à la base du décompilateur Boomerang [140], mais les instructions de type *Single Instruction Multiple Data (SIMD)* n'y sont toujours pas définies, alors qu'elles sont présentes dans de nombreuses applications multimédias. De la même façon, le projet BAP ne gère pas les instructions arithmétiques à virgule flottante [20]. Plus généralement, aucun des articles cités ne revendique une couverture complète du jeu d'instructions.

2.4 Conclusion

Dans ce chapitre, nous avons décrit l'environnement d'exécution x86, puis construit une machine abstraite qui nous a permis d'insister sur les particularités de l'architecture. Ainsi, le code auto modifiant, la complexité du jeu d'instructions x86 et l'absence d'informations de haut niveau rendent l'analyse de programmes machine particulièrement difficile. En particulier, nous avons vu que le désassemblage d'un programme est un problème indécidable, et qu'il n'existe pas de représentation complète de la sémantique du langage machine x86.

CHAPITRE 3

ANALYSE DYNAMIQUE

L'analyse dynamique étudie une exécution concrète d'un programme, au contraire de l'analyse statique. Cette méthode présente l'avantage de gérer en grande partie les spécificités de l'architecture x86 présentées au chapitre précédent, notamment le code auto modifiant.
Nous commencerons par justifier en détail notre choix de l'analyse dynamique pour étudier les protections de logiciels malveillants, pour ensuite définir formellement cette méthode, et décrire son implémentation. Finalement, nous présenterons quelques travaux de recherche basés sur cette approche.

3.1 Analyse de programmes

L'analyse de programmes est un processus visant à répondre à une question précise sur le comportement d'un programme. L'analyse *dynamique*, que nous utiliserons ici, se définie par opposition à l'analyse dite *statique*. Nous allons introduire ces deux types d'analyse, pour ensuite justifier notre choix de la méthode dynamique pour étudier les protections de logiciels malveillants.

3.1.1 Analyse statique

L'analyse statique de programmes est un champ de recherche dont le but est, selon Nielson *et al.* [109], « *to offer techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer* ». Autrement dit, il s'agit de raisonner sur un programme *sans l'exécuter*. L'analyse statique est habituellement construite de façon *sûre*, c'est-à-dire que ses résultats s'étendent à toutes les exécutions possibles du programme (pas de faux négatifs), ce qui se fait souvent au détriment de la *précision*, l'absence de faux positifs. Pour ce faire, elle fait preuve de *conservatisme*, c'est-à-dire qu'elle démontre des propriétés faibles, mais qui sont effectivement toujours vraies [58].

Par exemple, une question que l'on pourrait poser sur le programme P_1 de la Figure 3.1 serait « *Quelles sont les valeurs que le registre* `eax` *peut prendre à la fin de l'exécution ?* ». Une première réponse possible d'une analyse statique pourrait être « *l'ensemble des valeurs sur* $\mathbb{B}^{32}$ », qui contient effectivement la bonne valeur (pas de faux négatifs), mais aussi beaucoup

d'autres que le registre ne pourra en fait pas prendre (des faux positifs). Une analyse statique plus précise pourrait remarquer que `eax` est initialisé à `0x13`, que les opérations suivantes ne font qu'augmenter sa valeur, et donc répondre « *l'ensemble des valeurs supérieures à* `0x13` » (en supposant qu'il n'y ait pas de débordement arithmétique). Finalement, une analyse statique précise pourrait répondre « *la valeur* `0x18` », qui est bien la seule possible, puisque ce programme n'utilisant pas d'entrées il ne peut donner lieu qu'à une seule exécution. Pour atteindre ce niveau de précision, l'analyse statique doit être dotée d'une description complète de la sémantique des instructions. Autrement dit, la précision de l'analyse statique dépend directement de sa connaissance du langage étudié.

```
DATA:

CODE:
mov eax, 0x13
add eax, 0x2
add eax, 0x3
halt
```

Figure 3.1 Programme P_1

3.1.2 Analyse dynamique

L'analyse dynamique étudie de son côté une (ou plusieurs) exécution concrète, pour laquelle elle connaît donc l'état exact de la machine à chaque étape. De ce fait, elle est habituellement *précise*, mais *non sûre*, car son résultat ne se généralise pas forcément aux exécutions non observées du programme, qui pourraient différer du fait d'autres valeurs d'entrées. Remarquons qu'une analyse dynamique n'est pas intrinsèquement non sûre, par exemple le programme P_1 peut être analysé dynamiquement de façon sûre, puisqu'il ne peut donner lieu qu'à une seule exécution. Une analyse dynamique peut étudier plusieurs exécutions, et non pas une seule, dans le but d'augmenter la sûreté de ses résultats [107]. De plus, certaines analyses peuvent avoir lieu pendant l'exécution, et non après, par exemple pour interdire des comportements au moment où ils se produisent [11]. Dans cette thèse, nous entendons par analyse dynamique *l'étude a posteriori d'une exécution d'un programme informatique.*

Par exemple, une exécution du programme P_1 est présentée dans la Figure 3.2 avec les informations accessibles à l'analyse dynamique. Il est alors possible de répondre à la question « *Quelles sont les valeurs que le registre* `eax` *peut prendre à la fin de l'exécution ?* » de la façon suivante : « `eax` *peut prendre* ***au moins*** *la valeur* `0x18` », car c'est celle observée à la

fin de cette exécution.

Temps	Instruction à exécuter	État des registres
0	`mov eax, 0x13`	$\Delta[r_i] = 0, \forall r_i \in REG$
1	`add eax, 0x2`	$\Delta[r_i] = \begin{cases} \texttt{0x13} & \text{si } r_i = \texttt{eax} \\ \texttt{0} & \text{sinon} \end{cases}$
2	`add eax, 0x3`	$\Delta[r_i] = \begin{cases} \texttt{0x15} & \text{si } r_i = \texttt{eax} \\ \texttt{0} & \text{sinon} \end{cases}$
3	`halt`	$\Delta[r_i] = \begin{cases} \texttt{0x18} & \text{si } r_i = \texttt{eax} \\ \texttt{0} & \text{sinon} \end{cases}$

Figure 3.2 Exécution du programme P_1

Pour résumer, ce qui fait la différence entre l'analyse statique et l'analyse dynamique – au sens où nous l'avons définie ici – est la *nature* de l'objet étudié : d'un côté un programme, c'est-à-dire un ensemble d'exécutions possibles, et de l'autre une exécution, pour laquelle l'état exact de la machine est connu.

3.1.3 Pourquoi le choix de l'analyse dynamique ?

Les trois défis de l'analyse du langage machine x86 mentionnés au chapitre 2 – code auto modifiant, complexité du jeu d'instructions et absence d'informations de haut niveau – sont des obstacles importants à une analyse statique précise. De ce fait, les outils statiques [6, 21, 132] font habituellement des hypothèses sur les programmes. En particulier, ils supposent que ceux-ci ont été produits par un compilateur standard, ce qui leur permet de réduire les difficultés, par exemple en reconnaissant les constructions usuelles du compilateur – comme les fonctions – ou en ne traitant qu'un sous-ensemble du langage machine. Mais les protections de logiciels malveillants sont construites spécifiquement comme des programmes « hors-normes » ; on y trouve beaucoup de code auto modifiant, peu de structures, et des instructions x86 exotiques – c'est-à-dire peu communes dans les applications standards –, comme nous le verrons au chapitre 4. Ainsi, l'analyse statique est inadaptée dans l'état actuel des choses pour étudier des protections de logiciels malveillants.

De son côté, l'analyse dynamique gère de façon transparente le code auto modifiant, et ne nécessite pas une sémantique des instructions x86 (bien qu'il soit utile d'en posséder une version allégée, comme nous le verrons). Sa principale difficulté réside habituellement dans le choix

d'entrées pertinentes pour le programme, ce que Ernst décrit comme étant « *the chief challenge of performing a good dynamic analysis* » [58]. Autrement dit, une analyse dynamique de qualité nécessite des entrées qui mèneront le programme à un chemin d'exécution intéressant.

Dans le cas des logiciels malveillants « grand public », dont l'objectif est d'infecter un maximum d'utilisateurs, les programmes ne prennent pas d'entrées au sens traditionnel du terme, c'est-à-dire que l'utilisateur n'a pas à donner des valeurs pour diriger l'exécution. Par contre, l'environnement d'exécution fournit des données aux programmes, par exemple par le biais de son générateur pseudo-aléatoire, ou bien encore par les fonctions de bibliothèques standards. Ces valeurs qui viennent de l'environnement d'exécution constituent alors les entrées du programme, et celui-ci peut refuser de s'exécuter s'il n'est pas satisfait avec ces valeurs. Dans le contexte de nos logiciels malveillants non ciblés, il est néanmoins facile de construire un environnement d'exécution adéquat, car comme ces programmes veulent infecter un maximum d'utilisateurs il suffit de reproduire l'environnement le plus standard possible. La construction d'un tel environnement est facilitée par le faible nombre de configurations différentes qui dominent le marché des ordinateurs personnels (typiquement le système d'exploitation Windows). Le choix des entrées du programme, c'est-à-dire de son environnement d'exécution, n'est donc pas un obstacle dans le cadre de l'analyse de ces logiciels malveillants. Par un raisonnement similaire, la sortie du programme est constituée de l'ensemble des modifications apportées à l'environnement d'exécution.

Pour ces différentes raisons, nous avons fait le choix de nous tourner vers l'analyse dynamique, que nous allons maintenant définir en détail.

3.2 Processus d'analyse dynamique

L'analyse dynamique développée dans cette thèse est *l'étude a posteriori d'une exécution d'un programme informatique*. Le processus qui nous permet de réaliser cette étude est décrit en Figure 3.3. Tel qu'expliqué précédemment, les entrées du programme correspondent à l'environnement d'exécution, et cet environnement est simplement celui d'un utilisateur standard.

Le traceur et la trace sont communs aux chapitres 4 et 5, et nous les décrirons ici. L'étape d'analyse en elle-même est spécifique à chacun de ces chapitres.

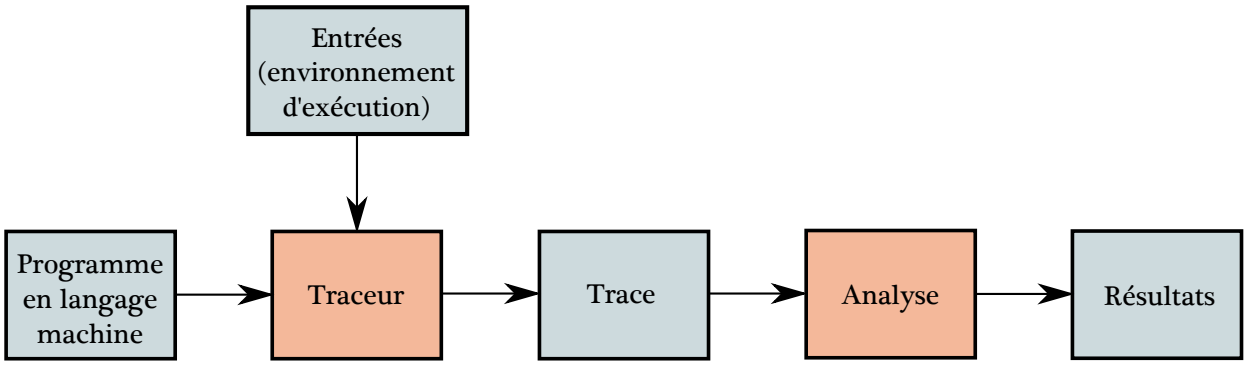


Figure 3.3 Processus d'analyse dynamique

3.2.1 Traceur

Un traceur est un outil qui permet d'observer l'exécution d'un programme. Pour réaliser sa fonction, il doit posséder au minimum les habiletés suivantes :

1. **La connaissance de l'état de la machine, à la granularité temporelle d'une instruction.** En d'autres termes, un traceur peut examiner l'état de la machine entre chaque instruction exécutée. En particulier, nous considérons que les outils qui ne peuvent observer l'état de la machine qu'à des moments particuliers (comme l'appel de fonctions de bibliothèques) ne sont pas des traceurs.
2. **La transparence de l'observation.** Cette notion est particulièrement importante pour l'analyse de logiciels malveillants, car ces derniers vérifient souvent qu'ils s'exécutent sans être observés. Dinaburg *et al.* définissent la transparence comme le fait d'être indétectable *par n'importe qu'elle méthode de détection* [53], ce que les auteurs reconnaissent comme impossible à satisfaire en pratique. Nous préférerons donc parler de *transparence relative* : pour toute méthode de détection D d'un traceur, celui-ci doit pouvoir être modifié de façon à devenir indétectable par D. Autrement dit, la présence du traceur ne doit pas perturber *inconditionnellement* l'exécution d'un programme.
3. **La capacité d'enregistrer des informations.** Un traceur doit pouvoir noter les données issues de son observation, dans ce qu'on appellera une trace d'exécution.

Il serait évidemment sous-optimal de faire enregistrer au traceur l'état complet de la machine à chaque étape de l'exécution, puisqu'une grande partie de cet état reste inchangée suite à l'exécution d'une instruction. Par conséquent, il est utile pour le traceur de posséder également **un critère de sélection des informations à enregistrer selon l'état de la machine**. Dans notre cas, nous collectons pour chaque état de la machine :

– L'adresse et le code machine de la prochaine instruction à exécuter.

– Toutes les opérandes d'entrées de la prochaine instruction à exécuter, c'est-à-dire *les registres ou les adresses mémoires qu'elle va lire*, ainsi que les valeurs qui seront lues.

– Toutes les opérandes de sortie de la prochaine instruction à exécuter, c'est-à-dire tous *les registres ou les adresses mémoires qu'elle va écrire*, ainsi que les valeurs qui seront écrites.

Pour préciser cela, l'Algorithme 1 présente le pseudo-code du traceur d'un programme $P \in RASM$ sur la machine abstraite du chapitre 2. Le mot-clé **Enregistrer** indique la collecte par le traceur d'une information. Nous définissons $EXE : MSTATE \to MSTATE$ la fonction qui fait exécuter sa prochaine instruction à la machine.

La fonction $ENTREES : RASM_{/Ins} \times MSTATE \to (REG \cup \mathbb{B}^{32})^*$ retourne l'ensemble des opérandes d'entrées d'une instruction, tandis que *SORTIES* retourne l'ensemble de ses opérandes de sorties. Remarquons que ces fonctions ont besoin de connaître l'état de la machine, car certaines opérandes en dépendent. Par exemple, posons $M \in MSTATE$ tel que `eax` a la valeur `0x107` dans M, et `ebx` a la valeur `0x0`, alors *ENTREES(`mov [eax], ebx`, M)*={`eax`, `ebx`} et *SORTIES(`mov [eax], ebx`, M)*={`0x107`}. Ainsi, l'adresse mémoire écrite ne peut être déterminée qu'en fonction de l'état de la machine.

Algorithme 1 Traceur du programme P

```
1: S = 𝒞(P)
2: tant que S ≠ ⊥ faire
3:     I = ProchaineInstruction(S)
4:     Enregistrer adresse et code machine de I
5:     Enregistrer tous les e ∈ ENTREES(I, S)
6:     Enregistrer la valeur de tous les e ∈ ENTREES(I, S)
7:     Enregistrer tous les s ∈ SORTIES(I, S)
8:     S = EXE(S)
9:     Enregistrer la valeur de tous les s ∈ SORTIES(I, S)
```

Remarquons que la lecture de la valeur d'une opérande de sortie d'une instruction ne peut se faire qu'après l'avoir exécutée. Il nous faut insister sur le fait que les fonctions *ENTREES* et *SORTIES* ne sont pas propres à l'analyse dynamique elle-même, car elles utilisent en fait une

partie de la sémantique de l'instruction. Pour expliciter cela, nous catégorisons la sémantique d'une instruction machine en trois niveaux de précision présentés en Figure 3.4.

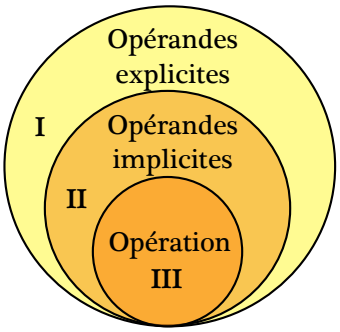


Figure 3.4 Niveaux de précision de la sémantique du langage machine x86

Une sémantique de niveau 1 correspond à la donnée des opérandes mentionnées dans l'instruction. Pour le niveau 2, il faut avoir en plus une connaissance des opérandes non mentionnées comme telles dans l'instruction. Finalement, le niveau 3 contient les connaissances précédentes plus une définition de l'opération réalisée par l'instruction. Par exemple, la sémantique de l'instruction x86 `push ecx` peut être catégorisée de la façon suivante :

- **Niveau 1** : le registre `ecx` est une entrée explicite.
- **Niveau 2** : le registre de pile `esp` est une entrée implicite, et l'adresse mémoire pointée par `esp - 4` ainsi que `esp` lui-même sont des sorties implicites. Remarquons que la pile croit vers les adresses décroissantes dans l'architecture x86.
- **Niveau 3** : l'instruction décrémente `esp` de 4 octets, puis écrit à l'adresse mémoire pointée par `esp` la valeur qui est dans `ecx`.

C'est au niveau 3 qu'évoluent habituellement les travaux sur la sémantique d'un langage, comme la sémantique opérationnelle du langage *RASM* de la Figure 2.3, et les articles que nous avons vus au chapitre précédent. Notre traceur possède de son côté, par le biais des fonctions *ENTREES* et *SORTIES*, une sémantique de niveau 2. Or l'obtention de celle-ci est plutôt aisée en pratique, même pour le langage machine x86, comme nous le verrons. Nous évitons de ce fait le problème de la complétude de la sémantique de niveau 3 décrit au chapitre précédent.

3.2.2 Trace d'exécution

Une trace d'exécution est la représentation de l'exécution d'un programme, c'est-à-dire essentiellement la concaténation des informations émises par le traceur. Nous en donnons ici

une définition formelle qui servira aux analyses des chapitres suivants, en commençant par décrire l'unité de base d'une trace.

Definition 2. *Une instruction dynamique D_i est un n-uplet composé de :*
- *une adresse mémoire $\mathcal{A}[D_i]$,*
- *une instruction machine $\mathcal{I}[D_i]$ exécutée à $\mathcal{A}[D_i]$,*
- *deux ensembles d'adresses mémoires lues et écrites $\mathcal{R}_M[D_i]$ et $\mathcal{W}_M[D_i]$ par $\mathcal{I}[D_i]$,*
- *deux ensembles de registres lus et écrits $\mathcal{R}_R[D_i]$ et $\mathcal{W}_R[D_i]$ par $\mathcal{I}[D_i]$.*

Les opérandes d'entrées de l'instruction sont donc l'ensemble $\mathcal{R}_M[D_i] \cup \mathcal{R}_R[D_i]$, tandis que les opérandes de sortie sont $\mathcal{W}_M[D_i] \cup \mathcal{W}_R[D_i]$. De plus, pour toute instruction dynamique D_i, nous nous donnons deux opérateurs $(\mathcal{V}_E[D_i], \mathcal{V}_S[D_i]) \in ((REG \cup \mathbb{B}^{32}) \to \mathbb{B}^{32})^2$ qui retournent respectivement la valeur associée à une opérande d'entrée, et celle associée à une opérande de sortie (c'est-à-dire la valeur produite par l'instruction pour cette opérande).

Definition 3. *Une trace d'exécution est une séquence finie d'instructions dynamiques D_i, avec les opérateurs de valuation $\mathcal{V}_E[D_i]$ et $\mathcal{V}_S[D_i]$.*

Par exemple la trace d'exécution du programme P_1 précédemment décrit est donnée en Figure 3.1. Par souci de compréhension, nous affichons la traduction assembleur de $\mathcal{I}[D_i]$.

Tableau 3.1 Trace d'exécution du programme P_1

D_i							
$\mathcal{A}[D_i]$	$\mathcal{I}[D_i]$ [1]	$\mathcal{R}_M[D_i]$	$\mathcal{R}_R[D_i]$	$\mathcal{W}_M[D_i]$	$\mathcal{W}_R[D_i]$	$\mathcal{V}_E[D_i]$	$\mathcal{V}_S[D_i]$
0x100	mov eax, 0x13				eax		eax=0x13
0x105	add eax, 0x2		eax		eax	eax=0x13	eax=0x15
0x108	add eax, 0x3		eax		eax	eax=0x15	eax=0x18
0x10D	halt						

Finalement, nous noterons $TRACE$ l'ensemble des traces d'exécution, et nous définissons $T_{/Ins}$ comme la séquence des instructions machine d'une trace $T = D_1; \ldots; D_n$, c'est-à-dire $T_{/Ins} = I_1; \ldots; I_n$ si $\forall k \in [1, n],\ \mathcal{I}[D_k] = I_k$.

1. Sous sa forme assembleur

3.3 Implémentation

3.3.1 Possibilités

Il existe différentes manières d'implémenter un traceur, comme :

– *L'instrumentation* : du code d'analyse est inséré dans le programme à tracer pour collecter des informations lors de son exécution. La majorité des outils effectuent cette insertion dynamiquement, c'est-à-dire qu'ils recompilent au moment de l'exécution le code du programme original en y ajoutant le code d'analyse. Parmi les outils les plus matures sous Windows, on peut citer DynInst [22], DynamoRIO [19] et Pin [95].

– *L'émulation* : l'environnement d'exécution est simulé, le plus souvent pour être exécuté sur une autre architecture matérielle. Cela permet de suivre précisément l'exécution et donc de construire un traceur. Par exemple, l'émulateur de système QEMU [12] a servi de base à de nombreux traceurs dans la communauté académique, comme l'outil TEMU du projet BitBlaze [132], ou encore l'environnement d'exécution pour logiciels malveillants Anubis [10].

– *Le débogage* : un débogueur est un outil qui permet à l'analyste de suivre pas à pas l'exécution d'un programme en posant des points d'arrêts. La majorité des débogueurs peuvent produire automatiquement une trace exécution, habituellement en utilisant un drapeau du processeur x86 nommé *Trap Flag* qui leur permet de prendre la main après chaque instruction exécutée.

Remarque 7. Comme nous le verrons, la séparation entre ces trois méthodes n'est pas aussi stricte en pratique : de nombreux outils vont en fait les combiner entre elles.

3.3.2 Notre choix

Nous avons choisi de réaliser l'implémentation du traceur de l'Algorithme 1 avec Pin [95], qui permet l'instrumentation dynamique de code. Cet outil fourni de nombreuses fonctions qui permettent à la fois d'abstraire les idiosyncrasies du jeu d'instruction x86, et de manipuler aisément l'état du programme avant et après chaque instruction (le requis 1 d'un traceur). D'autre part, cet outil fait preuve d'une grande robustesse (support des programmes multi *thread*, du code auto modifiant...), ce qui est particulièrement appréciable quand on veut analyser des protections de logiciels malveillants. De plus, Pin utilise la bibliothèque XED [79] qui fournit la sémantique de niveau 2 précédemment mentionnée, c'est-à-dire qu'elle définit

les opérandes, implicites ou explicites, de toutes les instructions machine x86. Finalement, cet outil permet aisément d'enregistrer l'information collectée par le biais des fonctions de bibliothèque standard (requis 3 d'un traceur).

Concernant la transparence relative (requis 2 d'un traceur), il faut remarquer que Pin a été conçu avant tout pour étudier des architectures machines, pas des logiciels malveillants. Par exemple, Intel s'en sert pour simuler de nouvelles instructions, avant de les introduire physiquement dans les processeurs [78]. De ce fait, il existe un certain nombre de signes évidents qui permettent à un programme de se savoir instrumenté par Pin, comme ceux décrits par Falcon *et al.* [60]. Malgré cela, la flexibilité de cet outil nous a permis de passer à travers toutes les techniques de protection rencontrées au cours de cette thèse, c'est-à-dire d'atteindre la *transparence relative* définie en §3.2.1.

Par exemple, Pin ne gère pas par défaut le fait qu'un programme manipule le *Trap Flag* du processeur x86, car il l'utilise lui même – c'est-à-dire que Pin emploie une fonctionnalité de débogage (cf. Remarque 7) –, or c'est une technique rencontrée dans certains logiciels malveillants. Nous avons donc instrumenté de façon particulière les instructions machine concernées pour les simuler sans toucher au drapeau, c'est-à-dire que nous avons fait de l'émulation. Ainsi, le code de notre traceur est constitué de plus de 600 lignes de code C++, et est disponible en [29].

Remarque 8. Pin ne permet pas l'instrumentation du code situé dans le noyau du système d'exploitation. Concrètement, cela implique que tous les logiciels malveillants que nous étudions dans cette thèse ne chargent pas de modules noyaux en mémoire.

3.4 Revue de littérature

Nous présentons ici les travaux sur l'analyse dynamique de programmes en langage machine, que ceux-ci soient des logiciels malveillants *ou pas*. En effet, il existe une motivation pour l'analyse de programmes exécutables en dehors de l'étude des logiciels malveillants, qui peut se résumer sous l'acronyme « *WYSINWYX (What You See Is Not What You Execute)* », donné par Balakrishnan *et al.* [7]. En d'autres termes, l'étude d'un programme en langage machine permet d'analyser ce qui est *vraiment* exécuté, au contraire de l'analyse du code source, à cause des transformations que le compilateur effectue.

3.4.1 Applications de l'analyse dynamique

Reconstruction de structures de données. Le problème est de retrouver l'organisation des structures (les différents champs, les imbrications) dans un programme exécutable et, dans certains cas, de découvrir également la sémantique des champs. Lin *et al.* [91] ont construit un outil qui déduit la sémantique d'une case mémoire par son implication dans une opération qui la « révèle », par exemple comme un argument d'un appel système documenté. Cette information est ensuite propagée en avant et en arrière sur la trace d'exécution. Les auteurs rajoutent une série d'heuristique pour regrouper les cases mémoires dans une même structure de données. Plus récemment, Slowinska *et al.* [129] ont étendu ce travail en s'intéressant à la reconstruction de structures internes d'un programme par l'observation des accès réalisés sur ces structures lors d'une exécution. L'intuition de base est que si A est un pointeur, un accès mémoire à l'adresse $A+4$ est un indice de la présence d'un champ de 4 octets à l'adresse A.

Protocoles réseau. Comprendre des protocoles réseau non documentés, tels que ceux utilisés par certaines familles de logiciels malveillants [34], est une tâche difficile, qui nécessite de retrouver à la fois le format des messages et la machine à état du protocole. Caballero *et al.* [26] déduisent le format des messages de la façon dont les données réseau sont manipulées lors d'une exécution. Parmi leurs intuitions il y a, par exemple, le fait qu'un champ A d'un message qui décrit la longueur d'un champ B de longueur variable va toujours être utilisé pour accéder au champ *après* B, ou bien encore qu'un champ de longueur fixe et inférieure à 4 octets sera accédé par une seule instruction. Ils ont appliqué leur technique sur des protocoles standards (HTTP, DNS, etc.) puis sur un protocole « maison » d'un logiciel malveillant [25]. Lin *et al.* [90] ont étendu ce travail en prenant en compte la pile d'appels de fonctions, ce qui leur permet en particulier de reconstruire les relations hiérarchiques entre les champs. Finalement, Comparetti *et al.* [46] se sont intéressés à la reconstruction de la machine à état d'un protocole. Pour ce faire, suite à l'observation d'un ensemble de dialogues réseau, ils regroupent les messages similaires, puis construisent une première machine à état qui les accepte, pour en déduire la machine minimale qui fait de même. La construction générale d'une telle machine est difficile lorsqu'on ne connaît que des chemins acceptants, alors pour rendre le problème attaquable ils introduisent des heuristiques, comme le fait que certains messages sont des préfixes obligatoires à d'autres, ou qu'il existe des messages pour indiquer la fin d'un échange.

Extraction d'algorithmes. L'objectif est ici de permettre à l'analyste d'exécuter avec des arguments de son choix des algorithmes précis d'un logiciel malveillant, comme la génération de noms de domaines à contacter, ou le déchiffrement du trafic réseau. Ainsi, dans [86] les

auteurs ont construit un outil qui *(i)* à partir d'un point d'intérêt dans une trace d'exécution, extrait le code et les données qui y sont reliés, *(ii)* construit un programme indépendant, et *(iii)* permet le rejeu de ce nouveau programme. Comme ils n'observent pas forcément tout le code nécessaire à un rejeu du programme pour toutes les entrées possibles, ils transforment les branchements conditionnels dont une seule des deux branches a été prise, en branchements inconditionnels. Caballero *et al.* [24] suivent une approche similaire, à ceci près qu'ils utilisent un algorithme de désassemblage combinant analyse statique et dynamique pour l'extraction. Une difficulté que ces travaux rencontrent est de faire correspondre au niveau machine les concepts des langages de programmation de haut niveau, comme les notions de fonctions et de paramètres. Finalement, Dolan-Gavitt *et al.* [54] ont présenté un algorithme pour reconstruire un programme à partir de plusieurs traces d'exécution, dans le cadre de leur travail sur l'introspection de machines virtuelles.

Ainsi l'analyse dynamique est une voie privilégiée pour attaquer de nombreux problèmes sur les programmes en langage machine x86, grâce à sa robustesse naturelle contre les spécificités de l'architecture. Il nous faut tout de même remarquer qu'il est très rare de trouver des réflexions sur la *nature* de l'analyse dynamique, ou même une simple définition d'une trace d'exécution. Une notable exception est le travail de Schwartz *et al.* [125] dans lequel les auteurs définissent formellement le *taint analysis* (suivie de valeurs) dans une trace d'exécution, à partir de la sémantique opérationnelle d'un langage intermédiaire proche du langage machine. Cette analyse dynamique particulière est à la base de la majorité des travaux cités ici.

3.4.2 Couverture de code

L'analyse dynamique ne raisonne que sur les exécutions qu'elle peut observer, ce qui a poussé à rechercher des moyens d'augmenter sa couverture du programme, c'est-à-dire d'exhiber un maximum d'exécutions différentes.

Une des méthodes utilisées est *l'exécution concolique*, qui combine l'analyse dynamique à l'exécution symbolique, afin de découvrir de nouveaux chemins d'exécutions. En particulier cette analyse comprend les étapes suivantes : *(i)* exécuter concrètement le programme sur une première entrée tout en récoltant les contraintes qui se sont exercées par le biais des branchements conditionnels, *(ii)* transformer ces contraintes en formules logiques, et *(iii)* trouver les entrées qui permettent d'obtenir l'inverse d'une contrainte, et donc de prendre un nouveau chemin d'exécution. Parmi les projets qui utilisent ce type d'analyse sur des programmes en langage machine, citons SAGE de Microsoft [67] et KLEE développé par Cadar *et al.* [27].

Dans le cadre des logiciels malveillants, Moser *et al.* [107] ont construit un outil d'exploration de différents chemins d'exécution. Pour ce faire, ils suivent l'utilisation des entrées du programme pour déterminer quels branchements en dépendent, prennent une copie de l'état de la machine au moment de chacun de ces branchements pour, une fois l'exécution terminée, remettre la machine dans l'état au moment du branchement tout en prenant soin de modifier la valeur de l'entrée concernée pour forcer l'exécution de l'autre chemin. La difficulté principale est que la modification de l'entrée doit être faite de façon consistante, c'est-à-dire en s'assurant que toutes les valeurs qui dépendent de cette entrée sont aussi modifiées en conséquence. De leur côté, Comparetti *et al.* [45] extraient d'une trace d'exécution des modèles de comportements, sous la forme de graphes de flot de contrôle enrichis, pour les rechercher dans le code non exécuté dans d'autres logiciels malveillants. Ainsi, ils trouvent des fonctionnalités non exécutées dans ces derniers.

Il existe donc des moyens d'étendre la sûreté d'une analyse dynamique, et celle-ci ne devrait par conséquent pas être vu comme un facteur intrinsèquement limitant, en particulier dans des applications où le nombre de chemins d'exécution différents est habituellement faible, comme pour les protections de logiciels malveillants que nous verrons au chapitre suivant.

3.5 Conclusion

Dans ce chapitre nous avons défini l'analyse dynamique, après avoir justifié son choix pour étudier les logiciels malveillants. Nous avons notamment décrit précisément ce que sont un traceur et une trace d'exécution. Ensuite, nous avons introduit l'implémentation d'un traceur qui va servir aux projets des deux chapitres suivants. Finalement, nous avons fait une revue des applications de l'analyse dynamique, ainsi que des travaux sur la couverture de code.

CHAPITRE 4

CARTOGRAPHIE DES PROTECTIONS

De nombreuses techniques ont été développées pour protéger les programmes informatiques, par exemple pour cacher la propriété intellectuelle qu'ils contiennent. Les logiciels malveillants emploient aussi des protections, mais dans le but d'éviter la détection par les antivirus, ou de complexifier le travail des analystes en sécurité. Néanmoins, les protections de logiciels malveillants restent peu étudiées, la majorité des travaux se concentrant plutôt sur leur charge utile. C'est pour remédier à cela que nous présentons dans ce chapitre les résultats d'une étude des protections de logiciels malveillants.
Nous définirons dans un premier temps la notion de protection, pour ensuite évoquer les travaux existants, et finalement présenter notre approche et ses résultats.

4.1 Fondations

Un protecteur implémente une *transformation de programme* qui produit un programme protégé à partir d'un programme non protégé. Dans le cadre des logiciels malveillants, l'analyste n'a pas nécessairement accès au protecteur lui-même, et encore moins au programme original non protégé. Ainsi, l'objet de l'analyse est seulement le *résultat* du protecteur, et il est alors difficile de discuter en détail la qualité de l'obfuscation et les techniques utilisées (tel que cela est fait pour la protection d'applications légitimes [42]). Nous allons définir ici un cadre simple et semi-formel pour englober l'ensemble des techniques de protection.

La notion de protection doit être abordée à partir de l'attaquant contre lequel elle protège. Les logiciels malveillants font face à divers adversaires, comme les antivirus, ou les analystes humains. Nous regroupons alors ceux-ci dans le modèle suivant.

Definition informelle 1. *Un attaquant A d'un programme P est un triplet $(D_A, C_A, O_{A,P})$, avec :*
- *D_A : son environnement d'exécution d'analyse,*
- *C_A : sa capacité de raisonnement,*
- *$O_{A,P}$: son objectif à atteindre en rapport avec P.*

Ainsi, un attaquant A d'un programme P étudie le programme dans un environnement D_A en utilisant sa capacité de raisonnement C_A afin d'atteindre $O_{A,P}$. Par exemple, si P est un logiciel malveillant, alors :

– Un analyste humain H peut être un attaquant de P, avec D_H son environnement d'exécution contenant ses outils d'analyse, C_H sa capacité cognitive, et $O_{H,P}$ la compréhension de P.
– Un antivirus AV peut être un attaquant de P, avec D_{AV} son émulateur, c'est-à-dire son environnement d'exécution factice, C_{AV} son algorithme de détection, et $O_{AV,P}$ l'identification de P comme malveillant.
– Une *sandbox* S peut être un attaquant de P, avec D_S l'environnement d'exécution qu'elle reproduit, C_S ses algorithmes de collecte d'informations, et $O_{S,P}$ la production d'un rapport correct sur le comportement de P.

Dans ce chapitre, nous laissons de côté les programmes *RASM* et notre machine abstraite, pour revenir dans le monde des programmes en langage machine x86. Nous notons alors $\mathcal{P}_{X86}$ l'ensemble de ces programmes, et $[\![P]\!]$ la fonction calculée par un programme P. Comme expliqué en §3.1.3, les entrées d'un programme correspondent dans notre contexte à son environnement d'exécution, tandis que ses sorties sont les modifications qu'il apporte à cet environnement d'exécution.

Definition informelle 2. *Pour tout $P \in \mathcal{P}_{X86}$, la fonction $\square : \mathcal{P}_{X86} \to \mathcal{P}_{X86}$ protège contre l'attaquant $A = (D_A, C_A, O_{A,P})$ si :*

1. *L'effet de P sur le système est partiellement conservé par $\square$:*
$$\forall x \neq D_A, \quad [\![P]\!](x) = [\![\square(P)]\!](x)$$
2. *Il est plus difficile pour A d'atteindre $O_{A,\square(P)}$ que $O_{A,P}$.*

Un protecteur contre un attaquant A est alors un programme implémentant une fonction qui protège contre A. Commençons par expliquer pourquoi la deuxième partie de la Définition informelle 2 reste générale : étant donné la multitude des techniques de protection possibles – traduction dans un nouveau langage qui sera interprété à l'exécution, obfuscation, tests contre l'environnement, etc. –, il est délicat de définir la notion de difficulté de façon à la fois précise et générique. Plus de précision n'étant pas nécessaire à notre raisonnement, cette définition générale est suffisante.

Deuxièmement, il est important d'insister sur le fait qu'à la différence des transformations de programmes telles que la compilation ou la spécialisation [82], la protection peut ne pas conserver *complètement* la sémantique du programme qu'elle transforme (définie ici comme la modification du système). En particulier, le programme protégé peut refuser de s'exécuter s'il détecte la présence de l'environnement d'exécution de l'attaquant.

Par exemple, lors de notre analyse de la famille *Swizzor* [33], nous avons observé que des tests ciblés contre des émulateurs antivirus étaient effectués. Si l'environnement d'exécution se comportait comme un de ces émulateurs ($x = D_A$, où A est l'antivirus), alors le programme terminait abruptement, sans exécuter sa charge utile. En cas d'exécution dans un environnement classique ($x \neq D_A$), la charge utile était bien exécutée.

Finalement, nous allons mettre en avant une caractéristique courante des fonctions de protection : celles-ci peuvent utiliser une entrée supplémentaire afin de renvoyer une sortie différente pour un même programme d'entrée. Pour ce faire, nous notons l'ensemble des valeurs d'entrées possibles $INPUT$. En pratique, cette entrée peut par exemple être donnée par un générateur pseudo-aléatoire, ou bien être fournie par l'utilisateur de la protection.

Definition informelle 3. *Pour tout $P \in \mathcal{P}_{X86}$, la fonction $\triangle : \mathcal{P}_{X86} \times INPUT \to \mathcal{P}_{X86}$ protège* de façon versatile *contre l'attaquant $A = (D_A, C_A, O_{A,P})$ si :*

1. *Sa restriction à son premier argument protège contre A, conformément à la Définition informelle 2,*
2. *$\forall e \in INPUT, \forall e' \in INPUT$, si $e \neq e'$ alors $\triangle(P, e)$ et $\triangle(P, e')$ diffèrent syntaxiquement.*

Un protecteur versatile contre un attaquant A est alors un programme implémentant une fonction qui protège de façon versatile contre A. Cette caractéristique permet aux protecteurs de produire des fichiers différents pour une même logique malveillante initiale. Ceci explique en partie le nombre astronomique de fichiers reconnus malveillants sur Internet.

Par exemple, lors de notre analyse de la famille *Waledac* [32], nous avons observé la publication d'un nouvel exemplaire toutes les deux heures, et ce de façon continue pendant plusieurs jours. La charge utile de ces exemplaires (les interactions avec le système) restait exactement la même. Ainsi, la production automatique de fichiers différents permettait à la famille d'avoir un temps d'avance sur la détection des antivirus.

Remarque 9. La versatilité peut sembler similaire à la notion de « polymorphisme » (ou *client-side polymorphism*), qui désigne des logiciels malveillants se propageant en créant des copies modifiées d'eux-mêmes. Dans ce cas, le protecteur est alors inclus dans le logiciel malveillant lui-même, et n'est pas un programme séparé, cette situation étant quant à elle nommée *server-side polymorphism*. C'est pour éviter la confusion et regrouper les deux notions sous un même terme que nous employons le mot « versatilité ».

4.2 Problème de recherche

4.2.1 Motivation

Comme nous l'avons montré dans les chapitres précédents, la complexité des programmes en langage machine a poussé au développement de méthodes d'analyse *spécifiques*. En particulier, de nombreux outils se focalisent sur des programmes produits par des compilateurs standards. Cela permet de faire de nombreuses hypothèses et d'obtenir alors des solutions acceptables à des problèmes indécidables, comme le désassemblage. Ainsi, ces outils n'analysent pas correctement *tous* les programmes en langage machine, mais seulement un sous-ensemble de ceux-ci, et c'est ce qui les rend performants.

De leur côté, les logiciels malveillants sont rarement la sortie directe d'un compilateur, car ils ont subi une transformation supplémentaire : la protection. Il est alors intéressant de déterminer s'il existe des caractéristiques communes dans les logiciels malveillants protégés, afin de développer des méthodes d'analyse *spécifiques* et performantes. Autrement dit, bien que la protection d'un logiciel malveillant puisse le transformer en un programme hors normes, afin de mettre en échec les outils habituels, n'existe-t-il pas en fait d'autres normes dans ces programmes protégés ?

4.2.2 Hypothèse

L'hypothèse de recherche que nous allons poser est *l'existence d'un modèle de protection particulièrement prévalent parmi les logiciels malveillants*. Notre intuition – que cette hypothèse est vraie – vient des analyses de logiciels malveillants réalisées au cours de la thèse, comme [33] et [34].

Afin de définir le modèle de protection en question, il nous faut expliciter le concept de *couche de code*. En plus des notations introduites au chapitre précédent, nous notons $\mathcal{ADDR}[D_i]$ l'ensemble des adresses mémoires occupées par une instruction dynamique D_i, ce qui contient en particulier $\mathcal{A}[D_i]$, l'adresse à laquelle elle commence. Pour la simplicité des explications, l'instruction dynamique D_i est à la position i dans sa trace d'exécution.

Definition 4. *Soit $T \in TRACE$, l'ensemble créateur de $D_i \in T$ est défini comme :*

$$Crea(D_i, T) = \{\ D_j \in T,\ j < i \mid \exists\ a \in \mathcal{ADDR}[D_i] \cap \mathcal{W}_M[D_j],\ \nexists\ k \in]\!]j; i[\![,\ a \in \mathcal{W}_M[D_k]\ \}$$

L'ensemble créateur d'une instruction contient donc les instructions dynamiques responsables de l'écriture en mémoire d'au moins un octet de celle-ci (qui n'a pas été réécrit depuis).

Remarquons qu'il s'agit bien d'un ensemble, puisqu'une instruction x86 comprend plusieurs octets qui peuvent avoir été écrits par différentes instructions.

Definition 5. *Soit $T \in TRACE$, la profondeur de $D_i \in T$ est définie comme :*

$$Prof(D_i, T) = \begin{cases} \max_{D_j}\big(Prof(D_j, T)\big)+1, \text{ pour } D_j \in Crea(D_i, T), & si\ Crea(D_i, T) \neq \varnothing \\ 0 & sinon \end{cases}$$

Une instruction dynamique qui était présente initialement dans le programme a donc une profondeur de 0, tandis qu'une instruction créée dynamiquement a une profondeur supérieure de 1 à la profondeur maximale des instructions qui l'ont écrite en mémoire.

Definition 6. *Soit $T \in TRACE$, la couche de code de T d'indice k est définie comme :*

$$Couche(k,T) = \{\ D_j \in T \mid Prof(D_j, T) = k\ \}$$

Ainsi, une trace d'exécution est divisée en couches de code qui regroupent les instructions d'une même profondeur. Nous pouvons alors énoncer notre hypothèse.

Hypothèse de recherche. *Le modèle de protection défini par les deux propriétés suivantes est commun pour les logiciels malveillants :*

1. *Il y a au moins deux couches de code auto modifiant dans le programme,*
2. *Le rôle des couches dépend de leur ordre de création :*
   - *Les premières couches, c'est-à-dire toutes sauf la dernière, servent à la protection de la charge utile,*
   - *La dernière couche implémente la charge utile du logiciel malveillant.*

La notion de charge utile est ici employée au sens large, c'est-à-dire comme l'ensemble des interactions entre le système et le logiciel malveillant, par exemple la manipulation de fichiers et les communications réseau. L'ordre de création des couches correspond aux indices de la Définition 6 : la couche d'indice 0 est la première, tandis que celle d'indice maximal est la dernière.

Le modèle de protection de notre hypothèse est donc basé sur le code auto modifiant, avec une dichotomie entre les premières couches, qui servent à la protection, et la dernière, qui réalise la charge utile du logiciel malveillant. D'autres hypothèses possibles auraient été par exemple un mélange charge utile et protection dans chaque couche, ou bien encore une protection basée sur une machine virtuelle et sans code auto modifiant. Le problème de recherche est alors de tester la validité de notre hypothèse.

4.3 Travaux existants

Nous allons présenter ici les travaux qui décrivent les protections utilisées par les logiciels malveillants, et qui pourraient nous donner des indications sur la validité de notre hypothèse.

4.3.1 Statistiques

De façon générale, l'industrie antivirus communique peu sur les protections de logiciels malveillants, probablement car leurs clients (et les médias) ont plus d'intérêt pour la charge utile. Néanmoins, quelques exceptions notables permettent de saisir l'évolution des protections au cours des dernières années :

– En 2006, plus de 92% des logiciels malveillants sont protégés, d'après Brosch *et al.* [18]. Dans ces programmes protégés, Morgenstern *et al.* [106] dénotent que pour plus de 70% des cas il s'agit de protecteurs du *domaine public*. Ces protections sont librement accessibles sur Internet (ou moyennant l'achat d'une licence, ce qui entraîne la disponibilité de versions pirates), et sont pour la plupart normalement destinées à la protection de la propriété intellectuelle. S'ajoute à cela des compresseurs comme UPX [113], qui ne sont pas à proprement parler des protecteurs, mais qui peuvent mettre en difficulté les antivirus par leur utilisation de code auto modifiant. De ce fait, il n'est pas possible de simplement détecter la protection elle-même, sous peine de faux positifs sur les applications légitimes qui les emploient. Cette nécessité d'une analyse en profondeur des protections se heurte alors à la complexité grandissante des protections commerciales, par exemple avec l'introduction de *machines virtuelles* [149].

– Plus récemment, les logiciels malveillants ont employé de plus en plus des protections *privées*, c'est-à-dire créées spécifiquement pour les logiciels malveillants. Ainsi, celles-ci représentent plus de 40% des cas en 2010 d'après Morgenstern *et al.* [106]. Ce changement de tactique – mentionné par d'autres sources industrielles [146, 23] – peut s'expliquer en partie par les accords établis entre les compagnies antivirus et celles qui commercialisent des protections, afin de détecter les licences utilisées par des logiciels malveillants [77].

Ces diverses statistiques montrent l'importance des protections de logiciels malveillants, et une tendance récente à privilégier des protections « maison ». Néanmoins, comme mentionné en §1.2.1, toutes ces études sont basées sur des outils de recherche *statique* de signatures, comme PEiD [130], SigBuster [85] ou Yara [154], et cela ne répond donc pas directement à notre problème de recherche :

– Les protections identifiées sont celles pour lesquelles une signature a été ajoutée, ce qui privilégie naturellement les protections connues et anciennes, tandis que les plus confidentielles ou les nouvelles versions sont invisibles.
– De par leur nature statique, ces outils ne peuvent reconnaître que la couche extérieure d'un programme ; si une partie du code de la protection n'est visible que lors de l'exécution, elle ne sera pas analysée. En particulier, si des protections sont combinées entre elles, seule la dernière appliquée risque d'être identifiée.
– Des faux positifs peuvent avoir lieu, et il est même facile de les créer délibérément, par exemple avec l'outil ExeForger [138].
– Le résultat donné par ces outils est le nom d'une protection, ce qui ne fournit pas d'indications sur le fonctionnement de celle-ci, et en particulier sur sa correspondance avec notre modèle.

4.3.2 Études en profondeur

Dans un autre type de travail industriel, on trouve des descriptions détaillées de protections, habituellement dans le cadre d'une analyse générale d'une famille. Par exemple, Boldewin [14] a écrit en 2007 une analyse de la famille *Storm Worm* dans laquelle il détaille sa protection « maison » : deux couches de code servent de protection, et une troisième contient la charge utile. Fallière *et al.* [64] de Symantec décrivent l'emploi de la protection commerciale *VMProtect* par la famille *Clampi* en 2009. Cette protection a la particularité de se baser, entre autres, sur une machine virtuelle. De leur côté, Zaytsev *et al.* [156] ont décrit la protection par machine virtuelle « maison » de la famille *Xpaj*. Tous ces travaux décrivent des exemples, choisis parce qu'ils présentent des originalités, et ils ne peuvent donc servir à montrer une tendance générale.

4.3.3 Recherche en milieu académique

Plusieurs travaux académiques, comme ceux de Martignoni *et al.* [99] et Royal *et al.* [123], ont étudié la *suppression* des protections basées sur le code auto modifiant. La méthode consiste à attendre que le code ait déchiffré une nouvelle couche pour en faire une copie et l'analyser. Cela ne donne pas directement d'indications sur la prévalence des protections basées sur le code auto modifiant. Reynaud [116] présente dans sa thèse une étude de presque 100 000 exemplaires de logiciels malveillants par analyse dynamique. En particulier, il montre que plus de 95% de ceux-ci utilisent du code auto modifiant. Aucune indication n'est donnée sur les fonctionnalités des couches de code, et cela ne nous permet donc pas de tester notre hypothèse sur la séparation des rôles entre les premières couches et la dernière. De plus, ces

résultats souffrent d'un biais important à cause de la méthode de collecte des exemplaires (*honeypot*) : près de 56% des exemplaires traités sont d'une seule et même famille. De leur côté, Roundy *et al.* [122] ont recensé les techniques utilisées par divers protecteurs du domaine public. Les auteurs ne discutent pas l'architecture des programmes protégés, et ne mentionnent pas les protections « maison » qui, comme nous l'avons précédemment décrit, sont désormais courantes parmi les logiciels malveillants.

Ainsi, il n'existe pas d'études à grande échelle sur l'architecture des protections de logiciels malveillants qui pourraient confirmer, ou infirmer, notre hypothèse.

4.4 Méthode de recherche

Le but de notre expérience est de vérifier la validité de notre hypothèse de recherche, et par la même d'améliorer la connaissance des protections de logiciels malveillants. Notre hypothèse postule la prévalence d'un modèle de protection, défini dans le cadre d'une analyse dynamique. Il est donc nécessaire d'avoir un environnement d'expérience permettant d'exécuter un logiciel malveillant sous le contrôle d'un traceur. Cet environnement est décrit en Figure 4.1, où sont notées les trois étapes de notre procédure expérimentale.

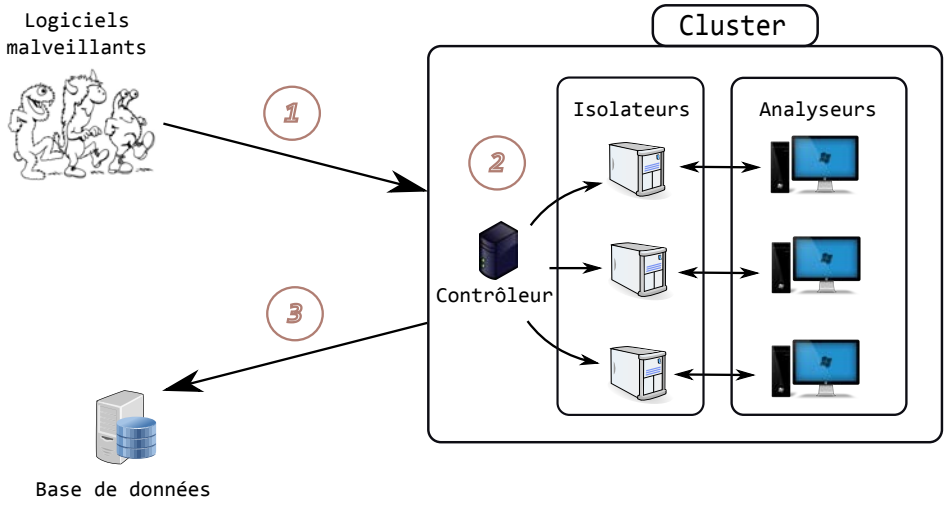


Figure 4.1 Environnement d'expérience

Étape 1 : Sélection des exemplaires à étudier.

Il serait illusoire de vouloir vérifier notre hypothèse sur l'ensemble des exemplaires reconnus malveillants sur Internet, étant donné leur nombre. De ce fait, nous devons sélectionner des

exemplaires de façon à ce que les résultats observés puissent être considérés comme représentatifs d'une tendance parmi les logiciels malveillants.

Par exemple, si nous choisissions aléatoirement des exemplaires parmi des bases de logiciels malveillants publiques, comme [115], nous aurions des biais expérimentaux du fait de la méthode de collecte des exemplaires – les *honeypots* attrapent des *vers*, tandis que les *spamtraps* récupèrent des pièces jointes de courriels –, de la localisation géographique d'où la collecte est effectuée, de sa durée, etc. Ainsi, une base d'exemplaires publique représente plus ou moins fidèlement ce qu'affronte un utilisateur sur Internet, selon la façon dont elle a été construite. Cela n'est pas en soi un problème, puisque de toute façon les utilisateurs ne font pas tous face à la même menace, mais *la représentativité des résultats peut être mise à mal si on ne connaît pas les biais dans la sélection des exemplaires*. En particulier, un grand nombre d'exemplaires uniques ne représente pas nécessairement une grande variété, car ils peuvent simplement être le résultat d'un protecteur versatile appliqué à un même programme initial. **Pour éviter cette situation, nous avons fait le choix de définir nous même nos biais expérimentaux, c'est-à-dire de choisir nos exemplaires à partir de critères que nous considérons pertinents.**

Notre point de départ est la notion de *famille de logiciels malveillants* qui – dans son sens commun – désigne le regroupement sous une même appellation d'exemplaires différents, mais provenant d'une même base de code. En choisissant des exemplaires dans plusieurs familles, on s'assure donc d'avoir de la diversité. Néanmoins, cette notion de famille n'a pas de définition formelle et en pratique son utilisation par les antivirus ne correspond pas toujours à la conception intuitive qu'on peut en avoir. Par exemple, certains antivirus regroupent des exemplaires avec un comportement particulier, comme l'injection dans un processus, dans des familles spéciales. Celles-ci ne contiennent alors pas des exemplaires issus d'une même base de code, mais des exemplaires implémentant un même comportement de différentes manières. Le manque d'informations publiques sur ces familles spéciales les rend inutilisables pour réaliser des expériences rigoureuses. Finalement, remarquons qu'il est commun que des antivirus donnent un nom différent à une même famille. Ainsi, la notion de famille de logiciels malveillants permet de s'assurer de la diversité des exemplaires, à condition de prendre en compte la multiplicité du nommage par les antivirus et de mettre de côté les familles spéciales utilisées par ces mêmes antivirus.

Il nous faut alors déterminer comment choisir nos familles de logiciels malveillants. Commençons par rappeler que notre objectif est l'estimation d'une tendance générale parmi les

familles « grand public », c'est-à-dire qui ne participent pas à des attaques ciblées. Pour ce faire, nous nous sommes alors focalisés sur celles dont on peut penser qu'elles ont *réussi*. Le critère de succès le plus évident est la *prévalence*, c'est-à-dire essentiellement le nombre de fichiers malveillants appartenant à la famille, disponible publiquement par l'intermédiaire des différents *top 10* des éditeurs antivirus. Mais cette mesure a tendance à favoriser artificiellement les familles qui infectent des fichiers légitimes, car elles produisent sur une seule machine de nombreux exemplaires différents, sans pour autant infecter plus d'utilisateurs. De ce fait, nous avons préféré choisir d'autres critères, notamment en observant la réponse apportée par la communauté de la sécurité informatique aux différentes familles :

– **La notoriété.** La communauté de la sécurité informatique, qu'elle soit académique ou industrielle, communique sur les familles de logiciels malveillants qu'elle considère particulièrement dangereuses. Nous mesurons le nombre de ces communications grâce au moteur de recherche personnalisé d'Alexander Hanel [72], qui parcourt les ressources industrielles – typiquement les blogs des entreprises antivirus –, ainsi que le moteur de recherche Google Scholar pour comptabiliser les publications académiques (les *whitepapers* de l'industrie sont considérés comme des publications académiques). Le nombre de résultats sur le nom d'une famille (avec ses alias) donne alors une indication de son importance pour la communauté de la sécurité informatique, et donc de son succès.

– **Les actions offensives.** Cette même communauté a lancé ces dernières années des attaques contre des familles de logiciels malveillants, en particulier pour neutraliser leurs *botnets*, par des moyens techniques [152, 136, 143] ou juridiques [101]. Étant donné les ressources nécessaires à de telles actions offensives, qui sont souvent le fruit d'une collaboration entre différentes entreprises, elles ne sont réalisées que pour les familles particulièrement dangereuses. Cela nous donne donc un critère supplémentaire pour trouver des familles qui ont eu du succès.

Pour résumer, nous considérons que lorsque la communauté de la sécurité montre de l'intérêt pour une famille, cela indique que celle-ci a eu un certain succès. Notre processus de sélection des exemplaires est alors le suivant :

1. Détermination d'une liste de familles de logiciels malveillants dont on peut estimer le succès sur la base des deux critères précédents. Notre sélection est montrée dans le Tableau 4.1. La période d'activité dénote seulement la période pendant laquelle les contrôleurs du logiciel malveillant étaient actifs. En particulier, certains de ses logiciels malveillants continuent toujours de se propager par eux-mêmes. Remarquons aussi que d'après les informations publiques disponibles sur ces familles, aucune d'entre elles n'a été protégée par une protection commerciale.

Tableau 4.1 Familles choisies pour l'expérience

Nom(s) de la famille	Période d'activité	Mentions industrielles	Mentions académiques	Actions offensives
Storm Worm *Nuwar* *Peacomm* *Zhelatin*	2007-2008	53 300	1 190	Attaque contre le protocole pair-à-pair réalisée en 2008 par des chercheurs indépendants [152].
Waledac	2008-2010	34 800	249	Attaque contre le protocole pair-à-pair réalisée en 2010 par Microsoft et ses partenaires [101].
Koobface	2008-2011	103 000	363	Attaque contre l'infrastructure de contrôle menée en 2010 par des chercheurs indépendants [143].
Conficker *Downadup* *Kido*	2008-2011	324 000	1 390	Microsoft a offert en 2009 une récompense de 250 000$ pour toutes informations sur les auteurs du logiciel malveillant [103].
Cutwail *Pushdo*	2008-2013 [1]	48 300	231	Attaque contre l'infrastructure de contrôle menée en 2010 par des chercheurs académiques [137].
Ramnit	2010-2013 [1]	129 000	44	Aucune attaque connue.

1. Toujours en activité au moment de l'écriture de la thèse.

2. Collecte de 100 exemplaires pour chaque famille sélectionnée dans la base d'exemplaires publique *malware.lu* [115]. Ces exemplaires sont **tous reconnus comme membres de la famille par au moins 4 antivirus** parmi une liste d'éditeurs majeurs [1] (en prenant en compte les différences de nommage). Ainsi, le nombre de faux positifs, ou des erreurs de nommage, est réduit, du fait du consensus entre les antivirus. **Nous avons donc analysé 600 exemplaires de logiciels malveillants.**

Remarque 10. Le nombre 100 représente en fait le nombre d'expériences *valides* que nous avons effectué avec chaque famille. En pratique, nous avons dû collecter plus de 100 exemplaires pour y parvenir, car certains d'entre eux n'étaient pas analysables, par exemple parce le fichier était corrompu. La notion de validité des expériences sera définie précisément par la suite.

Étape 2 : Analyse des exemplaires dans l'environnement d'expérience.

Cet environnement, décrit en Figure 4.1, a été construit dans le cluster du laboratoire de sécurité de l'École Polytechnique de Montréal. Le stagiaire Erwann Traourouder a mis en place le système de gestion d'expérience à l'hiver 2013. Ce cluster comprend 98 machines physiques sur lesquelles des machines virtuelles peuvent être déployées, et il est déconnecté physiquement d'Internet. Pour notre expérience, nous avons mis en place les entités suivantes :

– **Les analyseurs.** Ce sont des machines virtuelles avec Windows XP SP3 32 bits, qui suivent l'exécution d'un exemplaire avec notre traceur pendant *20 minutes maximum*, puis analysent la trace d'exécution. Nous détaillerons les mesures effectuées lors de la présentation des résultats. Chaque analyseur fonctionne en paire avec un *isolateur*, qui est la seule machine joignable par son interface réseau.

– **Les isolateurs.** Ce sont des machines virtuelles avec Linux, qui exécutent le simulateur réseau INetSim [76]. Celui-ci répond aux requêtes réseau qui lui parviennent avec des messages standards, afin de faire croire au logiciel malveillant qui s'exécute sur l'analyseur associé qu'il est connecté à Internet. Les isolateurs contrôlent l'expérience sur les analyseurs, et récupèrent les mesures lorsqu'elle est terminée.

– **Le contrôleur d'expérience.** Il se charge de répartir les exemplaires aux analyseurs, en les envoyant à leur binôme isolateur. C'est sur cette machine que l'expérience est définie par l'analyste.

1. ESET, Kaspersky, McAfee, Microsoft, Sophos et Symantec

Au final, **notre environnement comprend 388 couples analyseur/isolateur.**

Remarque 11. La durée d'exécution de 20 minutes a été choisie de façon à récolter une trace d'exécution d'environ 110 millions instructions dynamiques, indépendamment de la famille. D'après notre expérience, cela est habituellement suffisant pour qu'un logiciel malveillant exhibe la forme de sa protection.

Étape 3 : Rapatriement des résultats dans une base de données.

Les résultats de l'analyse sont centralisés dans une base de données, d'où sont extraites les statistiques que nous présenterons par la suite.

4.5 Validation des expériences

Un exemplaire de logiciel malveillant sélectionné durant la première étape de notre procédure expérimentale ne va pas nécessairement donner des résultats valides. Pour expliquer ce phénomène, nous allons définir deux caractéristiques de validation expérimentale, qui nous permettront de considérer toute expérience qui les satisfait comme significative. Nous discuterons également les conditions de fin d'expérience.

4.5.1 Validation technique

Proposition. *Une expérience est valide techniquement si :*

1. *Le traceur a terminé correctement son exécution,*
2. *La trace d'exécution contient plus de 500 instructions dynamiques.*

La validation technique correspond à une barre minimum d'intérêt d'une expérience. Un cas courant d'échec technique est celui des fichiers corrompus. Dans ce cas, une petite trace d'exécution est produite, correspondant à la gestion d'erreur. Une autre possibilité est la mauvaise terminaison de notre traceur, ce qui est dû à son incapacité de suivre l'exécution du programme (il n'est pas transparent). Nous avons corrigé la transparence de notre traceur en le modifiant dans les cas où un nombre conséquent d'exemplaires d'une famille posait problème, comme celui précédemment mentionné en §3.3.2 de l'émulation du *Trap Flag*.

4.5.2 Validation comportementale

Proposition. *Une expérience est valide comportementalement si l'exemplaire analysé a modifié l'état du système lors de son exécution.*

Pour tester ce critère, nous avons défini un ensemble de fonctions de bibliothèque correspondant à une *modification de l'état du système*, comme la création ou la suppression d'un fichier, l'ouverture d'une connexion réseau, la création d'un processus Windows, etc. Lors de notre analyse de la trace d'exécution, nous vérifions alors que le programme a appelé au moins une de ces fonctions, mais aussi que cet appel a réussi, en contrôlant la valeur de retour.

Il s'agit d'un critère fort qui montre que l'exemplaire a exhibé un comportement intéressant durant l'expérience, donnant ainsi de la valeur aux informations récoltées. Notre postulat est *qu'un logiciel malveillant ne modifie le système que s'il pense se trouver dans un environnement à infecter*. Autrement dit, si le programme reconnaît un outil de l'attaquant dans son environnement d'exécution, il ne dévoilera pas les modifications qu'il souhaite apporter au système.

Tous les résultats présentés par la suite concernent des expériences qui sont à la fois des réussites techniques *et* comportementales. Nous avons procédé de façon à ce que cela corresponde à 100 expériences pour chacune des familles sélectionnées. Environ 25% des exemplaires collectés n'ont pas passé la validation technique ou comportementale, en grande partie parce qu'il s'agissait de fichiers invalides (environ 20% des exemplaires collectés).

4.5.3 Conditions de fin d'exécution

Comme expliqué précédemment, nous exécutons chaque logiciel malveillant 20 minutes au maximum.

Tableau 4.2 Conditions de fin d'exécution

	Part des exemplaires
Exécution terminée avant le temps imparti	79.93%

Le Tableau 4.2 montre que 20% des exemplaires n'ont pas terminé leur exécution avant les 20 minutes. Ceci peut s'expliquer par le fait que certains logiciels malveillants deviennent persistants sur le système dans un seul processus, et donc ne terminent jamais leur exécution à proprement parler. Une autre possibilité est qu'ils soient en attente d'un stimulus extérieur, comme une communication réseau spécifique. La validation comportementale décrite précédemment nous assure que nous avons observé au moins une partie du comportement du logiciel malveillant, même pour ceux qui n'ont pas terminé leur exécution.

Remarque 12. L'exécution dont il est question ici est celle du processus initial du logiciel malveillant. En particulier, le logiciel malveillant continue habituellement de s'exécuter sur la machine dans un autre processus ou dans un service Windows.

4.6 Résultats

4.6.1 Auto modification

Dans cette première partie des résultats, nous nous focalisons sur l'auto modification des programmes, c'est-à-dire sur le nombre de couches de code auto modifiant et les relations entre celles-ci.

Nombre de couches de code

Nous présentons en Figure 4.2 le nombre de couches de code – selon la Définition 6 – mesuré lors de l'exécution de nos exemplaires. Cette mesure ne concerne que le premier *thread* de chaque exemplaire, c'est-à-dire celui qui commence l'exécution.

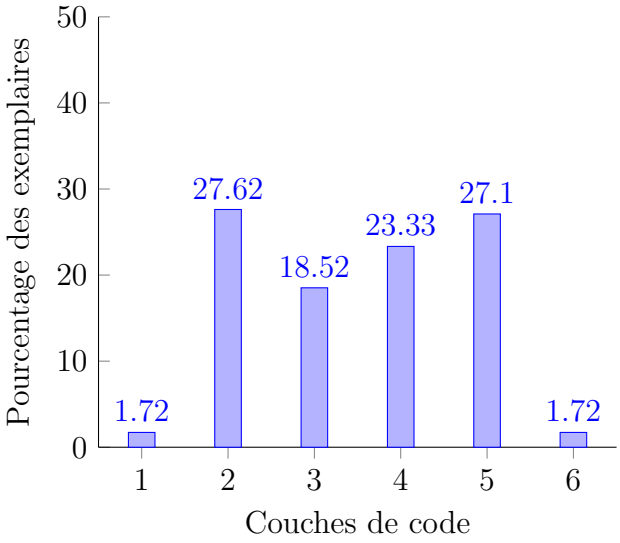


Figure 4.2 Nombre de couches de code

On peut observer que le code auto modifiant est très présent : **98.28% des exemplaires possèdent au moins deux couches de code**. Parmi ceux-ci, il n'apparaît pas de nombre de couches particulièrement privilégié.

Transition entre les couches de code

Chaque couche de code auto modifiant est associée à un indice dans la Définition 6 qui défini un ordre de création : les instructions de la couche d'indice k ont été créées par celles de la couche d'indice $k-1$, pour tout $k \geq 1$. Pour autant, l'ordre d'exécution des couches ne correspond pas nécessairement à leur ordre de création. En effet, rien n'empêche un « retour » de l'exécution sur une couche dont l'indice est inférieur à celui de la couche en cours.

Une exécution dont les couches sont exécutées strictement dans l'ordre de leur création est dite à *croissance monotone*. Dans ce cas, l'exécution ne revient jamais sur une couche créée précédemment. Nous avons mesuré cette caractéristique parmi nos exemplaires avec du code auto modifiant.

Tableau 4.3 Transition entre les couches de code auto modifiant

	Part des exemplaires
Croissance monotone entre les couches	54.1%

Les résultats sont présentés dans le Tableau 4.3 : on peut observer que **près de 55% des exemplaires ont une croissance monotone entre leurs couches.** Autrement dit, plus de la moitié de nos exemplaires possèdent l'architecture la plus simple possible, où chaque couche créée la suivante, qui est ensuite exécutée, pour ne jamais revenir sur une couche créée précédemment. Un cas courant d'exécution non monotone sera discuté en §4.7.2.

4.6.2 Forme des couches de code

Nous mesurons ici des caractéristiques liées à la structure des couches de code, car cela permet de déduire des informations sur leur rôle, comme nous le verrons.

Lecture des résultats

Avant de présenter la suite de nos mesures, nous expliquons ici comment lire nos tableaux de résultats. Pour ce faire, le Tableau 4.4 présente les résultats artificiels de la mesure d'une caractéristique factice, que nous nommerons *C*. **À quelques exceptions près**, tous nos résultats auront par la suite cette forme.

Tableau 4.4 Exemple de résultats artificiels

	...la première couche	...les couches du milieu	...l'avant-dernière couche	...la dernière couche
Part des exemplaires avec C dans...	15%	7%	10%	28%

Ainsi, nous divisons nos résultats en quatre catégories de couche, et le Tableau 4.4 s'interprète alors de la façon suivante :

1. Il y a 15% d'exemplaires qui possèdent la caractéristique C dans *la première couche*, c'est-à-dire celle d'indice 0. Cette couche est présente dans tous les exemplaires.
2. Il y a 7% d'exemplaires qui possèdent la caractéristique C dans *au moins une de leur couche du milieu*, c'est-à-dire une couche dont l'indice est strictement supérieur à 0 et strictement inférieur à l'indice de l'avant-dernière couche. Ainsi, seuls les programmes avec au moins 4 couches possèdent cette catégorie, ce qui fait 52.15% de nos exemplaires d'après la Figure 4.2.
3. Il y a 10% d'exemplaires qui possèdent la caractéristique C dans *l'avant-dernière couche*, c'est-à-dire celle d'indice $m-1$, si m est l'indice de la dernière. Seuls les programmes avec au moins 3 couches possèdent cette catégorie, ce qui fait 70.67% de nos exemplaires.
4. Il y a 28% d'exemplaires qui possèdent la caractéristique C dans *la dernière couche*, c'est-à-dire celle d'indice maximal. Cette couche est présente dans tous les exemplaires avec du code auto modifiant, soit 98.28% de nos exemplaires.

Les couches sont donc divisées selon leur position *relative* dans le programme, ce qui nous permettra de tester la différence de rôle entre les premières et la dernière mentionnée dans notre hypothèse. Insistons sur le fait que les caractéristiques mesurées pourront être présentes dans plusieurs couches pour un même exemplaire, mais aussi ne pas être présentes du tout.

Remarque 13. La catégorie « couches du milieu » regroupe un ensemble de couches, contrairement aux autres qui n'en contiennent qu'une. Ce regroupement peut paraître simplificateur, mais étant donné que le nombre de couches diffère d'un exemplaire à l'autre, il est nécessaire pour unifier les résultats.

Remarque 14. La part des exemplaires qui possèdent une caractéristique dans les couches du milieu et l'avant-dernière est calculée *par rapport à tous les exemplaires*, et pas seulement par rapport à ceux qui possèdent ces couches, afin de ne pas gonfler artificiellement l'importance de celles-ci.

Dans tous les résultats présentés par la suite, nous considérons seulement les exemplaires avec du code auto modifiant, les 1.72% restants (cf. Figure 4.2) ne correspondant pas de fait à notre modèle.

Taille

La taille d'une couche de code est définie comme le nombre d'instructions dynamiques qui la composent dans la trace d'exécution. Remarquons que comme 20% des exemplaires n'ont pas terminé leur exécution (cf. §4.5.3), la taille de la dernière couche est sous-estimée.

Tableau 4.5 Tailles des couches de code auto modifiant

	...la première couche	...les couches du milieu	...l'avant-dernière couche	...la dernière couche
Taille minimale de...	2 687	33	1 964	10
Taille moyenne de...	1 295 995	529 720	5 788 628	246 682
Taille médiane de...	429 294	297 729	719 788	2 484
Taille maximale de...	52 252 488	3 089 428	20 236 026	26 148 985

Les résultats sont présentés dans le Tableau 4.5. Premièrement, la taille maximale de la première couche est significativement plus élevée que pour les autres couches, ce qui peut être **le résultat d'une protection** : une exécution trop longue ne peut être complètement simulée par les émulateurs des antivirus, qui doivent prendre une décision rapidement pour ne pas dégrader les performances et l'expérience de l'utilisateur. Ensuite, **la taille médiane de la dernière couche est particulièrement faible**, ce qui ne peut seulement s'expliquer par sa sous-estimation, mais bien par la présence effective de petites couches. La présence de ces dernières couches de petite taille sera expliquée en §4.7.2. Finalement, les tailles maximales montrent qu'une analyse dynamique de logiciels malveillants doit être capable de gérer **plus de 100 millions d'instructions** pour être complète.

Jeu d'instructions

Une mnémonique est la représentation assembleur de l'*opcode* d'une instruction machine x86, tel que décrit en §2.1.1. Des exemples de mnémoniques sont `jmp`, `nop`, `push`, etc. Intuitivement, une mnémonique regroupe un ensemble d'instructions machines réalisant une même opération mais avec des opérandes différentes. Nous mesurons le nombre de mnémoniques différentes dans chacune des quatre catégories de couches de code.

De plus, nous avons établi une liste de 82 mnémoniques « classiques », présentée en Annexe A. Pour ce faire, nous avons utilisé le désassembleur IDA Pro [74] pour extraire les mnémoniques d'applications fournies avec Windows, car cet outil désassemble de façon fiable les programmes standards, dont il est capable de reconnaître le compilateur. Un ensemble d'instructions sera alors dit *exotique* s'il possède **plus de 10 mnémoniques non classiques**. Nous mesurons le nombre de programmes avec une couche de code exotique, pour chacune des quatre catégories.

Tableau 4.6 Jeu d'instructions

	...la première couche	...les couches du milieu	...l'avant-dernière couche	...la dernière couche
Nombre de mnémoniques différentes dans...	110	63	80	77
Part des exemplaires avec du code exotique dans...	10.12%	0%	0%	0%

Les résultats sont présentés dans le Tableau 4.6. On peut observer que **le nombre de mnémoniques différentes est significativement plus élevé dans la première couche** que dans les autres. En outre, **dans plus de 10% des exemplaires le code de la première couche est exotique**, alors que ça n'est jamais le cas dans les autres catégories.

Ainsi, une limite claire existe entre la première couche et les autres, en terme du jeu d'instructions utilisé. **Pour ces cas d'exotisme**, on peut en déduire que :

– **La première couche de code n'a probablement pas été créée par la même transformation de programmes que les autres**, puisque le choix des instructions diffère. En effet, étant donné la diversité du jeu d'instructions x86, il existe de multiples façons d'implémenter un même comportement, et ce choix est propre à chaque transformation de

programmes en langage machine.

– **L'objectif de la première couche n'est pas la performance.** En effet, les instructions que nous considérons « classiques » sont celles choisies par les compilateurs modernes du fait de leurs performances. À l'inverse, le choix d'utiliser des instructions exotiques (et donc moins performantes) montre que l'objectif de la première couche n'est pas la performance, mais probablement plus la protection. En particulier, la création de code hors normes permet de mettre en difficulté les outils d'analyse conventionnels.

Utilisation des appels de fonctions

Comme expliqué en §2.2.3, la notion de fonction n'existe pas formellement dans le langage machine x86, ce n'est qu'une convention des compilateurs. Ceux-ci implémentent habituellement un appel de fonction avec l'instruction assembleur « `call` », dont la sémantique est de rediriger l'exécution à une adresse cible, et de mettre l'adresse de l'instruction qui la suit en haut de la pile, afin de pouvoir facilement y revenir après avoir exécuté le code de la fonction (habituellement avec l'instruction `ret`). Pour chaque instruction `call`, nous parcourons la suite de la trace d'exécution afin de vérifier si l'adresse sauvegardée est effectivement exécutée ensuite, et nous en déduisons alors si :
– c'est un *bon* appel de fonction, c'est-à-dire que le code revient effectivement sur l'adresse de retour mise en haut de la pile.
– c'est un *mauvais* appel de fonction, c'est-à-dire que le code ne revient jamais sur l'adresse de retour mise en haut de la pile.

Tableau 4.7 Appels de fonctions

	...la première couche	...les couches du milieu	...l'avant-dernière couche	...la dernière couche
Part des exemplaires avec plus de 10 bons appels dans...	15.88%	11.17%	62.13%	60.38%
Part des exemplaires avec plus de 10 mauvais appels dans...	10.82%	0%	0%	0.17%

Les résultats sont présentés dans le Tableau 4.7 pour les quatre catégories de couche :

– **Il y a un nombre significativement plus élevé d'exemplaires avec plus de 10 mauvais appels dans la première couche, par rapport aux autres couches.** Cela montre une utilisation hors norme de l'instruction `call` dans la première couche, qui peut mettre en difficulté les outils d'analyse conventionnels, car ceux-ci l'utilisent pour diviser le programme en fonctions. C'est une technique de protection que nous avons par exemple observée lors de notre analyse de la famille *Swizzor* [33].

– **Il y a un nombre significativement plus faible d'exemplaires avec plus de 10 bons appels dans la première couche, ainsi que dans celles du milieu, par rapport aux autres couches.** Ainsi, l'avant-dernière et la dernière ont tendance à utiliser intensivement la notion habituelle de fonctions, au contraire des premières dans lesquelles elle est rarement présente. Cela indique en particulier qu'un compilateur standard a probablement produit les deux dernières couches.

De façon générale, les mesures présentées ici motivent la recherche d'autres abstractions que les fonctions pour étudier les premières couches, qui n'en comportent pas beaucoup, et qui l'utilisent de façon détournée. C'est ce que nous ferons au chapitre suivant pour analyser les protections.

4.6.3 Rôle des couches de code

Nous nous focalisons ici sur le comportement des couches de code, toujours afin de tester la validité de la division entre les premières et la dernière couche.

Levées d'exceptions

Une exception correspond à un problème logiciel, comme une division par zéro, un accès mémoire invalide, etc. Il existe différents moyens pour « attraper » une exception et la gérer. Nous mesurons le nombre d'exceptions levées par le programme en comptant le nombre d'appels au gestionnaire d'exceptions Windows.

Tableau 4.8 Levées d'exceptions

	...la première couche	...les couches du milieu	...l'avant-dernière couche	...la dernière couche
Part des exemplaires qui déclenchent une exception dans...	20.24%	0.52%	8.9%	0.35%

Les résultats sont présentés dans le Tableau 4.8 : **la première couche lève significativement plus d'exceptions que les autres**. Étant donné que nos expériences ont toutes été validées (cf. §4.7.1), ces exceptions ne sont pas le résultat d'un problème lors de l'exécution, mais plus vraisemblablement d'une technique de protection. En effet, le mécanisme de gestion d'exceptions Windows étant relativement complexe, lever des exceptions met en difficulté les outils d'analyse, ce qui est souvent employé dans les logiciels malveillants [66]. Après analyse manuelle, la levée d'exceptions dans l'avant-dernière couche est effectuée par une seule famille, *Ramnit*, dont un tiers des exemplaires réalise une lecture mémoire invalide à la fin de la boucle de déchiffrement de la dernière couche. Il s'agit alors simplement d'un moyen de déterminer que la dernière couche a été déchiffrée au complet.

Modifications du système

Comme expliqué en §4.5.2, nous ne considérons valides que les expériences où le logiciel malveillant modifie l'état du système. Nous mesurons ici dans quelle catégorie de couches cette modification prend place.

Tableau 4.9 Forte interaction avec le système

	...la première couche	...les couches du milieu	...l'avant-dernière couche	...la dernière couche
Part des exemplaires qui modifie le système dans...	2.09%	1.05%	24.96%	72.95%

Les résultats sont présentés dans le Tableau 4.9. **La dernière couche est, pour plus de 70% des exemplaires, un lieu de modification du système.** Cela montre que la charge utile d'un logiciel malveillant – qui comprend l'ensemble des interactions avec le système, et donc ses modifications – est implémentée très souvent dans la dernière couche. **À l'inverse, la première couche et celles du milieu ne comprennent presque jamais de modifications du système, leur objectif n'est donc pas d'interagir avec celui-ci.** Finalement, un nombre significatif d'exemplaires modifient le système dans l'avant-dernière couche, ce qui montre que celle-ci peut aussi participer à la charge utile du logiciel malveillant, comme nous le discuterons en §4.7.2.

Fonctionnalités

Nous mesurons ici la fonctionnalité d'une couche de code à partir de ses appels aux fonctions de bibliothèques standards. Pour ce faire, nous avons divisé les fonctions de bibliothèques

selon les groupes suivants (* indique un groupe qui participe à la charge utile) :

1. *Fonctions réseau **
2. *Fonctions de manipulation de base de registre **
3. *Fonctions de manipulation des processus **
4. *Fonctions de manipulation des fichiers et répertoires **
5. *Fonctions de résolution de bibliothèques* (permettent à un programme de trouver les adresses de fonctions de bibliothèques à partir de leurs noms)
6. *Fonctions d'allocation de mémoire dynamique*
7. *Fonctions de libération de mémoire dynamique*
8. *Fonctions de lecture d'informations système* (nom de l'ordinateur, version du système, etc.)

Nous considérons alors qu'une couche de code exhibe une fonctionnalité d'un groupe lorsqu'elle réalise **plus de 3 appels aux fonctions du groupe**. Les résultats sont présentés dans le Tableau 4.10.

Tableau 4.10 Part des exemplaires avec une certaine fonctionnalité dans une catégorie de couches.

		Première couche	Couches du milieu	Avant-dernière couche	Dernière couche
Charge utile	1. Réseau	0%	0%	0.17%	19.55%
	2. Base de registre	0%	0%	1.05%	28.62%
	3. Manipulations processus	0.87%	0.35%	7.33%	44.15%
	4. Manipulations fichiers	10.99%	0.35%	7.33%	36.47%
Divers	5. Résolution des bibliothèques	59.86%	24.78%	62.83%	54.28%
	6. Allocation mémoire	19.2%	1.92%	6.81%	19.55%
	7. Libération mémoire	0%	0.17%	5.06%	5.58%
	8. Lecture informations système	13.09%	0%	17.45%	0%

On observe que **la dernière couche est la seule à participer de façon significative – au moins 20% des exemplaires – aux 4 groupes de fonctionnalités propres à la charge utile.** À l'inverse, les autres couches participent peu à la charge utile. Pour les fonctionnalités non reliées à la charge utile, il n'y a pas de tendance qui se dégage.

Remarque 15. Il s'agit d'une sous-estimation de l'interaction avec le système, puisque le programme peut implémenter une fonctionnalité par lui même, sans faire appel à une bibliothèque externe.

4.7 Analyse

4.7.1 Validation de notre hypothèse

Notre hypothèse de recherche stipulait que le modèle de protection avec les caractéristiques suivantes était commun parmi les logiciels malveillants :

1. « *Il y a au moins deux couches de code auto modifiant* » : c'est le cas pour **plus de 98% de nos exemplaires**.

2. « *Les premières couches, c'est-à-dire toutes sauf la dernière, servent à la protection de la charge utile* » : ceci a été validé *en partie*, du fait que :

   - **La première couche est particulièrement agressive** :
     - Son jeu d'instructions est exotique pour plus de 10% des exemplaires, là où celui des autres couches est standard.
     - Elle déclenche des exceptions dans plus de 20% des exemplaires, là où les autres le font rarement.
     - Elle détourne la convention d'appels de fonctions dans plus de 10% des exemplaires, là où les autres ne le font pas.

   - **Les couches du milieu, bien que d'une forme conventionnelle, interagissent très peu avec le système (moins de 2% des exemplaires).** Plus particulièrement, elles ne participent qu'épisodiquement aux fonctionnalités de la charge utile (moins de 1% des exemplaires).

   La première couche interagit avec le système d'une façon similaire à celles du milieu, ce qui nous permet d'affirmer que **toutes les couches qui précèdent l'avant-dernière ne participent pas à l'implémentation de la charge utile, mais plutôt à la mise à l'épreuve des outils d'analyse.**

   D'un autre côté, **le rôle de l'avant-dernière couche n'est pas si clair, étant donné sa participation non négligeable aux interactions avec le système (25%**

des exemplaires). Il semble donc que la limite entre la protection et la charge utile puisse être placée dans certains cas à l'avant-dernière couche, et non à la dernière. Nous discuterons plus en détail les cas dans lesquels cela peut se produire en §4.7.2.

3. « *La dernière couche implémente la charge utile du logiciel malveillant.* » : ceci a été validé par les observations suivantes :

   – **La forme conventionnelle de la dernière couche** : pas d'exotisme dans le jeu d'instructions, très peu d'exceptions levées, et un grand nombre d'appels de fonctions classiques. Ceci tend à montrer que cette couche a été produite par un compilateur standard.

   – **Le système est modifié dans la dernière couche pour plus de 70% des exemplaires.**

   – **Chaque fonctionnalité liée à la charge utile est implémentée pour plus de 20% des exemplaires dans la dernière couche.**

Nous avons donc établi la prévalence du code auto modifiant parmi nos exemplaires et surtout l'existence de deux types de couches de code, celles qui protègent et celles qui implémentent la charge utile. **Cela nous renseigne en particulier sur le fonctionnement des protecteurs utilisés par ces logiciels malveillants : ceux-ci rajoutent simplement des couches de code de protection autour de la charge utile.** Le soin que nous avons apporté à la sélection de nos exemplaires nous permet d'affirmer que ce type de protection a une importance significative parmi les logiciels malveillants en général.

Notre expérience a établi que le rôle d'une couche de code est de protéger *ou* d'interagir avec le système, mais rarement les deux. Cela doit pousser au développement d'analyses dynamiques *différenciées*. Autrement dit, une analyse dynamique devrait prendre en compte le rôle de la couche afin d'appliquer soit un raisonnement classique – par exemple basé sur la notion usuelle de fonctions –, soit des méthodes adaptées au code de protection.

Notre expérience a aussi montré que, contrairement à ce que notre hypothèse stipulait, la limite entre ces deux types de couches n'est pas toujours la dernière, mais que l'avant-dernière couche peut aussi participer à la charge utile. Nous allons maintenant discuter ce phénomène.

Remarque 16. Les programmes étudiés dans ce chapitre sont *tous* étiquetés comme malveillants, et les caractéristiques mises en avant ici ne peuvent donc pas servir à créer des signatures de logiciels malveillants, puisque rien ne prouve a priori que les applications légitimes ne les possèdent pas également.

4.7.2 Rôle de l'avant-dernière couche

Le Tableau 4.9 montre un résultat contradictoire avec notre hypothèse : près de 25% des exemplaires modifient le système dans leur avant-dernière couche, et non la dernière. Pour comprendre ceci, nous avons suivi les étapes suivantes :

1. Nous avons mesuré le nombre d'exemplaires qui interagissent avec le système dans l'avant-dernière *et* la dernière couche. Nous avons trouvé qu'**aucun exemplaire n'interagit avec le système dans les deux couches à la fois**. Cette interaction a donc lieu soit dans l'avant-dernière, soit dans la dernière, mais jamais dans les deux.
2. Cela nous a amené à analyser en profondeur les exemplaires où l'interaction a lieu dans l'avant-dernière couche – soit 25% d'entre eux –, afin de comprendre le rôle de la dernière. Dans ce cas, ces programmes sont membres de deux familles seulement, *Conficker* (dont presque tous les exemplaires sont concernés) et *Ramnit* (la moitié de ses exemplaires), et nous avons pu identifier le même comportement dans les deux cas : **la mise en place d'un trampoline (*hook*)**. Cette technique consiste à écrire au début d'une fonction de bibliothèque standard $\mathcal{F}$ un saut vers le code du logiciel malveillant, afin que ce dernier soit appelé à chaque fois que $\mathcal{F}$ est appelée [88]. C'est une technique courante des logiciels malveillants, qui leur permet par exemple de modifier les paramètres de $\mathcal{F}$ avant de rendre la main à son code.
   La mise en place du trampoline est faite par la charge utile, et le code de celui-ci devient alors la dernière couche du logiciel malveillant. Cela explique en particulier la petite taille des dernières couches dans le Tableau 4.5, puisque celles-ci ne contiennent dans ce cas que les quelques instructions nécessaires au trampoline. Ensuite, cela créée des exécutions non monotones entre les couches de code (cf. Tableau 4.3), puisque l'exécution passe régulièrement du trampoline (dernière couche) à la charge utile (avant-dernière couche). C'est finalement la raison de l'absence d'interaction avec le système dans la dernière couche, et donc du « décalage » de certains exemplaires, pour lesquels l'avant-dernière couche est celle qui réalise la charge utile.

4.7.3 Leçons apprises

La mise en place de notre expérience a été un processus technique difficile ; en particulier nous avons appris les leçons expérimentales suivantes :

– L'analyse de la trace d'exécution est réalisée sur la machine qui a exécuté le logiciel malveillant, et qui est donc infectée. Il peut alors y avoir interaction entre le processus d'analyse et le logiciel malveillant (qui peut continuer à s'exécuter dans un nouveau processus). Par exemple, la famille *Mabezat*, que nous avons brièvement étudiée, chiffrait nos fichiers résultats avant que nous ayons pu les récupérer (son objectif est de demander ensuite une rançon à l'utilisateur en échange de la clé de déchiffrement). Une solution pour éviter cette situation serait de rapatrier la trace sur une autre machine pour en faire l'analyse, mais cela impliquerait de lourds mouvements de fichiers, car les traces font souvent plus de 2 Go.

– Durant nos expériences, nous nous sommes aperçus que la qualité des exemplaires collectés dans les sources publiques, comme [115], est souvent médiocre : environ 20% d'entre eux, selon la famille, sont des fichiers invalides, c'est-à-dire qu'ils ne sont même pas exécutables. Il y a donc une différence significative entre le nombre d'exemplaires collectés, et les expériences réalisables.

4.8 Voies de recherche futures

Au-delà de l'exploration de notre hypothèse de recherche, la mise en place de notre expérience nous a donné un outil puissant pour étudier les protections de logiciels malveillants.

Premièrement, cet outil pourrait nous permettre d'étudier en détail **le choix de la protection au sein d'une même famille**. Certaines familles de logiciels malveillants restent en particulier actives pendant plusieurs années – par exemple la famille *Sality* existe depuis 2003 –, et il serait intéressant de regarder si leur protection a évolué au cours de leur existence, en étudiant des exemplaires produits à différentes époques. Cela permettrait de mesurer si les produits de sécurité ont eu un impact en forçant un changement de méthode de protection, ou si au contraire ces familles ont toujours utilisé le même type de protection. Par exemple, lors de notre étude de la famille *Waledac* [32], nous avons pu étudier des exemplaires « expérimentaux »– cela était mentionné dans leurs noms – dont le type de protection était très différent des exemplaires habituels. Il n'existe à notre connaissance pas de données publiques sur le suivi des protections pour des familles de logiciels malveillants, seules les modifications

apportées à la charge utile sont documentées.

Ensuite, les mesures employées par notre outil pourraient être utilisées pour **créer des signatures de protecteurs**, afin de pouvoir reconnaître et classifier automatiquement les programmes qu'ils ont protégés. Les outils existants étant très limités – des signatures statiques définies sur le code binaire du programme –, il semble particulièrement intéressant d'utiliser les mesures présentées dans ce chapitre. Cela est d'autant plus important que les travaux usuels de classification de programmes – par exemple basés sur le graphe de flot de contrôle – sont inadaptés aux protections, car ils supposent que les programmes suivent les normes habituelles.

Finalement, notre outil pourrait aider à **établir la véritable importance des protecteurs du domaine public**. En effet, d'après nos analyses [32], ces protecteurs publics peuvent être utilisés comme avant-dernière couche (avant la charge utile). Les outils statiques – qui ne voient que la première couche – ne comptabilisent donc pas ces cas. D'un autre côté, lorsque ces protecteurs sont employés en première couche, ils peuvent être combinés avec d'autres protections, ce que notre outil pourrait également identifier.

4.9 Conclusion

Dans ce chapitre, nous avons commencé par établir un cadre semi-formel pour la notion de protection, puis nous avons posé une hypothèse de recherche stipulant qu'un certain modèle de protection est particulièrement prévalent dans les logiciels malveillants. Ce modèle est basé sur le code auto modifiant et la distinction entre les premières couches, qui protègent, et la dernière, qui implémente la charge utile. Après avoir montré que les travaux existants ne nous permettent pas de confirmer ou infirmer notre hypothèse, nous avons présenté l'expérience que nous avons construite afin de le faire.

Ceci nous a permis de valider l'importance du code auto modifiant parmi un groupe de logiciels malveillants particulièrement dangereux, ainsi que l'existence des deux types de couches de code. Néanmoins, bien que dans notre hypothèse seule la dernière couche implémente la charge utile, nous avons trouvé des cas où l'avant-dernière couche y participe aussi. De ce fait, notre hypothèse a été partiellement validée, et ce processus expérimental nous a permis de mettre en avant des caractéristiques des protections de logiciels malveillants.

CHAPITRE 5

IDENTIFICATION DE FONCTIONS CRYPTOGRAPHIQUES

L'étude des implémentations de fonctions cryptographiques est d'une grande importance pour la compréhension des logiciels malveillants, en particulier dans leurs protections où elles servent de base au code auto modifiant. Ces implémentations pouvant être celles de fonctions cryptographiques connues de l'analyste, cela ouvre la voie à leur identification automatique, qui permet d'éviter l'analyse manuelle. Cette identification ne peut se faire qu'en prenant en compte l'obfuscation des protections, qui fait disparaître un certain nombre de repères présents dans les programmes classiques.
Nous présentons dans ce chapitre une méthode d'identification des fonctions cryptographiques adaptée au contexte des protections. Nous introduirons d'abord notre problème de recherche, pour ensuite évoquer les travaux existants sur le sujet et montrer leur inadéquation à notre contexte. Finalement, nous présenterons notre solution en détail, ainsi que sa validation expérimentale. Une partie de ce chapitre a été publiée dans [38].

5.1 Problème de recherche

Les logiciels malveillants emploient souvent la cryptographie afin de protéger leurs communications réseau [25, 34, 134], ou encore pour chiffrer des informations dans le but de demander une rançon à l'utilisateur [3]. Mais ce ne sont pas leurs seules raisons d'utiliser de la cryptographie, elle leur sert également à protéger leur charge utile. En effet, tel que montré au chapitre 4, le code auto modifiant est un moyen classique de protection des logiciels malveillants, et il peut s'implémenter par le biais de fonctions cryptographiques, qui déchiffrent alors une partie du programme [14, 155, 112]. Nous nous focalisons ici sur cet emploi de la cryptographie à des fins de protection du code, en particulier car l'obfuscation qui l'englobe rend une analyse spécifique nécessaire.

La compréhension de ces implémentations cryptographiques de protection du code est précieuse pour l'analyste. Premièrement, elle lui permet de collecter précisément la charge utile déchiffrée, et ceci avant qu'elle ne soit exécutée. Ensuite, pour peu qu'il comprenne la façon dont le déchiffrement s'effectue, il peut automatiser celui-ci et l'introduire dans un antivirus. Cela l'autorise ensuite à écrire des signatures sur la charge utile elle-même, qui est habituellement peu variable entre les exemplaires d'une même famille. Finalement, les implémentations

cryptographiques constituent des caractéristiques intéressantes pour la classification des logiciels malveillants, et peuvent renseigner sur les groupes qui les ont créés.

L'implémentation d'une routine cryptographique peut être faite soit à partir d'un algorithme documenté, soit par la conception d'un nouvel algorithme. Les auteurs de logiciels malveillants étant des programmeurs (à peu près) comme les autres, ils ont tendance à privilégier le chemin du moindre effort, et donc à choisir des algorithmes du domaine public ou, tout du moins, à réutiliser souvent les mêmes. De ce fait, un analyste peut établir une base de référence des algorithmes cryptographiques qu'il est susceptible de rencontrer dans les protections, soit parce qu'ils sont dans le domaine public, soit parce qu'il les a déjà rencontrés dans ce contexte. Cela ouvre alors la voie à une alternative à l'analyse manuelle des implémentations cryptographiques : *l'identification automatique de la fonction implémentée*. En d'autres termes, il s'agit de déterminer si un programme en langage machine implémente une fonction cryptographique connue.

5.2 Travaux existants

La détection d'implémentations cryptographiques dans un programme exécutable, et en particulier l'extraction de leurs paramètres, a été étudiée précédemment dans divers contextes. Dans le cadre de l'analyse de preuves numériques, Halderman *et al.* [71] ont utilisé des propriétés particulières des algorithmes DES, AES et RSA afin de retrouver leurs paramètres en présence de *bit flipping errors* en mémoire. Maartmann-Moe *et al.* [97] ont étendu ce travail aux algorithmes Serpent et Twofish. Ces méthodes se basant sur des caractéristiques spécifiques à ces chiffrements, elles nécessitent une étude en profondeur de chacun d'entre eux.

Noé Lutz [96] est le premier à avoir exploré la *localisation* d'implémentations cryptographiques, c'est-à-dire trouver dans un programme les endroits où la cryptographie est possiblement implémentée. La méthode de Lutz est basée sur trois indicateurs : (1) la présence de boucles, (2) un ratio élevé d'instructions arithmétiques, et (3) un changement fort d'entropie entre les données d'entrée et de sortie manipulées par une implémentation cryptographique. Ensuite, Wang *et al.* [148] et Caballero *et al.* [25] ont utilisé des observations similaires pour localiser la cryptographie, notamment dans des logiciels malveillants. Plusieurs suppositions faites dans ces travaux ne sont pas réalistes dans le cadre de programmes obfusqués. En particulier, les instructions arithmétiques sont souvent employées à outrance dans les protections de logiciels malveillants, afin de rendre la compréhension du code plus complexe, et elles ne sont donc pas caractéristiques de la cryptographie. Plus récemment, Felix Matenaar

et al. [100] ont résumé ces travaux en comparant différentes méthodes de localisation cryptographique. Ils ont également développé une méthode de suivi des données pour mesurer la diffusion – caractéristique des algorithmes cryptographiques – de certaines données dans le Graphe de Flot de Contrôle (GFC). Mais la validation expérimentale de leur travail n'a été faite que sur des programmes standards, sans obfuscation. De façon générale, aucun de ces travaux ne vise *l'identification* de la fonction implémentée.

Gröbert *et al.* [69] ont proposé en 2010 un travail sur l'identification de fonctions cryptographiques. Ils ont d'abord travaillé sur la génération des meilleures signatures syntaxiques possible pour plusieurs algorithmes cryptographiques, en comparant un ensemble d'implémentations. Ces méthodes sont inefficaces en présence d'obfuscation, puisque le code aura tendance à cacher les signes syntaxiques de la fonction qu'il implémente. Dans le même travail, les auteurs ont développé une méthode d'identification basée sur la comparaison des paramètres d'entrée-sortie du code cryptographique dans une trace d'exécution. Malgré la similitude avec la méthode que nous allons présenter dans ce chapitre, les auteurs ont fait des choix non adaptés à notre contexte. Par exemple, les paramètres cryptographiques sont reconstruits à partir de la proximité des instructions qui les manipulent dans la trace d'exécution. En présence d'obfuscation, cette notion de proximité serait difficile à utiliser, puisque l'insertion de code inutile – méthode d'obfuscation classique – rend la proximité entre instructions utiles très variable. Zhao *et al.* [157] ont aussi développé une méthode basée sur la comparaison des entrées-sorties pour identifier des implémentations cryptographiques. Encore une fois, de nombreuses suppositions faites sur les programmes cibles, comme le ratio de ou-exclusifs dans le code cryptographique, ou l'utilisation d'un certain type de fonctions, sont rarement vraies dans les programmes obfusqués.

Ainsi, bien que ces travaux aient ouvert la voie à l'identification de fonctions cryptographiques, il sont inadaptés à notre contexte d'analyse et ne peuvent donc répondre de façon satisfaisante à notre problème de recherche.

5.3 Solution proposée

Avant de décrire en détail la méthode que nous avons mise en place, nous présentons l'intuition qui a servi à la concevoir, pour ensuite discuter brièvement son implémentation.

5.3.1 Intuition

Nous recherchons une propriété conservée dans toutes les implémentations d'une même fonction, et qui ne serait donc pas susceptible d'être modifiée par la présence de l'obfuscation. Commençons par poser $\mathcal{P}$, un programme qui calcule la fonction $[\![\mathcal{P}]\!]$, et $\mathcal{F}$, une fonction. Il est clair que si $[\![\mathcal{P}]\!] = \mathcal{F}$, c'est-à-dire si $\forall x$, $[\![\mathcal{P}]\!](x) = \mathcal{F}(x)$, alors $\mathcal{P}$ peut être identifié comme une implémentation de la fonction $\mathcal{F}$, et ceci quel que soit sa forme. Cette équivalence fonctionnelle est malheureusement un problème indécidable. Par contre, on peut s'en inspirer pour énoncer une propriété décidable : *soit* une entrée x telle que $\mathcal{P}$ termine son exécution sur x, si $[\![\mathcal{P}]\!](x) = \mathcal{F}(x)$, alors $\mathcal{P}$ est une implémentation de la fonction $\mathcal{F}$ *sur l'entrée* x. Cette nouvelle propriété s'accorde bien à l'analyse dynamique telle que nous l'avons définie au chapitre 3, c'est-à-dire l'analyse d'une exécution *a posteriori*, dont la terminaison est donc assurée.

Afin de reformuler notre intuition dans le contexte cryptographique, posons $\mathcal{D}$, une fonction cryptographique telle que $\mathcal{D}(c,k)$ renvoie le texte déchiffré à partir du texte c et de la clé k. On peut alors énoncer la propriété suivante : *si durant une exécution du programme* $\mathcal{P}$ *les entrées* c *et* k *ont été utilisées pour produire la valeur* $\mathcal{D}(c,k)$*, alors* $\mathcal{P}$ *a implémenté la fonction* $\mathcal{D}$ *durant cette exécution.*

Remarque 17. Un tel raisonnement ne permet pas de conclure sur les autres exécutions de $\mathcal{P}$, mais énumérer les chemins d'exécution d'un logiciel malveillant – augmenter la couverture de l'analyse dynamique – est un autre problème de recherche.

Remarque 18. On peut imaginer que deux fonctions cryptographiques $\mathcal{D}_1$ et $\mathcal{D}_2$ puissent vérifier $\mathcal{D}_1(c,k) = \mathcal{D}_2(c,k)$, notamment parce que nous autorisons les fonctions créées par les auteurs de logiciel malveillant eux-mêmes, et pas seulement les fonctions cryptographiques usuelles (pour lesquelles nous ne connaissons pas de telle paire c et k). Dans ce cas, nous concluons que P a implémenté *les* fonctions $\mathcal{D}_1$ et $\mathcal{D}_2$ durant l'exécution, ce qui est correct.

5.3.2 Survol

La solution que nous proposons pour l'identification des fonctions cryptographiques dans des programmes en langage machine obfusqués s'inscrit dans le cadre du processus d'analyse dynamique décrit en Figure 3.3. Nous récoltons donc une trace d'exécution, sur laquelle nous appliquons une analyse en deux étapes :

1. **Extraction du code cryptographique et de ses arguments :** Afin d'obtenir les possibles arguments cryptographiques contenus dans la trace d'exécution, nous avons défini un critère permettant de distinguer le code qui les manipule du reste des instructions. Pour cela, nous nous appuyons sur une notion de *boucle*, pour laquelle nous définissons des paramètres d'entrée-sortie. Ensuite, nous observons le flux de données entre les boucles pour regrouper celles participant à un même algorithme. Finalement, nous obtenons une abstraction nommée *boucles avec flux de données* qui contient un possible algorithme cryptographique avec ses arguments.

2. **Comparaison avec des fonctions cryptographiques connues :** Nous disposons pour chaque fonction cryptographique connue d'une implémentation de référence dans un langage de haut niveau. Nous pouvons alors vérifier si une de ces implémentations produit la même valeur de sortie à partir des mêmes valeurs d'entrée qu'un des algorithmes extraits de la trace. Si c'est le cas, nous pouvons conclure que celui-ci implémente la même fonction que l'implémentation de référence (durant l'exécution observée). En dépit de son apparente simplicité, cette étape de comparaison se heurte à la différence d'abstraction entre les implémentations de références et la trace d'exécution.

Nous allons maintenant décrire en détail notre solution. Nous commencerons par l'étape d'extraction du code cryptographique, pour ensuite expliquer la phase de comparaison.

5.4 Extraction du code cryptographique et de ses arguments

Le code cryptographique ne constitue pas la totalité d'une trace d'exécution, mais simplement une partie. Nous avons donc besoin d'un critère pour distinguer les instructions qui pourraient participer à un algorithme cryptographique.

5.4.1 Boucles

Motivation

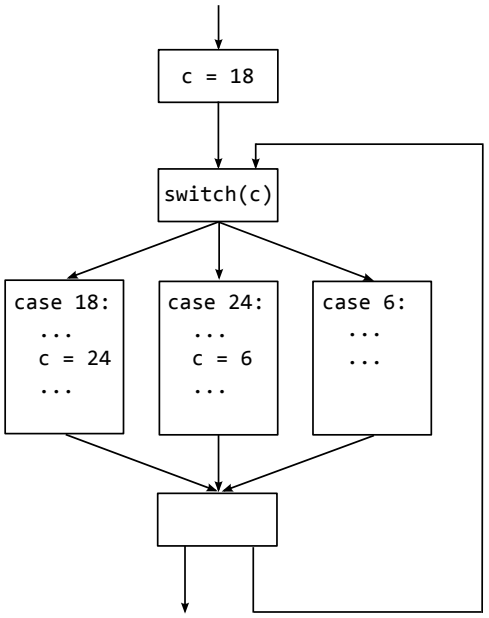
Les critères habituels de localisation des algorithmes cryptographiques – comme les constantes ou les instructions arithmétiques – ne sont pas fiables dans du code obfusqué, tel qu'expliqué en §5.2. De plus, la notion de fonction n'étant qu'une convention de compilateurs rarement respectée dans les protections – tel que montré au chapitre 4 –, elle ne peut pas servir à l'analyse de ces protections. Nous avons donc besoin de notre propre abstraction pour distinguer les possibles algorithmes cryptographiques et retrouver leurs paramètres.

Dans sa thèse sur la localisation de fonctions cryptographiques, Noé Lutz écrit la remarque suivante : « *loops are a recurring feature in decryption algorithms* » [96]. En effet, les algorithmes cryptographiques appliquent habituellement *un même traitement de façon itérative sur leurs paramètres*, ce qui fait des boucles une structure très commune dans leurs implémentations. En particulier, c'est à l'intérieur de ces boucles que le cœur de l'algorithme se trouve, ce qui en fait donc un point de départ intéressant pour notre abstraction.

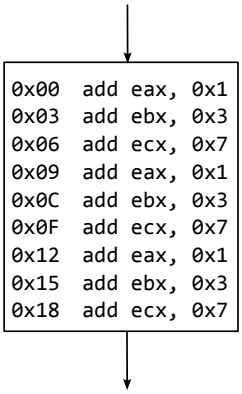
Néanmoins, il y a des comportements itératifs dans la majorité des programmes informatiques, et pas seulement lorsqu'ils implémentent de la cryptographie. Il nous faut donc d'abord définir précisément ce que nous entendons par boucle. Pour illustrer ce besoin, nous présentons en Figure 5.1(a) une technique d'obfuscation nommée *écrasement de flux de contrôle* [147] – en pseudo-C pour simplifier les explications – qui est utilisée par exemple dans le logiciel malveillant *Mebroot* [62]. Une partie de programme séquentielle, c'est-à-dire sans boucles, est transformée en une « boucle » qui, à chaque itération, réalise une partie *différente* du code original, par le biais d'une variable de contrôle. Ainsi, une logique différente est exécutée à chaque itération. Est-ce que cela doit être considéré comme une boucle dans notre contexte cryptographique ? Ensuite, la Figure 5.1(b) présente une optimisation classique de compilateurs, qui peut aussi être employée à des fins d'obfuscation, dans laquelle une boucle est « déroulée ». Pour ce faire, ses différentes itérations sont accolées explicitement, en supprimant les branchements conditionnels et donc les arcs retour. Dans cet exemple, une séquence de 3 instructions est répétée trois fois, est-ce que cela doit être considéré comme une boucle dans notre contexte cryptographique ?

La reconnaissance de boucles dans un programme est un problème classique, et plusieurs définitions de boucles ont été proposées pour les programmes en langage machine. En particulier, les trois définitions suivantes :

1. **Boucles naturelles :** C'est la définition habituelle des boucles dans le cadre de l'analyse statique, par exemple dans le domaine de la compilation [2]. Des arcs retour sont définis sur le *Graphe de Flot de Contrôle (GFC)* comme des arcs entre un nœud a et un de ses dominants b, c'est-à-dire que b précède forcément a dans tous les chemins d'exécution possibles. Chacun de ces arcs correspond alors à une boucle différente, dont le corps est constitué des noeuds du graphe qui peuvent rejoindre a sans passer par b. Par conséquent, avec cette définition la Figure 5.1(a) est une boucle, mais pas la Figure 5.1(b).
2. **Boucles sur les adresses :** Une autre définition, construite spécialement pour l'analyse dynamique, est celle de Tubella *et al.* [139]. Une boucle est alors identifiée par



(a) Écrasement du flux de contrôle



(b) Déroulement de boucles

Figure 5.1 Graphe de flot de contrôle de cas limites de boucles

une adresse mémoire « cible » où sautent un ensemble d'arcs retour. À la différence de la définition précédente, plusieurs arcs retour peuvent donc correspondre à une même boucle. Le corps de la boucle est alors défini comme l'intervalle mémoire entre l'adresse cible et son arc retour le plus éloigné. Cette définition considère aussi la Figure 5.1(a) comme une boucle, mais pas la Figure 5.1(b).

3. **Boucles sur les instructions :** Kobayashi définit de son côté les boucles comme une répétition d'une même séquence d'instructions machine dans une trace d'exécution [84]. Par exemple, si T est une trace d'exécution telle que $T_{/Ins} = I_1; I_2; I_3; I_1; I_2; I_3$, alors T est une boucle qui itère deux fois avec $I_1; I_2; I_3$ comme corps de boucle. Avec cette simple définition, la Figure 5.1(a) n'est pas considérée comme une boucle, tandis que la Figure 5.1(b) l'est.

Boucles simples

La notion de boucle représente dans notre contexte l'application d'un *même traitement* sur les paramètres d'entrée-sortie. De ce fait, la Figure 5.1(a) ne devrait pas être considérée comme une boucle – puisqu'une logique différente est exécutée à chaque « itération »– et par contre la Figure 5.1(b) devrait l'être. Nous avons donc choisi de suivre l'approche de Kobayashi : une boucle est identifiée par la répétition d'une même séquence d'instructions machine (son corps). Ainsi, la Figure 5.1(b) est une boucle dont le corps est constitué des trois instructions

`add eax, 0x1`, `add ebx, 0x3` et `add ecx, 0x7`.

Une même boucle peut être exécutée plusieurs fois durant l'exécution d'un programme, à chaque fois avec possiblement un nombre d'itérations différent. Nous nommons une de ces exécutions particulières une *instance* de la boucle. De plus, nous considérons que la dernière itération n'a pas besoin d'être complète, c'est-à-dire qu'une instance ne termine pas forcément à la dernière instruction de son corps de boucle. Nous pouvons alors définir la notion d'instance de boucle à l'aide d'un langage formel sur l'alphabet des instructions machine $\mathcal{X}86$. Pour un mot $\alpha \in \mathcal{X}86^*$, nous notons $Pref(\alpha)$ l'ensemble des préfixes de α, c'est-à-dire $\beta \in Pref(\alpha)$ si $\exists \gamma \in \mathcal{X}86^*, \alpha = \beta.\gamma$. De plus, nous notons $\alpha \in \mathcal{X}86^+$ quand $\alpha \in \mathcal{X}86^*$ et $|\alpha| \geq 1$.

Definition 7. *Le langage SLOOP des instances de boucles simples est défini comme l'ensemble des $\mathcal{L} \in TRACE$ telles que :*

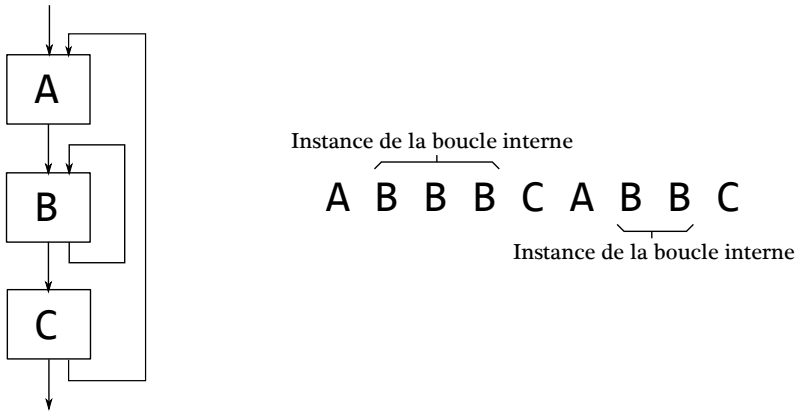
$$\mathcal{L}_{/Ins} \in \{\alpha^n.\beta \mid \alpha \in \mathcal{X}86^+, n \geq 2, \beta \in Pref(\alpha)\}$$

Une instance de boucle simple est donc constituée d'au moins deux répétitions d'une séquence d'instructions machine, appelée son corps et notée α dans la Définition 7. Cette notion est en fait plus large que celle de Kobayashi, dont la définition interdit qu'un corps de boucle puisse contenir deux fois la même instruction machine. Autrement dit, il n'y a pas d'instruction $\ell \in \mathcal{X}86$ telle que $\alpha = u\ell u'\ell u''$ dans le travail de Kobayashi. L'auteur a fait ce choix pour simplifier la reconnaissance des boucles. Avant de présenter notre propre algorithme de reconnaissance, nous devons enrichir notre langage pour prendre en compte l'imbrication de boucles.

Boucles imbriquées

L'imbrication de boucles est une construction courante dans les programmes informatiques. La Figure 5.2(a) présente le GFC simplifié d'une telle situation, et la Figure 5.2(b) la trace d'exécution associée. Le bloc B possède un arc retour sur lui même, tandis qu'un deuxième arc retour englobe les blocs A, B et C. Dans la trace d'exécution, il y a deux instances de boucle basées sur le bloc B, mais qui n'itèrent pas le même nombre de fois. Par conséquent, l'application directe de la Définition 7 ne peut pas reconnaître une boucle basée sur l'arc retour extérieur, car ses deux « itérations » ne sont pas composées exactement de la même séquence d'instructions. Néanmoins, il s'agit d'une situation cohérente avec notre notion de boucle : le même traitement est appliqué de façon répétée.

Pour résoudre ce problème, il suffit d'abstraire chaque boucle du langage $SLOOP$, en la remplaçant par un symbole représentant son corps, et de réappliquer la Définition 7. En



(a) GFC (b) Trace d'exécution

Figure 5.2 Exemple très simplifié d'imbrication de boucles.

d'autres termes, chaque instance d'une même boucle est remplacée par le même *identifiant de boucle*. Nous introduisons alors l'ensemble $\mathcal{L}_{ID}$ des identifiants de boucle. Par exemple, la trace d'exécution de la Figure 5.2(b) est réécrite comme $A\ X\ C\ A\ X\ C$, où $X \in \mathcal{L}_{ID}$ est l'identifiant de la boucle interne. La prochaine application de la Définition 7 peut alors reconnaître la boucle externe comme telle, avec $A\ X\ C$ comme corps de boucle. Nous pouvons maintenant formaliser cette notion de boucles, en notant $TRACE^{\mathcal{L}_{ID}}$ l'ensemble des traces d'exécution où des identifiants de boucles peuvent remplacer des instructions dynamiques.

Definition 8. *Le langage LOOP des instances de boucles est défini comme l'ensemble des* $\mathcal{L} \in TRACE^{\mathcal{L}_{ID}}$ *telles que :*

$$\mathcal{L}_{/Ins} \in \{\alpha^n.\beta \mid \alpha \in (\mathcal{X}86 \cup \mathcal{L}_{ID})^+, n \geq 2, \beta \in Pref(\alpha)\}$$

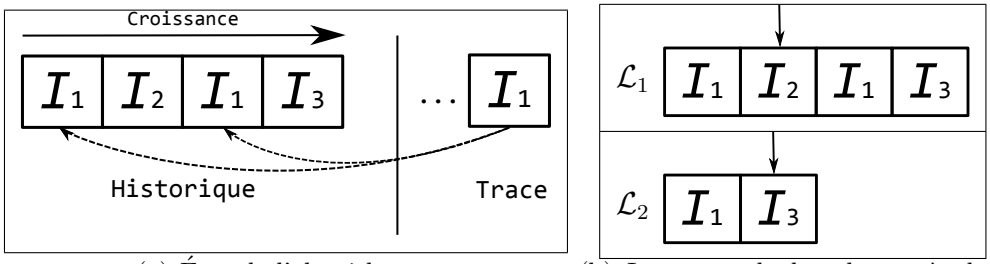
Pour $\mathcal{L} \in LOOP$, nous notons $BODY[\mathcal{L}] \in (\mathcal{X}86 \cup \mathcal{L}_{ID})^+$ son corps, c'est-à-dire son α dans la Definition 8.

Algorithme de reconnaissance des instances de boucles

La détection des boucles dans une trace d'exécution revient donc à reconnaître le langage $LOOP$ de la Définition 8. Celui-ci étant une extension du langage $\{\omega.\omega\}$ – *non* hors contexte [75] – il n'existe pas de grammaire hors contexte à partir de laquelle nous pourrions générer automatiquement un analyseur syntaxique. Nous avons donc dû construire notre propre algorithme de détection de boucles. Avant de le présenter, nous allons donner un exemple de son fonctionnement pour faciliter sa compréhension.

Exemple

L'algorithme de reconnaissance du langage *LOOP* parcourt successivement les instructions machine d'une trace d'exécution et les stocke à la fin d'une liste nommée *historique*. Une situation commune est alors celle décrite en Figure 5.3(a) : les instructions I_1, I_2, I_1, I_3 ont déjà été enregistrées dans l'historique et l'instruction en cours de traitement est I_1. Celle-ci apparaît donc deux fois dans l'historique.

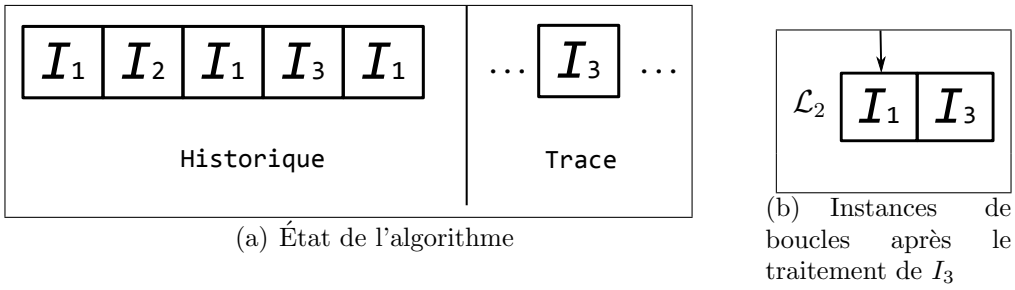


(a) État de l'algorithme

(b) Instances de boucles après le traitement de I_1

Figure 5.3 Exemple de détection de boucles : étape une

Chacune des deux occurrences de I_1 dans l'historique correspond à un possible début d'instance de boucle, c'est-à-dire à un mot de la forme $\alpha.\alpha$. Dans le premier cas, le corps de la boucle serait $\alpha = I_1; I_2; I_1; I_3$, tandis que dans le second $\alpha = I_1; I_3$. De ce fait, notre algorithme créé deux instances de boucle, nommées respectivement $\mathcal{L}_1$ et $\mathcal{L}_2$, et pour chacune d'entre elles un *curseur* est positionné sur la prochaine instruction attendue : I_2 pour $\mathcal{L}_1$ et I_3 pour $\mathcal{L}_2$ (cf. Figure 5.3(b)).



(a) État de l'algorithme

(b) Instances de boucles après le traitement de I_3

Figure 5.4 Exemple de détection de boucles : étape deux

L'instruction I_1 est alors ajoutée à la fin de l'historique. Maintenant, supposons que I_3 est la prochaine instruction dans la trace, tel que montré dans la Figure 5.4(a). $\mathcal{L}_1$ est alors supprimée des instances de boucle en cours, puisqu'elle n'attendait pas cette instruction. De son côté, $\mathcal{L}_2$ voit son curseur incrémenté, et revenir sur la première instruction du corps de

boucle : une nouvelle itération vient d'être observée (cf. Figure 5.4(b)). À ce moment, nous avons vu exactement deux itérations de $\mathcal{L}_2$, c'est-à-dire la suite d'instructions $I_1; I_3; I_1; I_3$ et nous pouvons donc considérer $\mathcal{L}_2$ comme une instance de boucle *confirmée*. Par conséquent, nous remplaçons son code dans l'historique par $X \in \mathcal{L}_{ID}$, son identifiant de boucle associé, tel que décrit dans la Figure 5.5

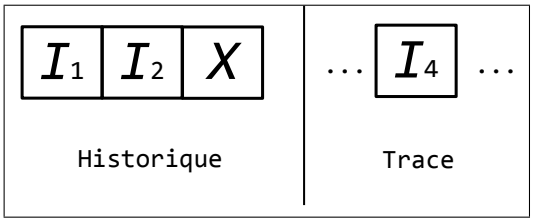


Figure 5.5 Exemple de détection de boucles : étape trois

Supposons maintenant que la prochaine instruction machine dans la trace est I_4. Alors l'instance de boucle $\mathcal{L}_2$, qui attendait I_1, est supprimée des instances de boucles en cours et *enregistrée*. Le remplacement de son code par son identifiant X permettra la détection d'une boucle externe, indépendamment du nombre d'itérations de ses instances, tel que mentionné en §5.4.1.

Pseudo-code

Nous présentons maintenant le pseudo-code de l'algorithme de reconnaissance du langage $LOOP$. Pour ce faire, nous utilisons les structures de données suivantes :

- $Historique$: liste où sont stockées les instructions machine déjà vues, ainsi que les identifiants de boucles déjà reconnues. Dans le pseudo-code la variable de type $Historique$ est notée H.
- $InstanceBoucle$: structure contenant les informations relatives à une instance de boucle, comme son corps et son curseur.
- $BouclesEnCours$: ensemble des instances de boucles *en cours* ; chacune d'entre elles est une pile représentant une imbrication : la tête contient l'$InstanceBoucle$ la plus imbriquée, tandis que le reste de la pile contient les boucles extérieures, dans l'ordre de leur imbrication. Dans le pseudo-code la variable de type $BouclesEnCours$ est notée RL.

Le pseudo-code de la procédure principale est donné dans l'Algorithme 2, tandis que la logique interne de la détection de boucle est dans la procédure récursive $Match()$, décrite dans l'Algorithme 3. Certaines des opérations sont décrites en langage naturel pour faciliter la compréhension, et des détails sont omis pour la même raison.

L'algorithme est construit à partir d'une règle simple : une instruction machine est *soit* dans une instance de boucle confirmée (il n'y en a qu'une à la fois), *soit* le début possible de nouvelles instances de boucle. Ce principe est exprimé dans la procédure principale : la boucle de la ligne 3 vérifie pour chaque instruction si une instance de boucle confirmée l'attend et, si c'est le cas (la procédure $Match()$ retourne alors 1), les autres instances ne sont pas testées et l'instruction n'est pas considérée comme un possible début de boucle. D'un autre côté, si aucune instance de boucle n'attend cette instruction, alors la seconde partie de la procédure principale – à partir de la ligne 9 – vérifie si elle pourrait être un début de boucle. La procédure récursive $Match()$ se charge, en plus de vérifier si une instruction est attendue, de la gestion des instances de boucles (incrémentation du curseur, ajout de l'identifiant dans l'historique, etc.).

Remarque 19. De par sa façon de traiter les instructions, notre algorithme retourne les boucles minimales et les plus à gauche dans la trace d'exécution.

Algorithme 2 Procédure principale de la détection d'instance de boucle

Entrées : *T : TRACE*, *H : Historique*, *RL : BouclesEnCours*

```
 1: pour i = 1 à Longueur(T) faire
 2:         AttenduParInstanceConfirmee ← Faux
 3:         pour PileDeBoucles dans RL faire
 4:                 si Match(PileDeBoucles, I[D_i], H) = 1 alors
 5:                         AttenduParInstanceConfirmee ← Vrai
 6:                         stop
 7:         si AttenduParInstanceConfirmee = Faux alors
 8:                 Ajouter(H, I[D_i])
 9:                 si d'autres occurrences de I[D_i] sont dans H alors
10:                         Créer les instances de boucles commençant par I[D_i]
11:                         Ajouter les nouvelles instances dans RL
12:         sinon
13:                 Jeter toutes les boucles, sauf celle qui est confirmée
```

Algorithme 3 Procédure Match()

Entrées : *PileDeBoucles : Pile(InstanceBoucle)*, $I_j : \mathcal{X}86$, *H : Historique*

```
 1: si PileDeBoucles est vide alors
 2:         retourner 0 // Cas de base
 3: instanceActuelle ← Tête(PileDeBoucles) // Prendre la boucle la plus imbriquée
 4: si instanceActuelle.curseur pointe sur un identifiant de boucle X alors
 5:         Incrémenter instanceActuelle.curseur
 6:         Créer nouvelle instance I pour la boucle X
 7:         Empiler(PileDeBoucles, I)
 8:         retourner Match(PileDeBoucles,I_j,H)
 9: sinon
10:         si instanceActuelle.curseur pointe sur I_j alors
11:                 Incrémenter instanceActuelle.curseur
12:                 si instanceActuelle itère pour la seconde fois alors
13:                         Supprimer les instructions de instanceActuelle dans H
14:                         instanceActuelle.confirmee ← 1
15:                         Ajouter(H, instanceActuelle.ID)
16:                 si instanceActuelle.confirmee = 1 alors
17:                         retourner 1
18:                 sinon
19:                         retourner 0
20:         sinon
21:                 si instanceActuelle.confirmee = 1 alors
22:                         Dépiler(PileDeBoucles)
23:                         retourner Match(PileDeBoucles,I_j,H)
24:                 sinon
25:                         Jeter PileDeBoucles
26:                         retourner 0
```

Complexité

Nous estimons la complexité en temps du pire cas de l'algorithme à partir des observations suivantes :

– La boucle principale itère m fois, où m est la taille de la trace.

– Le nombre de boucles en cours est $\frac{m}{2} - 1$ dans le pire cas. Par exemple, pour $T \in TRACE$ telle que $T/ins = I_1; I_2; I_3; I_1; I_2; I_3$, le pire cas se produit lorsque l'algorithme analyse la deuxième occurrence de I_3, deux boucles sont alors en cours (celles dont les corps sont $I_1; I_2; I_3$ et $I_2; I_3; I_1$).

– La fonction $Match()$ s'exécute en temps constant, sauf lorsqu'il y a des appels récursifs. Un appel récursif correspond à une imbrication de boucle, et le nombre maximal d'imbrications de boucles est inférieur à $log(m)$, pour une trace de taille m. En effet, si une boucle A contient une boucle B, alors une instance de A possède au moins deux fois plus d'instructions que la plus petite instance possible de B, puisque cette instance de A itère au moins deux fois. L'imbrication des boucles peut ainsi être schématisée sous la forme d'un arbre, qui est binaire dans le pire cas.

– La recherche dans l'historique d'autres occurrences d'une instruction en ligne 9 nécessite au maximum $\frac{m}{2}$ comparaisons. En effet, il n'est pas nécessaire de remonter tout l'historique si on a déjà testé plus de $\frac{m}{2}$ instructions, car une boucle ne peut être de taille supérieure à m (ceci n'est pas mentionné dans l'algorithme pour simplifier la compréhension).

La boucle principale réalise donc m fois une opération nécessitant dans le pire cas $((\frac{m}{2} - 1) * log(m) + \frac{m}{2})$ opérations, ce qui donne une complexité en temps de $O(m^2 * log(m))$.

5.4.2 Paramètres de boucles

Les boucles permettent de distinguer le possible code cryptographique dans une trace d'exécution, mais l'objectif final de notre phase d'extraction est de récolter ses paramètres d'entrée-sortie. Nous présentons donc ici une notion de paramètres de boucles, ainsi que l'algorithme pour les extraire de la trace. Finalement, nous étudierons un exemple basique pour faciliter la compréhension.

Définition

Les paramètres de boucles sont les équivalents à bas niveau des paramètres des implémentations en langage haut niveau (que nous appellerons *paramètres de haut niveau* par la suite). Les octets lus et écrits par chaque instruction dynamique d'une trace constituent notre point de départ et, pour une instance de boucle $\mathcal{L}$, nous définissons ses paramètres par les trois conditions nécessaires suivantes :

1. Les octets appartenant à un paramètre *d'entrée* de $\mathcal{L}$ ont été lus sans être précédemment écrits par le code de $\mathcal{L}$, tandis que les octets appartenant à un paramètre de *sortie* ont été écrits par le code de $\mathcal{L}$. Cela correspond à la notion intuitive de ce qu'est une entrée, une valeur utilisée, et une sortie, une valeur produite.
2. Les octets appartenant à un même paramètre de $\mathcal{L}$ sont soit *adjacents en mémoire*, soit *dans un même registre au même moment*. Cette condition seule aurait tendance à regrouper différents paramètres de haut niveau dans le même paramètre de $\mathcal{L}$. En effet,

il n'est pas rare que des paramètres de haut niveau se retrouvent à côté en mémoire, en particulier quand ils sont dans la pile. Une telle sur approximation compliquerait fortement la phase de comparaison finale, puisqu'il faudrait décomposer les paramètres de boucles. C'est la raison pour laquelle nous avons introduit la condition suivante.

3. Les octets appartenant à un même paramètre de $\mathcal{L}$ sont *manipulés de la même façon* (lus ou écrits) par la *même* instruction dans $BODY[\mathcal{L}]$. En effet, une instruction à une certaine position dans le corps de boucle peut manipuler des adresses différentes à chaque itération – par le biais de registres jouant le rôle de pointeurs – mais ces adresses tendent alors à désigner le même type de paramètres.

Afin de définir nos paramètres, nous introduisons maintenant une notion de *variable concrète* qui représente un tableau d'octets démarrant à une adresse mémoire particulière, ou bien encore un registre. Si une variable concrète démarre à l'adresse `0x400000` et contient 4 octets, nous la notons alors `0x400000:4`. Similairement, nous notons `eax:4` les 4 octets contenus dans le registre `eax`. La valeur associée à une variable concrète peut évoluer au cours d'une exécution, un paramètre est alors une variable concrète dont la valeur est fixée.

Algorithme de construction des paramètres

Pour simplifier les explications, nous décrivons cet algorithme en termes informels. Étant donné une instance de boucle extraite d'une trace, l'algorithme regroupe les octets dans des variables concrètes en appliquant les deux dernières conditions nécessaires décrites précédemment. Ensuite, ces variables sont séparées en deux groupes, paramètres d'entrée et paramètres de sortie, en appliquant la première condition (la même variable concrète peut alors être dans les deux groupes).

Dans un second temps, l'algorithme associe à chaque variable une valeur, pour en faire un paramètre. Comme expliqué au §3, notre traceur collecte les valeurs de chaque accès fait par une instruction dynamique. À partir de cela, nous fixons la valeur d'un paramètre avec les deux règles suivantes : (1) la *première* fois qu'un paramètre d'entrée est lu fournit sa valeur, et (2) la *dernière* fois qu'un paramètre de sortie est écrit fournit sa valeur.

Finalement, pour chaque instance de boucle $\mathcal{L}$, l'algorithme retourne les ensembles $IN_M(\mathcal{L})$ et $IN_R(\mathcal{L})$ contenant respectivement les paramètres d'entrée en mémoire et registres, et les ensembles $OUT_M(\mathcal{L})$ et $OUT_R(\mathcal{L})$ contenant les paramètres de sortie. Le besoin de distinguer entre les paramètres mémoire et registres sera expliqué en §5.4.3. Cet algorithme fonctionne en $O(m)$ étapes, avec m la taille de la trace d'exécution.

Exemple

Dans le but de faciliter la compréhension des définitions précédentes, nous présentons ici un exemple basique et artificiel. La Figure 5.6 présente un programme *RASM* implémentant le chiffrement du masque jetable, ou chiffrement de Vernam [127], c'est-à-dire l'application d'un ou-exclusif entre une clé et un texte chiffré de même longueur. Le programme réalise ce chiffrement entre la clé `0xDEADBEEFDEADBEEF` et le texte `0xCAFEBABECAFEBABE`. Afin d'identifier la fonction cryptographique par l'approche présentée dans ce chapitre, nous devons collecter les deux paramètres d'entrée avec le résultat associé, c'est-à-dire `0x1453045114530451`.

$\mathscr{C}(P)$		P
a	$\Sigma[a_{/k}],\ k \in \mathbb{N}$	
		`DATA:`
`0x0`	`DE AD BE EF DE AD BE EF`	`0xDEADBEEF 0xDEADBEEF`
`0x8`	`CA FE BA BE CA FE BA BE`	`0xCAFEBABE 0xCAFEBABE`
`...`	`00 00 ...`	
		`CODE:`
`0x100`	`8B 35 00 00 00 00`	`mov esi, 0x0`
`0x106`	`8B 3D 08 00 00 00`	`mov edi, 0x8`
`0x10C`	`B9 00 00 00 00`	`mov ecx, 0x0`
`0x111`	`8B 06`	`mov eax, [esi]`
`0x113`	`8B 1F`	`mov ebx, [edi]`
`0x115`	`33 C3`	`xor eax, ebx`
`0x117`	`89 07`	`mov [edi], eax`
`0x119`	`83 C6 04`	`add esi, 0x4`
`0x11C`	`83 C7 04`	`add edi, 0x4`
`0x11F`	`83 C1 04`	`add ecx, 0x4`
`0x122`	`83 F9 08`	`cmp ecx, 0x8`
`0x125`	`75 EA`	`jnz 0x111`
`0x127`	`F4`	`halt`

Figure 5.6 Programme *RASM* implémentant le chiffrement du masque jetable. Le texte chiffré est déchiffré sur place en mémoire.

Le résultat que donne notre méthode de reconstruction des paramètres est présenté en Figure 5.7 sous la forme d'un *graphe de paramètres*, qui représente les paramètres d'entrée en

orange, et les paramètres de sortie en bleu, pour chaque boucle extraite de la trace. Il s'agit d'un exemple fictif à but pédagogique, puisqu'en réalité notre outil – que nous présenterons par la suite – fonctionne sur une machine x86 et non pas la machine abstraite *RASM*.

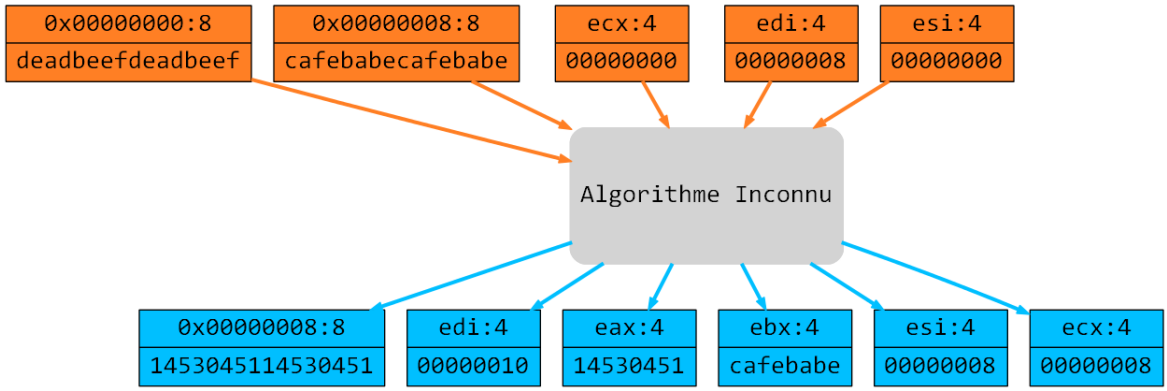


Figure 5.7 Graphe de paramètres. Les nœuds orange sont les paramètres d'entrée, tandis que les bleus sont les paramètres de sortie. Leurs valeurs sont notées en dessous de la ligne.

Ainsi, un algorithme inconnu a été extrait, et on peut observer que les paramètres cryptographiques ont bien été récupérés : en entrée `0x00000000:8` contient la clé (lue par l'instruction à l'adresse `0x111`) et `0x00000008:8` le texte d'entrée (lue en `0x113`), tandis qu'en sortie `0x00000008:8` contient le texte déchiffré (écrit en `0x117`). Remarquons que nous avons aussi collecté des paramètres spécifiques à cette implémentation : (i) `esi:4` et `edi:4` contiennent des adresses mémoires, (ii) `ecx:4` contient le compteur de boucle, et (iii) `eax:4` et `ebx:4` contiennent des valeurs intermédiaires du calcul. Ces paramètres sont donc liés à cette implémentation et ils devront être mis de côté lors de l'étape de comparaison.

5.4.3 Flux de données entre boucles

Jusqu'à présent, nous avons considéré que chaque possible implémentation d'une fonction cryptographique ne comprenait qu'une seule boucle. Mais, certains algorithmes cryptographiques sont en fait habituellement constitués de différents comportements itératifs, comme l'algorithme RC4 [118] qui en comprend trois. De ce fait, notre notion de boucle ne peut capturer complètement ces algorithmes. Pour remédier à cela, nous introduisons maintenant une notion de flux de données entre boucles afin de regrouper celles participant à une même implémentation cryptographique, en suivant un raisonnement similaire au *dynamic slicing* [1]. Nous allons donc décrire la construction de ce flux de données, ce qui nous amènera à notre modèle final, dont nous définirons ensuite les paramètres associés.

Construction

Le flux de données entre des instances de boucles est une notion semblable aux *def-use chains* utilisées en compilation [2]. Intuitivement, deux instances de boucles $\mathcal{L}_1$ et $\mathcal{L}_2$ sont connectées par un flux de données, si $\mathcal{L}_1$ produit un paramètre de sortie qui est ensuite utilisé par $\mathcal{L}_2$ comme paramètre d'entrée. Pour des raisons de performance, nous ne considérons que les paramètres mémoire pour construire le flux de données, car les paramètres registres nécessiteraient un suivie précis dans le code séquentiel (entre les instances de boucles). En d'autres termes, nous supposons que les paramètres mémoire ne sont modifiés que par l'intermédiaire des boucles, et qu'ils suffisent à rendre compte du flux de données d'un algorithme cryptographique. Pour simplifier les explications, la boucle $\mathcal{L}_i$ est à la position i parmi celles extraites de sa trace, selon l'ordre de commencement des boucles.

Definition 9. *Soient $\mathcal{L}_1, \ldots, \mathcal{L}_n$ les instances de boucles extraites d'une trace, il y a un flux de données de $\mathcal{L}_i$ vers $\mathcal{L}_j$, noté $\mathcal{L}_i \trianglelefteq \mathcal{L}_j$, si :*

1. *$\mathcal{L}_i$ termine avant que $\mathcal{L}_j$ ne commence*
2. *$\exists\ a \in OUT_M(\mathcal{L}_i) \bigcap IN_M(\mathcal{L}_j)$, $\nexists\ k \in \rrbracket i; j \llbracket$, $a \in OUT_M(\mathcal{L}_k)$*

Il y a donc un flux de données lorsqu'au moins un octet est transmis sans modifications entre deux boucles. Nous allons maintenant raisonner à partir du graphe $\mathcal{G} = (\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \trianglelefteq)$, qui est orienté et acyclique, avec possiblement plusieurs composants connectés $\mathcal{G}_1, \ldots, \mathcal{G}_m$. Pour un composant connecté $\mathcal{G}_k$, nous notons $ROOT[\mathcal{G}_k]$, et $LEAF[\mathcal{G}_k]$, les ensembles d'instances de boucles *racines*, et *feuilles*, de $\mathcal{G}_k$, c'est-à-dire respectivement les nœuds sans arcs entrants, et sans arcs sortants. Nous construisons le graphe en testant la relation binaire $\trianglelefteq$ pour chaque paire d'instances de boucles, et nous obtenons ainsi ses composants connectés. Chacun d'entre eux constitue alors une abstraction semblable à la notion de fonction dans les programmes classiques. Autrement dit, $\mathcal{G}_k$ est un candidat pour être testé comme une implémentation d'une fonction cryptographique. Par la suite, nous appellerons les composants connectés des *Boucles avec Flux de Données (BFD)*.

En cas de compositions de plusieurs fonctions cryptographiques, c'est-à-dire si la sortie d'une fonction est utilisée en entrée d'une autre fonction, elles vont être regroupées dans le même BFD, ce qui rendra leur identification compliquée (il faudrait avoir une implémentation de référence pour chaque combinaison possible entre les fonctions cryptographiques connues). De ce fait, nous considérons en pratique chaque *sous-graphe* d'un composant connecté comme un BFD. Par exemple, supposons que $\mathcal{G}$ est le graphe $(\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}, \trianglelefteq)$, tel que $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ et $\mathcal{L}_2 \trianglelefteq \mathcal{L}_3$, alors nous considérons comme possible implémentation cryptographique $\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}$, mais aussi $\{\mathcal{L}_1, \mathcal{L}_2\}$, $\{\mathcal{L}_2, \mathcal{L}_3\}$, et finalement chaque instance de boucle seule. Grâce à cela,

nous sommes capables d'identifier les fonctions cryptographiques utilisées en combinaison entre elles, comme nous le montrerons lors de la validation expérimentale.

Remarque 20. La façon dont nous construisons le flux de données suppose que nous étudions des programmes non parallèles, c'est-à-dire des programmes dans lesquels seul le *thread* analysé modifie les données.

Paramètres de BFD

Les BFD sont notre modèle d'implémentation cryptographique et notre objectif final reste l'extraction de leurs paramètres. Nous définissons ces paramètres comme les paramètres mémoire des instances de boucles qui composent un BFD, mais qui *ne participent pas au flux de données interne* du BFD. Pour les paramètres registres, nous considérons simplement ceux en entrée des premières instances de boucles (les racines du BFD), et ceux en sortie des dernières (les feuilles).

Definition 10. *Soit* $\mathcal{G}_k = (\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \trianglelefteq)$ *un BFD, ses paramètres d'entrée sont :*

$$IN_{\mathcal{G}_k} = \bigcup_{1 \leq i \leq n} \left(IN_M(\mathcal{L}_i) - \bigcup_{\mathcal{L}_j \trianglelefteq \mathcal{L}_i} OUT_M(\mathcal{L}_j) \right) \bigcup_{\mathcal{L}_r \in ROOT[\mathcal{G}_k]} IN_R(\mathcal{L}_r)$$

et ses paramètres de sortie sont :

$$OUT_{\mathcal{G}_k} = \bigcup_{1 \leq i \leq n} \left(OUT_M(\mathcal{L}_i) - \bigcup_{\mathcal{L}_i \trianglelefteq \mathcal{L}_j} IN_M(\mathcal{L}_j) \right) \bigcup_{\mathcal{L}_l \in LEAF[\mathcal{G}_k]} OUT_R(\mathcal{L}_l)$$

Les *valeurs* de ses paramètres ont été collectées durant l'extraction des paramètres d'instances de boucles. Nous avons donc maintenant un modèle pour extraire les paramètres des possibles implémentations cryptographiques, et nous pouvons alors construire l'étape de comparaison elle-même.

5.5 Comparaison

La dernière étape de notre méthode d'identification est la comparaison entre les BFD extraits d'une trace et les implémentations de référence des fonctions cryptographiques connues. Nous avons alors deux types d'entrées :

- Les valeurs des paramètres dans $IN_{\mathcal{G}_k}$ et $OUT_{\mathcal{G}_k}$ pour chaque BFD $\mathcal{G}_k$ extrait de la trace d'exécution.
- Une implémentation de référence $\mathbf{P}_{\mathcal{F}}$ dans un langage de haut niveau pour chaque fonction cryptographique connue $\mathcal{F}$. En particulier, le prototype de l'implémentation décrit

ses paramètres, par exemple s'ils sont de taille variable ou fixe. Le choix d'utiliser des implémentations de référence dans un langage de haut niveau facilite l'exécution de ces implémentations avec des paramètres arbitraires.

L'objectif de la comparaison est alors d'examiner si la relation entre les valeurs dans $IN_{\mathcal{G}_k}$ et celles dans $OUT_{\mathcal{G}_k}$ est aussi vérifiée par $\mathbf{P}_{\mathcal{F}}$. En d'autres termes, il s'agit de tester si les valeurs de sortie de l'implémentation de référence sont dans $OUT_{\mathcal{G}_k}$ lorsque celle-ci est exécutée sur des valeurs d'entrée choisies dans $IN_{\mathcal{G}_k}$. Si c'est le cas, on peut alors conclure que la partie de la trace d'exécution représentée par $\mathcal{G}_k$ implémente la même fonction que $\mathbf{P}_{\mathcal{F}}$, c'est-à-dire $\mathcal{F}$. Nous allons maintenant décrire les difficultés qui rendent cette étape de comparaison non triviale, pour ensuite décrire l'algorithme lui-même.

Difficultés

Comme nous utilisons des implémentations de référence dans des langages de haut niveau, il y a une différence d'abstraction avec le code extrait d'une trace d'exécution. De ce fait, la correspondance des paramètres entre ces deux types d'implémentation n'est pas forcément directe, en particulier pour les raisons suivantes :

– **Types des paramètres.** Les paramètres des BFD sont constitués d'octets extraits de la mémoire, ou des registres, de la machine. D'un autre côté, les paramètres de haut niveau peuvent être manipulés avec un type de données tel qu'une structure ou un objet. De ce fait, l'encodage des paramètres au niveau machine peut différer sensiblement de la représentation attendue par les implémentations de haut niveau, par exemple parce qu'une structure est réorganisée par le compilateur afin de prendre moins de place, ou encore parce que les grands nombres sont encodés de façon spéciale. Cela pourrait mener à un échec de la comparaison lorsqu'une même valeur est représentée différemment à bas niveau et à haut niveau. C'est pour cela que nous avons choisi (sauf exception) des implémentations de référence qui prennent des paramètres au niveau d'abstraction le plus bas possible, c'est-à-dire sans types complexes et avec le même encodage que ceux habituellement utilisés au niveau machine.

– **Ordre des paramètres.** Une implémentation de haut niveau déclare ses paramètres dans un certain ordre, tandis que les paramètres des BFD sont non ordonnés. De ce fait, nous devons tester tous les ordres possibles pour mettre les paramètres des BFD aux bonnes positions dans les implémentations haut niveau.

– **Fragmentation des paramètres.** Un même paramètre cryptographique de haut niveau peut se retrouver dans plusieurs paramètres de BFD, par exemple parce qu'il est contenu dans plusieurs registres. De ce fait, les valeurs des paramètres de BFD doivent être combinées pour reconstruire la valeur du paramètre cryptographique. Ainsi, la relation entre les paramètres de BFD et les paramètres cryptographiques n'est pas 1-à-1, mais n-à-1.

– **Nombre des paramètres.** Les paramètres d'un BFD ne sont pas tous des paramètres cryptographiques, certains sont liés à l'implémentation elle-même, tel que montré dans l'exemple du §5.4.2. Par conséquent, ces paramètres n'auront pas d'équivalent dans les implémentations de haut niveau.

Algorithme de comparaison

Étant donné les difficultés présentées précédemment, l'algorithme de comparaison essaye d'identifier un BFD $\mathcal{G}_k$ en suivant les étapes suivantes :

1. **Génération de toutes les combinaisons possibles d'entrée-sortie.** Les valeurs dans $IN_{\mathcal{G}_k}$ sont combinées en les joignant *bout à bout*. Nous générons les combinaisons de toutes les longueurs possibles (chaque valeur de $IN_{\mathcal{G}_k}$ étant présente une seule fois dans une combinaison). La même chose est faite avec les valeurs dans $OUT_{\mathcal{G}_k}$. Par exemple pour le chiffrement de masque jetable du §5.4.2, nous générons pour les paramètres d'entrée 4 valeurs de longueur 4, 8 valeurs de longueur 8 (les 6 paires possibles avec les arguments de longueur 4, plus les deux arguments de longueur 8), etc. En pratique, le nombre de valeurs générées peut être réduit en mettant de côté les paramètres qui dépendent de l'implémentation, comme les adresses mémoires.

2. **Correspondance des paramètres d'entrée.** Pour chaque implémentation cryptographique de référence $\mathbf{P}_{\mathcal{F}}$, l'algorithme sélectionne pour chacun de ses paramètres d'entrée les valeurs dont la longueur correspond, parmi celles générées lors de l'étape précédente. En particulier, si aucune valeur compatible n'est trouvée, alors la comparaison ne se fera pas avec cette implémentation.

3. **Comparaison.** Le programme $\mathbf{P}_{\mathcal{F}}$ est exécuté successivement sur chacune des combinaisons possibles des valeurs d'entrée précédemment sélectionnées. Si à un moment les valeurs produites sont présentes dans l'ensemble des valeurs de sortie générées lors de l'étape 1, alors c'est un succès. Sinon, l'algorithme itère jusqu'à ce que toutes les

combinaisons aient été testées.

La complexité de cet algorithme est exponentielle en fonction du nombre de paramètres, mais nous verrons qu'en pratique celui-ci reste raisonnable.

5.6 Validation expérimentale

Nous avons construit un outil, nommé Aligot, appliquant le processus d'identification décrit précédemment à une trace d'exécution, tel que décrit en Figure 5.8. Au final, l'outil est constitué d'environ 2 000 lignes de Python et a été rendu disponible en [29]. Les implémentations de référence sont construites à partir de la bibliothèque PyCrypto [93].

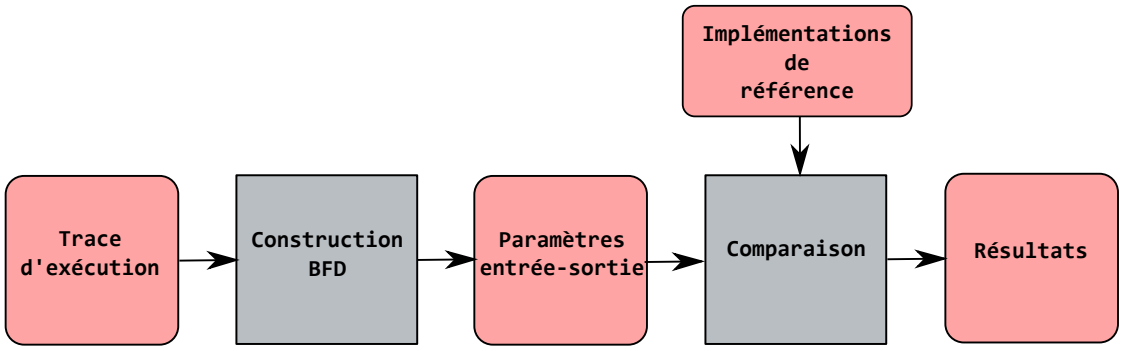


Figure 5.8 Architecture d'Aligot

Pour évaluer notre approche, nous avons testé Aligot sur des implémentations des algorithmes *TEA* et *RC4* qui sont couramment employés par des logiciels malveillants dans leurs protections. Dans un second temps, nous avons étendu nos expérimentations à des implémentations de *MD5* et *AES*, afin de montrer la généricité de notre méthode. Finalement nous avons expérimenté avec l'algorithme *RSA*, ainsi qu'avec des modifications et des combinaisons d'algorithmes.

Dans chacun de nos tests, nous avons comparé Aligot aux outils d'identification cryptographique existants actuellement, et décrits dans le Tableau 5.1.

5.6.1 Tiny Encryption Algorithm (TEA)

L'algorithme TEA est un chiffrement par bloc de 8 octets à l'aide d'une clé de 16 octets, et construit comme un réseau de Feistel à 64 rondes, c'est-à-dire avec un seul comportement itératif – une seule *boucle* [151]. Une de ses particularités est l'utilisation d'une constante

Tableau 5.1 Outils d'identification cryptographique. La colonne « S/D » indique si l'outil est statique ou dynamique.

Nom	Version	Auteur(s)	S/D	Commentaires
Draca [87]	0.5.7b	Ilya O. Levin	S	
FindCrypt [70]	2.0	Ilfak Guilfanov	S	Plugin pour IDA Pro
Hash&Crypto Detector	1.4	Mr. Paradox	S	
Kerckhoffs [69]	1.0	Felix Gröbert	D	Thèse de maîtrise
KANAL [131]	2.92	snaker et igNorAMUS	S	Plugin pour PEiD
Signsrch [4]	0.1.7a	Luigi Auriemma	S	
SnD Crypto Scanner	0.5	Loki	S	Plugin pour OllyDbg

spéciale, nommée *delta*, dont la valeur est fixée à `0x9E3779B9` dans la spécification de l'algorithme.

Exemples synthétiques

Afin de vérifier la correction de notre outil, nous avons créé trois exemples synthétiques à partir du code source publié dans l'article original décrivant TEA [151]. Dans chacun de ces exemples, nous avons appelé l'algorithme de déchiffrement sur la clé `0xDEADBEE1...DEADBEE4` (16 octets) et sur le texte chiffré `0xCAFEBABECAFEBABE` (8 octets). Nous avons alors obtenu les programmes suivants :

– $\mathcal{B}_1$: code source original de TEA, compilé avec le compilateur *Microsoft Visual Studio (MVS)* sans aucune optimisation (option `/Od`).

– $\mathcal{B}_2$: code source original de TEA, compilé avec *MVS* avec une optimisation maximale de la taille (option `/O1`). Cela a pour effet de changer la forme du code et ses instructions : alors que dans $\mathcal{B}_1$ la boucle de déchiffrement de TEA est traduite en trois *Bloc Basique (BB)* et 35 instructions machines, dans $\mathcal{B}_2$ cette boucle comprend un seul BB de 25 instructions.

– $\mathcal{B}_3$: similaire à $\mathcal{B}_1$, excepté que nous avons remplacé dans le programme compilé l'initialisation de la constante *delta* (`mov reg, delta`) par une version légèrement obfusquée de trois instructions (`mov reg, 0` et `add reg, delta/2` deux fois). De ce fait, la sémantique de l'algorithme est préservée, mais la constante *delta* n'est plus visible statiquement.

Les résultats des différents outils sur ces programmes sont présentés dans le Tableau 5.2. Nous pouvons les interpréter de la façon suivante :

Tableau 5.2 Résultats d'identification sur TEA. Le nom de la fonction identifiée par chaque outil est écrit et ✕ indique une absence d'identification. La couleur indique la correction du résultat : vert pour un résultat correct, et rouge à l'inverse (cf. Remarque 21)

	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_3$	*StormWorm*	*SilentBanker*
Aligot	TEA	TEA	TEA	RTEA	RTEA
Draca	TEA	TEA	✕	✕	TEA
FindCrypt2	✕	✕	✕	✕	✕
Hash&Crypto Detector	TEA	TEA	✕	✕	TEA
Kerckhoffs	✕	✕	✕	✕	✕
KANAL	TEA	TEA	✕	✕	TEA
Signsrch	TEA	TEA	✕	✕	TEA
SnD Crypto Scanner	✕	✕	✕	✕	✕

– $\mathcal{B}_1$ et $\mathcal{B}_2$ permettent de calibrer l'évaluation. Les outils n'arrivant pas à identifier ces implémentations classiques ne doivent pas être considérés comme pertinents, puisqu'ils n'ont en fait probablement pas de moyens de reconnaître TEA. Néanmoins, nous les mentionnons par souci d'exhaustivité.
– $\mathcal{B}_3$ est identifié comme une implémentation de TEA seulement par Aligot. Nous pouvons donc supposer que tous les autres outils basent entièrement leur détection sur la visibilité statique de la constante *delta*.

Exemples de protections de logiciels malveillants

Nous avons examiné deux familles de logiciels malveillants, nommées *Storm Worm* et *Silent Banker*, qui ont été publiquement référencées comme utilisant TEA [14, 112]. Dans les deux cas, l'implémentation de l'algorithme fait partie des couches de protection du programme. Nous avons donc collecté un exemplaire de ces deux familles et testé les différents outils d'identification, dont les résultats sont présentés dans le Tableau 5.2.

Storm Worm. Aucune implémentation de TEA n'est identifiée, et ce pour tous les outils, incluant Aligot. Cela signifie en particulier que la relation entrée-sortie de chaque BFD extrait de la trace d'exécution n'est pas reproductible avec une implémentation de TEA, selon notre outil. Pour confirmer cela, nous avons étudié en profondeur la partie de la protection de notre exemplaire décrite comme une implémentation de TEA par des sources publiques [14] : (i) le code se comporte comme un chiffrement par bloc de 8 octets, avec une clé de 16 octets, exactement comme TEA, (ii) la constante *delta* est utilisée durant le processus de

déchiffrement, et (iii) les opérations arithmétiques sont similaires à celles de TEA (décalage à droite de 5 bits et à gauche de 4 bits).
Néanmoins, nous avons trouvé une différence. La Figure 5.9(a) représente en pseudo-C une ligne extraite de l'implémentation de *Storm Worm*, tandis que la Figure 5.9(b) montre la partie équivalente du code source original de TEA. Dans la version du logiciel malveillant, on peut observer que les parenthèses sont autour des ou-exclusifs, alors que dans le code original elles englobent les additions. Ainsi, la même expression est évaluée dans un ordre différent, ce qui explique que la relation entrée-sortie ne corresponde pas.

$$\boxed{z = z - (y << 4) + (k[2] \oplus y) + (sum \oplus (y >> 5)) + k[3]}$$
(a) Storm Worm

$$\boxed{z = z - ((y << 4) + k[2]) \oplus (y + sum) \oplus ((y >> 5) + k[3])}$$
(b) TEA

Figure 5.9 Comparaison entre implémentations

Nous pensons que les auteurs de *Storm Worm* ont simplement fait un copier-coller d'une source contenant cette erreur. En effet, nous avons trouvé exactement la même implémentation – à l'instruction assembleur près – sur un site Internet russe [128]. De ce fait, nous avons nommé ce nouvel algorithme *Russian Tiny Encryption Algorithm (RTEA)*. Cette observation a pu être confirmée en ajoutant une implémentation de référence pour RTEA dans Aligot, qui a ensuite identifié avec succès la fonction dans *Storm Worm*, tel qu'indiqué dans le Tableau 5.2.

Silent Banker. De façon tout aussi surprenante, Aligot n'a pas identifié d'implémentations de TEA dans l'exemplaire de *Silent Banker*, au contraire des autres outils. Plutôt que TEA, notre outil a en effet identifié une implémentation de RTEA, l'algorithme rencontré dans *Storm Worm*. Nous avons confirmé ce résultat par une inspection manuelle de l'exemplaire : les deux familles de logiciels malveillants ont bien fait la même « erreur ». Remarquons que *Silent Banker* a aussi été créé en Russie d'après certaines sources [142]. Le fait que les autres outils identifient une implémentation de TEA s'explique par la visibilité statique de la constante *delta* dans ce programme.

Pour résumer, Aligot a identifié avec succès que, contrairement aux apparences, *Storm Worm* n'implémentait *pas* l'algorithme TEA, mais une version modifiée que nous avons nommée RTEA. De plus, après avoir ajouté une implémentation de référence pour RTEA, Aligot a

automatiquement identifié cette nouvelle fonction cryptographique dans *Silent Banker*, tandis que les autres outils l'ont classifiée à tort comme TEA.

Remarque 21. Notre outil ne montre pas une équivalence générale entre un programme et une fonction, mais l'égalité entre les deux sur une entrée, durant l'exécution observée. Pour être cohérent dans notre comparaison entre outils, il nous faut préciser que :
– Nous considérons que les autres outils répondent eux aussi à la question de l'identification d'une exécution, même si pratiquement tous analysent statiquement le programme au complet.
– Pour vérifier la correction du résultat d'un outil (couleurs vert et rouge du Tableau 5.2), il y a deux cas. Pour les exemples synthétiques, comme nous connaissons l'algorithme implémenté, nous pouvons attester de la validité (ou de l'invalidité) de l'identification sur l'exécution observée. Dans le cas des logiciels malveillants, nous nous basons sur une analyse manuelle a posteriori afin de vérifier l'identification.

5.6.2 RC4

L'algorithme RC4 est un chiffrement à flot utilisant une clé de longueur variable [118]. Un flot de bits pseudo-aléatoire est d'abord généré à l'aide de la clé, pour ensuite appliquer un ou-exclusif entre ce flot et le texte d'entrée, afin de chiffrer ou déchiffrer celui-ci. En particulier, trois comportements itératifs distincts sont présents : deux pour la création du flot pseudo-aléatoire, et un troisième pour l'application du ou-exclusif.

Exemples synthétiques

De la même façon que pour TEA, nous avons d'abord vérifié la correction de notre processus d'identification à l'aide d'exemples synthétiques. Ainsi, nous avons construit les deux programmes suivants :

– $\mathcal{B}_4$: code source original de RC4 [118], compilé avec MVS, sans aucune optimisation (option `/Od`).

– $\mathcal{B}_5$: code source original de RC4, compilé avec MVS, avec une optimisation maximale de la taille (option `/O1`).

Les résultats d'identification sont présentés dans le Tableau 5.3. Aligot est le seul outil qui identifie $\mathcal{B}_4$ et $\mathcal{B}_5$ comme des implémentations de RC4. Ceci s'explique par l'absence de signes syntaxiques particuliers dans cet algorithme – tel que montré dans la thèse de Gröbert [69] – qui rend inutile les outils classiques.

Tableau 5.3 Résultats d'identification sur RC4. Le nom de la fonction identifiée par chaque outil est écrit et ✕ indique une absence d'identification. La couleur indique la correction du résultat : vert pour un résultat correct, et rouge à l'inverse.

	$\mathcal{B}_4$	$\mathcal{B}_5$	Sal_1	Sal_2	Sal_3	Sal_4
Aligot	RC4	RC4	RC4	RC4	RC4	RC4
Draca	✕	✕	✕	✕	✕	✕
FindCrypt2	✕	✕	✕	✕	✕	✕
Hash&Crypto Detector	✕	✕	✕	✕	✕	✕
Kerckhoffs	✕	✕	✕	✕	✕	✕
KANAL	✕	✕	✕	✕	✕	✕
Signsrch	✕	✕	✕	✕	✕	✕
SnD Crypto Scanner	✕	✕	✕	✕	✕	✕

Exemples de protections de logiciels malveillants

Nous avons investigué une famille de logiciel malveillant, nommée *Sality*, pour laquelle des sources publiques mentionnaient l'utilisation de RC4 dans ses couches de protection [155]. Nous avons collecté quatre exemplaires de cette famille sur Internet : Sal_1, Sal_2, Sal_3 et Sal_4. Les résultats de leur identification sont présentés dans le Tableau 5.3. Encore une fois, Aligot est le seul outil capable d'identifier effectivement une implémentation de RC4 dans ces programmes.

Pour insister sur la robustesse de notre méthode, nous présentons dans le Tableau 5.4 une comparaison des implémentations de RC4 identifiées par Aligot. En particulier, nous montrons le nombre de BB ainsi que le nombre d'instructions moyen dans chacun de ces BB (cette comparaison concerne seulement la partie du code reconnue comme une implémentation de RC4 par Aligot). On peut observer une grande variabilité entre les implémentations, ce qui montre la résistance de notre méthode face aux modifications syntaxiques.

Tableau 5.4 Comparaison entre les implémentations de RC4

	$\mathcal{B}_4$	$\mathcal{B}_5$	Sal_1	Sal_2	Sal_3	Sal_4
Nombre de BB	18	12	9	17	4	14
Nombre d'instructions moyen par BB	7	6	40	25	97	29

De plus, grâce aux graphes de paramètres extraits de chaque implémentation de RC4 dans les exemplaires de *Sality*, nous avons pu identifier une répétition dans la façon dont le logiciel

malveillant récupère ses paramètres cryptographiques. Les graphes des exemplaires Sal_1 et Sal_2 sont présentés dans la Figure 5.10. On peut y observer que chaque paramètre cryptographique (clé, texte d'entrée et de sortie) commence à la même position par rapport au début du fichier (`0x542000` pour Sal_1 et `0x410000` pour Sal_2), et ce bien que leurs valeurs soient différentes. Cette information est vraie pour tous les exemplaires de *Sality* que nous avons étudiés, et cela permet donc d'accéder aux paramètres cryptographiques de façon générique. Cela est utile pour implémenter le déchiffrement dans un antivirus, sans exécuter le logiciel malveillant.

5.6.3 Advanced Encryption Standard (AES)

L'algorithme AES est un chiffrement par bloc de 16 octets, avec une clé qui peut être de 128, 256 ou 512 bits [48]. Le texte d'entrée est transformé par un réseau de substitution-permutation où chaque itération (ronde) utilise une clé dérivée de la clé d'entrée. À la différence des algorithmes TEA et RC4, toutes les opérations de AES ne sont pas contenues dans un comportement itératif. En effet, la première et la dernière ronde de l'algorithme sont en dehors de la boucle principale, car elles n'appliquent pas exactement les mêmes opérations que les autres. De ce fait, notre outil ne peut extraire d'une trace d'exécution que les rondes internes de AES (celles à l'intérieur d'une boucle). Ainsi, nous avons ajouté une implémentation des rondes internes de AES dans notre base de référence.

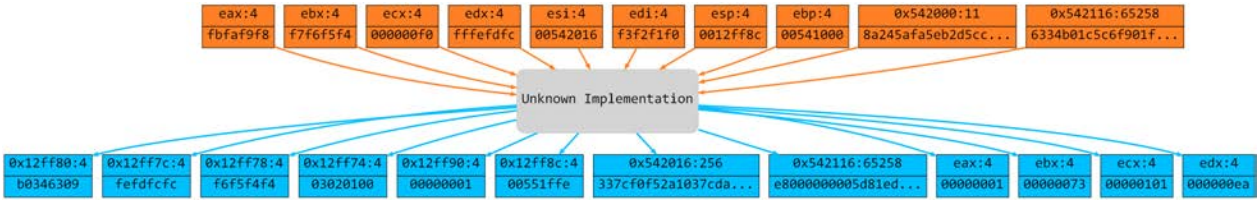
Exemples synthétiques

Afin de vérifier la correction de notre outil pour cet algorithme, nous avons créé les programmes suivants :

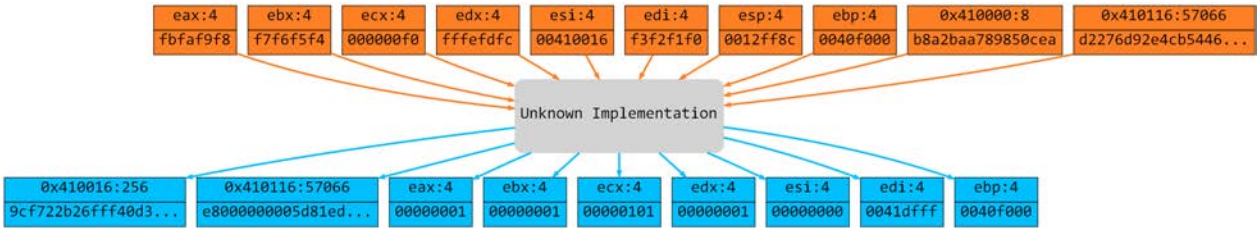
– $\mathcal{B}_6$, qui est un code source basé sur la bibliothèque OpenSSL [73], compilé avec MVS sans aucune optimisation, et dans lequel nous chiffrons un texte en AES-128.

– $\mathcal{B}_7$, qui est le programme $\mathcal{B}_6$ auquel nous avons appliqué la protection commerciale AsProtect [133].

Comme montré dans le Tableau 5.5, la plupart des outils sont capables d'identifier $\mathcal{B}_6$, notamment grâce aux boites de substitution de AES, qui sont initialisées avec des valeurs spéciales, et donc reconnaissables. De son côté, Aligot identifie seulement les rondes internes – pour la raison précédemment évoquée – mais peut en plus conclure sur la taille de la clé, du fait de sa connaissance exacte des paramètres (la taille des clés dérivées dépendant de la taille de la clé initiale). Concernant $\mathcal{B}_7$, seul Aligot est capable d'identifier l'implémentation, les autres outils ne pouvant pas passer à travers la protection qui lui a été appliquée.



(a) Graphe de paramètres extrait de Sal_1, `542000:11` est la clé, `542116:65258` le texte d'entrée et `542116:65258` le texte de sortie.



(b) Graphe de paramètres extrait de Sal_2, `410000:8` est la clé, `410116:57066` le texte d'entrée et `410116:57066` le texte de sortie.

Figure 5.10 Graphes de paramètres pour la partie de la trace reconnue comme une implémentation de RC4 dans Sal_1 et Sal_2. Seuls les 8 premiers octets de la valeur de chaque paramètre sont affichés.

Exemple de logiciel malveillant

Nous n'avons pas trouvé de références de l'algorithme AES utilisé dans une protection de logiciel malveillant. Par contre, cet algorithme est souvent employé pour protéger les communications réseau, donc dans la charge utile camouflée par la protection. Par exemple, nous avons analysé un exemplaire de la famille *Waledac*, dont nous avions montré dans un travail précédent qu'elle utilise AES pour chiffrer ses communications [34]. Tel qu'indiqué dans le Tableau 5.5, Aligot est le seul outil à identifier l'implémentation, les autres ne pouvant pas passer à travers la protection du logiciel malveillant.

5.6.4 MD5

L'algorithme MD5 est une fonction de hachage cryptographique qui calcule une empreinte de 128 bits [117]. Le message d'entrée est découpé en morceaux de 512 bits qui sont ensuite traités de façon itérative. Il y a donc un comportement répétitif au cœur de MD5 mais, comme pour AES, celui-ci ne constitue pas l'algorithme dans son ensemble : une dernière transformation de l'entrée peut avoir lieu après la boucle principale. Nous avons donc ajouté cette boucle principale dans notre base d'implémentations de référence.

Tableau 5.5 Résultats d'identification sur AES. Le nom de la fonction identifiée par chaque outil est écrit et ✕ indique une absence d'identification. La couleur indique la correction du résultat : vert pour un résultat correct, et rouge à l'inverse.

	$\mathcal{B}_6$	$\mathcal{B}_7$	*Waledac*
Aligot	AES-128 (rondes internes)	AES-128 (rondes internes)	AES-128 (rondes internes)
Draca	✕	✕	✕
FindCrypt2	AES	✕	✕
Hash&Crypto Detector	✕	✕	✕
Kerckhoffs	✕	✕	✕
KANAL	AES	✕	✕
Signsrch	AES	✕	✕
SnD Crypto Scanner	AES	✕	✕

Exemples synthétiques

Afin de vérifier la correction de notre outil, nous avons créé les programmes suivants :

– $\mathcal{B}_8$, qui est un code source basé sur la bibliothèque OpenSSL [73], compilé avec MVS sans aucune optimisation, et dans lequel nous calculons l'empreinte d'un texte.

– $\mathcal{B}_9$, qui est le programme $\mathcal{B}_8$ auquel nous avons appliqué la protection commerciale AsProtect [133].

Tableau 5.6 Résultats d'identification sur MD5. Le nom de la fonction identifiée par chaque outil est écrit et ✕ indique une absence d'identification. La couleur indique la correction du résultat : vert pour un résultat correct, et rouge à l'inverse.

	$\mathcal{B}_8$	$\mathcal{B}_9$	*Waledac*
Aligot	MD5 (boucle principale)	MD5 (boucle principale)	MD5 (boucle principale)
Draca	MD5	SHA-1	✕
FindCrypt2	MD4	✕	✕
Hash&Crypto Detector	MD5	SHA-1	✕
Kerckhoffs	✕	✕	✕
KANAL	MD5	✕	✕
Signsrch	✕	✕	✕
SnD Crypto Scanner	MD5	✕	✕

De façon similaire à AES, et comme montré dans le Tableau 5.6, la plupart des outils sont capables d'identifier $\mathcal{B}_8$, notamment parce que MD5 initialise sa structure interne avec 4 constantes spéciales, et donc reconnaissables. De son côté, Aligot identifie seulement la boucle principale, mais il est également le seul à identifier $\mathcal{B}_9$. On peut remarquer plusieurs faux positifs, qui s'expliquent par le fait que les constantes d'initialisation de MD5 sont communes avec d'autres algorithmes cryptographiques, montrant une fois de plus la précarité d'une telle méthode d'identification.

Exemple de logiciel malveillant

Les fonctions de hachage sont habituellement utilisées dans la charge utile des logiciels malveillants, et pas dans leurs protections. Nous avons donc repris l'exemplaire de *Waledac* précédemment étudié dont nous avions observé l'utilisation de MD5 [34]. Tel qu'indiqué dans le Tableau 5.6, Aligot est le seul outil à identifier correctement l'implémentation, les autres ne pouvant pas passer à travers la protection du logiciel malveillant.

5.6.5 Autres

Nous discutons ici de résultats supplémentaires obtenus durant la validation expérimentale d'Aligot. Nous présentons une brève étude de RSA, mais aussi d'une implémentation modifiée de TEA, et finalement des combinaisons entre algorithmes cryptographiques. Il s'agit avant tout d'études préliminaires pour rendre compte de la généralité de notre approche et explorer de futures voies de recherche.

RSA

L'algorithme RSA [119] est un chiffrement asymétrique dont la sécurité est basée sur la difficulté de factoriser des grands nombres. Son opération mathématique principale est *l'exponentiation modulaire*. Celle-ci est habituellement implémentée par une boucle où, lors de chaque itération, un bit de l'exposant est testé et, en fonction de sa valeur, des opérations différentes sont effectuées. Ainsi, cette « boucle » ne réalise pas le même traitement à chaque itération, et ne génère donc pas une trace dans le langage $LOOP$ de la Définition 8.

Par contre, l'opération de *multiplication modulaire* – qui est à la base de l'exponentiation – est habituellement implémentée au sein d'une boucle correspondant bien à notre langage. Remarquons que cette opération n'est pas seulement caractéristique de RSA, mais son identification peut faciliter l'analyse de ce type d'algorithme. Nous avons donc construit un exemple synthétique de chiffrement RSA à l'aide de la bibliothèque PolarSSL [44], dont la multipli-

cation modulaire est faite par l'algorithme de Montgomery [105] (tout comme la majorité des implémentations de RSA). En ajoutant la multiplication modulaire dans notre base de référence, Aligot a pu alors identifier avec succès l'implémentation de cette opération dans notre programme, contrairement aux autres outils.

Un des problèmes rencontrés dans cette brève étude est *l'encodage des grands nombres*. En effet, il y a une différence entre la façon dont ces nombres sont représentés au niveau de la machine, et dans les implémentations de haut niveau. Pour remédier à cela, nous avons dû ajouter une procédure de décodage qui fournit la représentation haut niveau des valeurs collectées dans la trace d'exécution. Comme il existe en pratique peu d'encodages différents, cela reste réaliste d'ajouter toutes ces routines avant de tester la relation entrée-sortie. Par contre, un programmeur pourrait créer son propre encodage, afin de désynchroniser à escient la relation entrée-sortie entre implémentations d'un même algorithme.

TEA modifié

Nous avons considéré la possibilité que des auteurs de logiciels malveillants modifient la valeur de la constante *delta* d'une implémentation de TEA – ou d'un quelconque algorithme utilisant des constantes spéciales – dans le but de devenir indétectable par les outils basés sur sa valeur habituelle. En effet, tel qu'observé avec les familles *Storm Worm* et *Silent Banker*, les auteurs de logiciels malveillants se soucient peu de la qualité cryptographique de leurs algorithmes, et sont plutôt concernés par les signes qui permettent d'identifier leurs créations. Pour identifier ces modifications, nous avons créé une implémentation de référence de TEA où *delta* n'est plus une constante interne à l'algorithme, mais un paramètre. Nous avons alors construit un exemple d'un chiffrement TEA avec une constante différente. Lors de l'analyse, Aligot a collecté la nouvelle valeur de *delta* dans la trace d'exécution, pour ensuite la positionner dans le paramètre correspondant, et ainsi il a identifié l'implémentation comme une modification de TEA. Ceci s'est fait de façon automatique et Aligot est le seul outil à réussir l'identification.

Combinaisons d'algorithmes

Comme expliqué en §5.4.3, Aligot teste en pratique tous les sous-graphes d'un BFD, afin d'identifier les algorithmes qui auraient été composés et qui seraient donc reliés par un flux de données. Pour valider cela, nous avons créé un exemple de *Double TEA* (deux applications de l'algorithme d'affilée), ainsi qu'un masque jetable suivi d'un RC4. Tel que prévu, Aligot a pu identifier avec succès chacune des implémentations, et également montrer qu'elles étaient

combinées.

5.7 Performances

Nous présentons dans le Tableau 5.7 les performances en temps de l'outil, mesurées lors de l'analyse des exemplaires de logiciels malveillants sur une machine utilisateur standard. Les performances sur les exemples synthétiques ne sont pas indiquées, car elles sont toutes en dessous de 20 minutes au total.

Ces résultats doivent être considérés comme purement indicatifs, puisqu'Aligot a été conçu avant tout comme une preuve de concept de la méthode d'identification décrite dans ce chapitre, et pas pour être un outil efficace. Observons tout de même que la phase de comparaison ne nécessite que 30 minutes au maximum, bien que sa complexité théorique soit exponentielle (cf. §5.5). Ceci s'explique par l'efficacité de nos heuristiques pour réduire le nombre de paramètres à recombiner, par exemple en ignorant ceux qui semblent être des adresses mémoires. Remarquons que l'outil réintroduit ces paramètres ignorés s'il n'arrive pas à identifier d'algorithme. Ensuite, la détection des boucles est la tâche la plus coûteuse, car comme nous l'avons expliqué en §5.4.1, sa performance dépend du nombre de boucles dans la trace. Si les boucles – selon notre définition – sont rares, alors la taille de l'historique grandit et les performances diminuent (car l'algorithme doit parcourir tout l'historique pour chaque instruction). À l'inverse, lorsqu'il y a beaucoup de boucles, la taille de l'historique reste limitée et le traitement de chaque instruction est plus rapide.

5.8 Limitations

Au-delà de la limitation évidente de l'analyse dynamique aux chemins d'exécution observés, il convient d'insister sur différentes restrictions. Premièrement, et malgré nos résultats expérimentaux très positifs, il serait illusoire de penser que notre modèle de BFD capture toutes

Tableau 5.7 Performance en temps d'Aligot. ID signifie "Instruction Dynamique"

	$Storm$	$SBank$	Sal_1	Sal_2	Sal_3	Sal_4	Wal
Taille de la trace (ID)	3M	3.5M	4.1M	1M	4.8M	4.2M	20k
Traçage (min)	4	3	2	1	2	2	1
Construction BFD	4hr	6hr	10hr	4hr	10hr	15hr	40mn
Comparaison (min)	30	30	3	3	4	4	10
Temps total	4,5hr	6,5hr	10hr	4hr	10hr	15hr	51mn

les implémentations possibles de cryptographie. En effet, les programmes en langage machine x86 n'ayant que très peu de normes à respecter pour être exécutables, toute abstraction sur ces programmes n'est qu'une *convention*. Comme toute convention, elle peut ne pas être respectée par certains programmes (en particulier quand ceux-ci y ont intérêt pour complexifier leur analyse). Par exemple, des auteurs de logiciels malveillants pourraient simplement choisir d'implémenter des comportements itératifs avec des boucles qui ne respectent pas notre définition. En réponse à cela, nous pourrions alors modifier notre abstraction. De même, les paramètres cryptographiques pourraient être encodés d'une façon particulière, qui ne correspondrait pas à celle de nos implémentations de haut niveau. Notre réponse serait alors d'ajouter une nouvelle routine de décodage. Ainsi, la liberté que fournit la programmation au niveau machine, et l'aspect offensif des logiciels malveillants, empêchent l'existence d'une solution générique à notre problème d'identification cryptographique. Tout en acceptant cette limitation, la construction d'une solution dans un contexte spécifique demeure très utile pour l'analyste.

Deuxièmement, notre outil n'a la capacité d'identifier que ce qu'il connaît, c'est-à-dire les fonctions pour lesquelles nous possédons une implémentation de référence. Néanmoins, il est permis de pouvoir ajouter facilement de nouvelles implémentations, et c'est ce que nous avons fait dans le cas de RTEA en §5.6.1.

5.9 Voies de recherche futures

Notre expérience avec RTEA, c'est-à-dire l'ajout d'un algorithme cryptographique « maison » dans notre base de référence, afin d'être ensuite en mesure de l'identifier dans d'autres programmes, a été particulièrement inspirante. Il serait intéressant d'importer automatiquement dans notre base de référence tous les possibles algorithmes cryptographiques rencontrés dans les traces d'exécution, même si on échoue à les identifier, afin de pouvoir ensuite les reconnaître s'ils sont réutilisés. Notre outil contient déjà le nécessaire pour extraire les algorithmes, le problème est ici de reconstruire un programme et de l'exécuter sur de nouveaux arguments à partir d'une trace d'exécution, ce qui a déjà été abordé dans d'autres travaux présentés au chapitre 3 [24, 86, 54].

Le principe de notre comparaison entrée-sortie peut être utilisé pour n'importe quel algorithme déterministe. Par exemple, il est courant de rencontrer des algorithmes de compression dans les protections de logiciels malveillants. Il s'agirait donc ici d'appliquer la même méthode, mais avec une abstraction adaptée à ce type d'algorithme.

Une question de recherche plus large est celle de la détermination d'une *bonne* abstraction sur le code d'un programme. En effet, le fait d'avoir une abstraction, c'est-à-dire de regrouper un ensemble d'instructions dans une structure commune, facilite la compréhension humaine et permet de développer des raisonnements complexes, comme celui qui nous a permis de partir des boucles pour identifier des fonctions cryptographiques. Sachant qu'un logiciel malveillant obfusqué aura tendance à ne pas implémenter les abstractions usuelles (comme les fonctions ou les boucles naturelles), peut-on déterminer automatiquement s'il existe une abstraction équivalente ? À quels langages les abstractions intéressantes appartiennent-elles ? Nous soupçonnons que dans de nombreux cas de programmes obfusqués, il existe dans le code des structures simples, en fonction de la transformation de programme qui les a créés.

5.10 Conclusion

Dans ce chapitre nous avons présenté une méthode pour l'identification de fonctions cryptographiques dans des programmes obfusqués. Pour ce faire, nous avons introduit une abstraction nommée *Boucles avec Flux de Données* pour extraire le code possiblement cryptographique d'une trace d'exécution. Nous avons pu alors définir les paramètres de ce code, et collecter ses valeurs. Finalement, nous avons comparé celles-ci avec des fonctions cryptographiques de référence. Ce processus a été implémenté dans un outil nommé Aligot, que nous avons validé expérimentalement sur des implémentations d'algorithmes tels que TEA, RC4, MD5, AES et RSA. Ainsi, même s'il ne s'agit que d'une preuve de concept évaluée sur un nombre restreint d'implémentations, Aligot ouvre une voie de recherche prometteuse pour aider à l'analyse des protections de logiciels malveillants.

Deuxième partie

RÉSEAUX DE MACHINES ZOMBIES

CHAPITRE 6

ENVIRONNEMENT D'EXPÉRIENCE

Afin d'étudier les réseaux de machines zombies, nous avons adopté l'émulation, une approche qui consiste essentiellement à observer de véritables machines infectées dans des conditions aussi proches que possible de la réalité. Cela permet d'atteindre un niveau de réalisme élevé, mais pose de nombreux défis pratiques, que nous allons décrire dans ce chapitre.
Nous commencerons par introduire le problème de l'étude des réseaux de machines zombies, ainsi que les approches existantes, pour ensuite décrire en détail notre solution. Des parties de ce chapitre ont été publiées dans [35, 36, 37].

6.1 Problème de recherche

Un réseau de machines zombies – un *botnet* – est constitué de machines infectées par un logiciel malveillant – des *bots* – et sous le contrôle d'une même entité distante – un *Command & Control (CC)*. Une même famille de logiciel malveillant peut donner lieu à des *botnets* distincts, chacun étant alors contrôlé par une entité différente. Ce contrôle à distance permet aux auteurs de logiciels malveillants de maximiser la rentabilité des machines infectées. Ainsi, il n'est pas un utilisateur qui n'ait expérimenté un des effets des *botnets* sur Internet, que ça soit par la réception de pourriels, le vol d'informations bancaires, ou encore l'impossibilité d'accéder à un site soumis à une attaque de déni de service.

Les *botnets* d'aujourd'hui comprennent plusieurs milliers de machines ; par exemple *Torpig* a été estimé à plus de 180 000 ordinateurs [136], et *Waledac* à plus de 390 000 [135]. Ce développement en nombre s'est agrémenté d'une évolution technologique : s'il fut une époque où les *botnets* étaient contrôlés par de simples serveurs IRC, certains reposent aujourd'hui sur des architectures réseau complètement décentralisées, avec des protocoles construits pour l'occasion, les rendant de ce fait particulièrement résistants aux attaques qui visent leur infrastructure réseau [63]. Afin de contrer efficacement cette menace, la recherche en sécurité informatique se doit alors d'étudier les *botnets* de façon *globale*, c'est-à-dire en prenant en compte le réseau dans son ensemble. En effet, l'analyse *locale* d'une machine infectée – par exemple par la rétro-ingénierie du logiciel malveillant [33, 64] – ne permet pas de comprendre un *botnet* complètement, puisque son architecture réseau est alors difficile à établir. De ce fait, *l'analyse des botnets par une approche globale* constitue le problème de recherche que

nous allons traiter dans cette seconde partie de la thèse.

6.2 Travaux existants

Il existe trois grandes façons d'aborder l'analyse des *botnets* de façon globale : les étudier directement sur Internet, construire un modèle théorique et le simuler, ou bien encore créer une copie du réseau dans un environnement d'expérience isolé. Nous ne parlerons pas ici des travaux sur la détection des *botnets*, car ils ne répondent pas directement à notre problème de recherche.

Étude dans la nature

Il s'agit ici d'interagir directement avec des *botnets* en activité. Ainsi, Stone-Gross *et al.* [136] ont pris le contrôle du réseau de machines zombies *Torpig*. Pour ce faire, ils ont utilisé le fait que le *botnet* changeait automatiquement le nom de domaine servant de point de contact avec les machines infectées. Les auteurs ont réussi à enregistrer l'un de ces domaines avant les véritables contrôleurs du réseau. Grâce à cela, ils ont administré le réseau pendant une dizaine de jours, avant que les machines infectées ne soient mises à jour avec un autre nom de domaine. Cela leur a permis d'estimer précisément la taille du réseau, à plus de 180 000 *bots*, et d'étudier les informations collectées par le CC. Une telle prise de contrôle reste rare, par contre il existe de nombreux exemples d'infiltrations de *botnets* réalisées par des équipes de recherche académiques. Dans ce cas il s'agit de surveiller, ou de reproduire, une machine infectée afin d'observer les ordres qu'elle reçoit. Par exemple, Caballero *et al.* [25] ont infiltré le réseau *MegaD* et ont pu collecter des informations sur les pourriels envoyés. De leur côté, Stock *et al.* [135] ont construit un clone d'une machine infectée par *Waledac* afin de participer aux échanges de listes de pairs dans ce réseau pair-à-pair. Grâce à cela, ils ont pu estimer le nombre de *bots* à plus de 390 000. Plus récemment, Rossow *et al.* [120] ont appliqué différentes méthodes d'estimation de taille à 11 *botnets* en activité. Ils ont ainsi montré que l'injection de senseurs était une méthode plus précise que la reconstruction active du graphe – en interrogeant les nœuds –, car bien souvent une grande partie des machines n'est pas joignable de l'extérieur. Ils ont également implémenté plusieurs attaques directement sur Internet, par exemple la redirection des *bots* vers une machine sous leur contrôle, afin de mesurer de façon pratique la résistance de ces *botnets*.

Cette première approche à l'avantage de collecter des informations permettant de mieux comprendre les *botnets* et les objectifs de leurs contrôleurs. D'un autre côté, il y a plusieurs

inconvénients :

- **L'absence de furtivité :** l'expérimentation avec un *botnet* en activité implique habituellement une participation au réseau, qui pourrait être remarquée par ses contrôleurs. Ceux-ci pourraient alors décider de changer leur tactique, rendant de ce fait la recherche plus difficile. À l'inverse, une participation trop timide au *botnet* risque de ne pas amener d'informations intéressantes.
- **La non-reproductibilité des observations :** il est difficile d'obtenir des résultats statistiquement significatifs par cette approche, car il y a peu de chance de pouvoir expérimenter plusieurs fois. Le risque est alors d'observer un cas particulier sans s'en rendre compte, et il n'est pas de plus possible d'explorer différentes configurations. Finalement, certaines variables expérimentales ne sont pas sous contrôle, ou même mesurables, du fait de la vision partielle du *botnet*.
- **Le manque d'éthique :** la participation à un *botnet* pose au moins deux problèmes d'éthique importants : premièrement des utilisateurs de machines infectées pourraient être impactés par la recherche, sans avoir donné leur consentement, et ensuite les chercheurs pourraient aider à la réalisation d'activités criminelles. Il s'agit à notre connaissance d'une question ouverte, notamment pour des équipes de recherche académiques, dont la légitimité pour mener ce genre d'action peut paraître discutable.

Modèle formel et simulation

Dans cette approche un modèle théorique de *botnet* est construit afin d'être simulé pour réaliser diverses expériences. Dagon *et al.* [49] ont employé des graphes pour simuler différentes architectures de réseaux de machines zombies, afin de comparer leurs résistances respectives face à des attaques. Les auteurs montrent que les architectures pair-à-pair sont les plus robustes contre les désinfections. Dans le même ordre d'idée, Davis *et al.* [51, 50] ont simulé des réseaux pair-à-pair pour tester des attaques, avec d'autres métriques de performance. Ils en ont conclu qu'un réseau pair-à-pair *structuré* – dans lequel la liste de pairs suit une organisation particulière – résiste mieux face à des désinfections ciblant les nœuds les plus connectés. De leur côté, Van Ruitenbeek *et al.* [141] ont construit un modèle stochastique d'un *botnet* utilisant un réseau pair-à-pair structuré, toujours afin de simuler sa résistance. Ils ont pu en déduire le nombre de machines à désinfecter pour avoir un impact sur l'utilisation d'un *botnet*. Finalement, Rossow *et al.* [120] n'ont pas seulement lancé des attaques sur Internet, tel que décrit précédemment, mais ils ont également créé un modèle formel basé sur la théorie des graphes pour les définir. Dans ce cas l'objectif du modèle est surtout pédagogique.

La simulation a l'avantage de permettre une reproduction facile des expériences, et donc l'obtention de résultats statistiquement significatifs, ainsi que l'exploration de différentes configurations. Elle permet également d'expérimenter avec des réseaux de plusieurs milliers de nœuds, tout en restant invisible des contrôleurs du vrai *botnet*, et en ne mettant pas en danger des utilisateurs. D'un autre côté, les *botnets* sont des objets délicats à modéliser, en particulier parce qu'ils emploient souvent des protocoles réseau construits pour l'occasion pour lesquels aucune description de haut niveau n'est disponible, seulement une implémentation [34, 63, 25]. Plus généralement, la logique interne des logiciels malveillants est difficile à simuler fidèlement, puisqu'il s'agit de programmes en langage machine interagissant avec un système d'exploitation et un réseau. S'ajoute à cela le fait que le comportement des utilisateurs rend la topologie réseau très dynamique, par exemple avec le cycle d'extinction-allumage des machines, ou celui d'infection-désinfection, ce qui est difficile à inclure dans un modèle. Ainsi, la simulation fidèle de *botnets* est une tâche ardue, et leur complexité grandissante n'aide pas.

Émulation

Une troisième approche consiste à reproduire un *botnet* au complet avec de véritables machines dans un environnement contrôlé, ce que nous appelons *émulation*. Ce type d'approche expérimentale est couramment utilisée dans le domaine de la recherche en réseau, en particulier avec la plate-forme Emulab [57] qui permet de déployer des expériences sur un ensemble de clusters répartis autour du monde. En ce qui concerne l'émulation de *botnets*, Barford *et al.* [8] ont construit sur la plate-forme Emulab un environnement de test spécifique aux logiciels malveillants. Ils mettent notamment à disposition des images de machines virtuelles infectées, afin d'expérimenter. Leur plate-forme comporte de nombreuses limitations pratiques, par exemple le fait que tous les nœuds doivent avoir seulement des adresses IP locales afin de ne pas être contactés de l'extérieur. De plus, leur environnement n'a été testé qu'avec une centaine de *bots*, ce qui est loin d'être une taille suffisante pour une expérience réaliste. La plate-forme DETER se veut être l'équivalent d'Emulab pour la recherche en sécurité informatique [104]. Elle relie sous une interface commune deux clusters académiques, accessibles à distance pour diriger des expériences. Jackson *et al.* [81] ont utilisé DETER pour construire un système d'investigation de *botnets* permettant aux chercheurs de déployer des logiciels malveillants *fictifs*, afin d'observer le trafic réseau. C'est à notre connaissance le seul projet d'émulation de *botnet* qui ait été réalisé avec DETER.

L'émulation possède plusieurs avantages, notamment par rapport aux approches précédentes :

1. **Reproductibilité :** les expériences peuvent être reproduites à volonté, jusqu'à obtenir des résultats statistiquement significatifs, et de façon à explorer différentes configurations.
2. **Contrôle :** un grand nombre de variables expérimentales peuvent être directement contrôlées ou, à défaut, mesurées, puisque l'environnement d'expérience au complet est accessible au chercheur. Cela permet en particulier d'observer des phénomènes invisibles sur Internet.
3. **Confidentialité :** le fait de travailler en isolement permet d'explorer différents paramètres expérimentaux sans risque de prévenir les contrôleurs du véritable *botnet*. Ceci est particulièrement important pour le développement d'attaques.
4. **Éthique :** aucun véritable utilisateur n'est impacté par l'émulation, et les chercheurs ne participent pas à des activités criminelles, ce qui rend cette méthode éthiquement acceptable.
5. **Réalisme :** l'utilisation de véritables machines infectées, au contraire d'un modèle formel, permet de résoudre de nombreux problèmes de réalisme, comme le comportement du logiciel malveillant, du système d'exploitation, etc.

Les avantages 1 à 4 sont partagés avec la simulation, mais l'avantage 5 permet de dépasser (en grande partie) le principal défaut de cette approche. Par contre, le problème de la simulation du comportement de l'utilisateur se pose aussi avec l'émulation. De plus, l'émulation de *botnets* est coûteuse, puisqu'elle nécessite un réseau avec de véritables machines. Finalement, il existe peu d'outils pour diriger ce genre d'expériences, dont l'objectif n'est pas du traitement parallélisé de données, mais la reconstruction d'un réseau hétérogène dont les nœuds communiquent intensément entre eux.

6.3 Solution proposée

L'émulation nous est apparue comme la seule voie offrant une rigueur scientifique et un niveau d'éthique acceptable pour étudier les *botnets* de façon globale. Ainsi, notre solution est de construire un environnement d'expérience dans lequel un *botnet* puisse être déployé. Nous avons bénéficié pour ce projet de ressources matérielles et logicielles importantes, qui nous ont permis de développer une approche plus adaptée à l'émulation de *botnets* que les travaux existants, notamment par l'utilisation de véritables logiciels malveillants à une échelle relativement grande. Nous commencerons ici par décrire les critères de conception de notre solution, puis l'environnement de laboratoire lui-même, pour ensuite parler de la méthodologie expérimentale et des défis. Dans la suite des explications, nous supposerons que le

botnet à émuler possède un CC, mais notre approche s'étend naturellement à des réseaux complètement décentralisés.

6.3.1 Critères de conception

Dans le but de pouvoir expérimenter avec les *botnets*, nous avons défini des critères à respecter :

- **Réalisme.** Étant donné la difficulté de modéliser les *botnets*, nous avons fait le choix d'expérimenter avec de véritables logiciels malveillants. Afin de rester fidèles, ces programmes exécutables ne doivent être modifiés qu'au minimum – idéalement pas du tout –, par exemple pour assurer leur exécution en enlevant les couches de protection contenant des tests sur l'environnement (anti-virtualisation, etc.). De plus, il faut s'assurer que la prise des mesures durant l'expérience ne modifie pas le comportement du *botnet*.
- **Sécurité.** Comme nous utilisons de véritables logiciels malveillants, et qu'ils ont parfois tendance à se propager par eux-mêmes, il faut mettre en place des mesures de confinement aux niveaux physique et logique, afin d'empêcher une propagation incontrôlée.
- **Échelle.** Afin de réaliser des expériences réalistes, il est nécessaire de pouvoir observer un nombre conséquent de machines.
- **Flexibilité.** Comme tous les *botnets* ne se ressemblent pas, il est nécessaire que notre environnement supporte de multiples topologies réseau avec différentes proportions d'adresses IP privées ou publiques. Il doit aussi permettre au chercheur de définir les services habituels d'Internet (serveurs DNS, SMTP, HTTP, etc.).
- **Stérilisation.** Afin de garantir l'intégrité des expériences, les machines doivent pouvoir être ramenées dans un état initial non infecté.

6.3.2 Matériel et logiciels

Notre environnement d'émulation de *botnets* a été développé conjointement dans le cluster du laboratoire SecSI à l'École Polytechnique de Montréal [41], et celui du Laboratoire de Haute Sécurité au LORIA [94]. L'implémentation finale a été effectuée dans le laboratoire de Montréal. L'accès physique à ce cluster est contrôlé par un système d'authentification multi facteurs, et il est complètement déconnecté d'Internet, ce qui satisfait parfaitement notre **critère de sécurité**.

Matériel. Le cluster est constitué de 98 machines physiques, chacune ayant un processeur quatre cœurs et 8 Go de mémoire vive, avec une carte réseau de 4 ports gigabit Ethernet.

Deux réseaux séparés relient les machines physiques entre elles : un réseau de contrôle, pour transmettre les commandes et les données nécessaires au contrôle des expériences, et un réseau d'expérience, utilisé pour le trafic du *botnet* dans notre cas. L'intérêt d'avoir deux réseaux séparés est de s'assurer que le contrôle de l'expérience n'influe pas sur les résultats, ce qui satisfait notre **critère de réalisme**.

Logiciels. Déployer une expérience sur notre cluster a nécessité des outils adaptés :

– **Virtualisation.** Afin de maximiser l'utilisation des machines physiques, et de pouvoir faire des expériences à une échelle raisonnable – pour satisfaire notre **critère d'échelle** – nous avons installé *VMware ESX* [144] sur les machines physiques afin de pouvoir y déployer des machines virtuelles facilement.

– **Configuration et administration.** À la base de notre système de gestion d'expérience se trouve l'outil *Extreme Cloud Administration Toolkit (xCAT)* [153], qui a été adapté au déploiement à grande échelle de machines virtuelles *VMWare*. De façon sommaire, xCAT prend en entrée une table remplie par l'utilisateur, qui indique les images virtuelles devant être déployées sur chaque machine physique, ainsi que leurs configurations réseau. Nous avons construit différentes couches d'abstraction au-dessus de cet outil, notamment afin d'automatiser le démarrage et l'arrêt d'une expérience, et satisfaire nos **critères de flexibilité et stérilisation**. Cette étape a été réalisée par le stagiaire Antonin Auroy à l'été 2011.

– **Capture du trafic réseau.** De nombreuses équipes de recherche s'intéressent à la détection des *botnets* à partir du trafic réseau, et un de leurs problèmes est le faible nombre de jeux de test disponibles. Notre environnement d'expérience est une occasion d'en générer, et pour ce faire nous avons mis en place un système de capture du trafic réseau à différents endroits du cluster.

6.3.3 Méthodologie expérimentale

Afin de conduire des expériences d'émulation de *botnets*, nous avons suivi la procédure expérimentale suivante :

1. Récupération d'un exemplaire du logiciel malveillant que l'on veut étudier. Il faut s'assurer que ce programme peut s'exécuter correctement dans un environnement virtualisé.
2. Détermination de *(i)* l'architecture du *botnet* et son protocole de communication, et *(ii)* des services nécessaires à son bon fonctionnement, comme les serveurs DNS et

SMTP. Cette étape peut être complexe, notamment car le CC n'est habituellement pas accessible à l'analyste, et qu'il faut donc déterminer sa logique seulement à partir des réponses qu'il envoie aux *bots*.

3. Reproduction dans le cluster de l'infrastructure observée lors de l'étape 2. Cela implique de construire les images des machines virtuelles nécessaires (*bots*, serveurs, etc.). De plus, il faut reconstruire un CC capable de communiquer selon le protocole du *botnet*.
4. Mise en place de métriques pour mesurer l'activité du *botnet* dans notre cluster, en fonction de la question de recherche.

6.3.4 Défis

Parmi les nombreux défis rendant difficile l'émulation de *botnets*, il faut insister sur les suivants :

– **La charge de l'environnement.** Le fait d'avoir plusieurs machines virtuelles sur une même machine physique peut entraîner des lenteurs qu'un véritable *bot* ne connaîtrait pas, et qui pourraient fausser les résultats de l'expérience. Il faut donc tester avec soin la charge des nœuds de notre cluster, pour s'assurer que chaque machine virtuelle fonctionne de façon normale. Dans notre cas, nous avons établi empiriquement une limite de 30 machines virtuelles par nœud pour un *bot* avec le système d'exploitation Windows XP, ce qui fait presque 3 000 machines virtuelles au total. De plus, les services utilisés par le réseau (DNS, SMTP, etc.) doivent pouvoir répondre à la demande simultanée de tous les *bots*.

– **Le démarrage du botnet.** Le lancement du *botnet* a un fort impact sur le reste de l'expérience. Par exemple, si toutes les machines démarrent en même temps – ce qui est improbable dans le véritable *botnet* – cela entraînera une charge artificielle sur le CC qui peut fausser les observations. À notre connaissance, il n'existe pas de données empiriques sur le commencement d'un *botnet*. Néanmoins, une majorité d'entre eux utilisent de nos jours des systèmes de *rémunération contre installation*, tel que celui décrit en Figure 1.1. Nous avons alors établit un modèle d'évolution de population « par vague », où un premier groupe de 100 *bots* rejoint le réseau pendant une courte période de temps, puis nous attendons une durée aléatoire entre 0 et 5 minutes, pour ensuite démarrer un second groupe de 100, et ainsi de suite jusqu'à ce que toutes les machines soient démarrées. Nous verrons que cette stratégie est encore loin d'être parfaite.

– **La simulation du comportement des utilisateurs.** Dans un *botnet* le comportement des utilisateurs a un impact sur la population du réseau. Tout d'abord, il y a un phénomène

de *diurnal behavior*, c'est-à-dire des variations de populations en fonction de l'heure de la journée (plus d'utilisateurs en journée qu'au milieu de la nuit). Ensuite, dans une population de taille donnée, l'allumage et l'extinction des machines font que de nouveaux *bots* joignent le réseau, tandis que d'autres le quittent. Finalement, certains utilisateurs peuvent être désinfectés, ce qui les fait sortir du réseau, tandis que des nouveaux sont infectés. Nous n'avons pas trouvé de modèles satisfaisants pour simuler ces trois comportements. Nous reviendrons sur ce problème en §7.7.

6.4 Conclusion

Dans ce chapitre, nous avons décrit les différentes méthodes d'étude des réseaux de machines zombies de façon globale. En particulier, nous avons montré que l'émulation permet de combiner rigueur scientifique et éthique. Puis, nous avons introduit l'environnement d'émulation de *botnets* que nous avons construit. Grâce à celui-ci, nous sommes en mesure de reproduire un réseau de machines zombies à une échelle raisonnable (environ 3 000 *bots*) dans un environnement d'expérience contrôlé.

CHAPITRE 7

CAS D'ÉTUDE : WALEDAC

Nous présentons dans ce chapitre un cas d'étude d'émulation d'un *botnet* dans l'environnement d'expérience présenté précédemment. Ce cas est celui du logiciel malveillant *Waledac*, qui a la particularité de mettre en place des communications pair-à-pair entre ses *bots*. Nous avions conjecturé que ces communications étaient vulnérables dans un travail précédent, et qu'il était possible de prendre le contrôle du *botnet* [34]. Nous avons pu tester cette hypothèse grâce à notre environnement.
Dans un premier temps, nous présenterons le contexte historique de ce travail, puis nous introduirons *Waledac* et son *botnet*, pour ensuite décrire la mise en place de l'expérience, et finalement les résultats. Des parties de ce chapitre ont été publiées dans [32, 34, 36, 37].

7.1 Contexte historique

Afin de comprendre dans quel contexte s'est déroulée la recherche de ce chapitre, nous présentons brièvement les événements qui ont conduit à la réalisation de notre expérience :

1. **Printemps-Été 2009 :** durant un stage au sein du laboratoire de sécurité de l'École Polytechnique de Montréal, nous réalisons une étude en profondeur de *Waledac*, en collaboration avec Pierre-Marc Bureau de la compagnie antivirus ESET. Nous mettons notamment en place une infiltration du *botnet* afin de suivre les commandes distribuées. Ce travail débouche sur la découverte (comme le font conjointement d'autres groupes de recherche) d'une vulnérabilité dans le protocole de communication du réseau, qui sera publiée dans [32, 34].

2. **Automne 2009 :** nous entamons une collaboration avec d'autres équipes de recherche qui s'intéressent à *Waledac*. Nous fournissons notamment toutes les informations nécessaires à l'implémentation de l'attaque. Ce groupe de travail met en place un suivi poussé du réseau, et discute la faisabilité d'une action offensive.

3. **Février 2010 :**
   – Nous commençons la construction d'une expérience pour valider dans notre laboratoire la faisabilité de l'attaque, les premiers résultats – positifs – sont récupérés en mars 2010.

– Microsoft et des partenaires lancent une attaque contre le *botnet*, qui conduira à son abandon par ses opérateurs à la fin du mois. Nous ne sommes malheureusement pas associés à cette initiative, mais nous avons pu observer précisément les événements grâce à notre infiltration du *botnet*. Du point de vue technique, l'attaque est exactement la même que celle que nous étions en train de construire.

Notre expérience a donc été conçue pour explorer la faisabilité d'une attaque contre *Waledac*, dans l'objectif de fournir un maximum d'informations pour sa réalisation sur Internet. De ce fait, il est regrettable que l'initiative de Microsoft n'ait pas bénéficié de nos résultats. À notre connaissance, les mesures réalisées durant la véritable attaque (s'il y en a eu) n'ont jamais été publiées, et il est donc impossible de comparer avec notre expérience.

7.2 Fondations

Dans cette partie nous introduisons *Waledac*, en particulier à partir des résultats de notre propre analyse réalisée en 2009 [32, 34].

7.2.1 Contexte

Historique. *Waledac* est apparu sur Internet en novembre 2008 et il a rapidement attiré l'attention des chercheurs par sa ressemblance – au niveau de son infrastructure réseau et de sa méthode de propagation – avec *Storm*, un logiciel malveillant célèbre pour avoir été le premier à déployer un réseau pair-à-pair entre ses *bots*, qui venait alors de s'éteindre. *Waledac* a été l'objet de nombreuses mises à jour, avec des numéros de version indiqués dans chaque exemplaire, mais son protocole réseau est resté le même à partir de début 2009. En février 2010, Microsoft a lancé un assaut d'envergure contre *Waledac*, en combinant l'attaque technique présentée dans ce chapitre avec une action en justice [101]. Suite à cela, le *botnet* a été complètement stoppé, et n'est pas réapparu depuis. Pour faciliter la lecture (et l'écriture) de ce chapitre, nous emploierons tout de même le temps présent pour en parler.

Taille. Le nombre de systèmes ayant été infectés par *Waledac* est difficile à estimer précisément, et sa population a varié significativement plusieurs fois au cours de son existence. Par exemple, elle a augmenté de façon conséquente lorsqu'un autre logiciel malveillant, *Conficker*, l'a déployé sur une partie de ses propres *bots* [68]. Parmi les différentes estimations, celle de Stock *et al.* [135] indique que 55 000 machines étaient actives par jour durant le mois d'août 2009, et plus de 390 000 en tout durant la même période.

7.2.2 Caractéristiques techniques

Moyens de propagation

À son commencement, le mécanisme de propagation de *Waledac* fut l'ingénierie sociale par courriel, afin de convaincre des internautes de télécharger le programme et de l'exécuter. Les thèmes de ces campagnes évoluaient rapidement en suivant l'actualité, avec par exemple des cartes de souhaits pour la période de Noël, ou encore l'annonce de la démission de Barack Obama deux semaines après son élection.

Mais un autre moyen de propagation a été mis en place début 2009 : l'installation par d'autres familles de logiciels malveillants, contre rémunération. À cet effet, les opérateurs de *Waledac* se sont dotés d'un mécanisme pour comptabiliser les installations des autres groupes : lors de la première connexion au réseau, chaque nouveau *bot* communique un « label » permettant d'identifier celui qui a réalisé l'infection de la machine (et donc de le rémunérer). Par exemple, le label `twist` a été utilisé pour les installations par le logiciel malveillant *Conficker* et `dmitry777` pour celles par *Bredolab*. De plus, des règles contractuelles ont été définies, notamment pour fixer les rémunérations, dont un extrait est montré dans la Figure 1.1.

Suite à notre infiltration du *botnet*, nous avons observé le trafic du *botnet* – malgré l'utilisation de cryptographie, comme nous le décrirons en §7.5.2 – et nous avons comptabilisé que les machines infectées par des « partenaires » représentaient 98% de la population du *botnet* de mai à juillet 2009, le reste étant lié aux campagnes d'ingénierie sociale.

Charges utiles

Waledac possède plusieurs moyens de monétiser les machines infectées :

– **Pourriels.** L'activité principale du *botnet* est d'envoyer des courriels indésirables pour d'autres groupes, contre rémunération. Pour ce faire, le programme embarque un moteur SMTP capable d'envoyer des courriels et, afin d'être plus efficace, des modèles de messages sont distribués aux machines par le biais du réseau. Un exemple de modèle est donné dans la Figure 7.1.

  On y observe les différents champs d'un courriel, à remplir avec des macros, par exemple `%^F` (*fill*) suivie d'un mot pour une valeur à choisir dans un ensemble distribué aux *bots* par le réseau (ce qui permet de générer des courriels avec différents noms d'émetteurs, sujets, etc.), ou encore `%^R` (*random*) suivi d'un intervalle pour une valeur aléatoire.

```
Received: from %^C0%^P%^R3-6^%:qwertyuiopasdfghjklzxcvbnm^%^%
          by %^A^% with Microsoft SMTPSVC(%^Fsvcver^%);
Message-ID: <%^Z^%.%^R1-9^%0%^R0-9^%0%^R0-9^%0%^R0-9^%@%^C1%^Fdomains^%^%>
Date: %^D^%
From: "%^Fmynames^% %^Fsurnames^%" <%^Fnames^%@%^V1^%>
User-Agent: Thunderbird %^Ftrunver^%
MIME-Version: 1.0
To: %^0^%
Subject: %^Fjuly^%
Content-Type: text/plain; charset=ISO-8859-1; format=flowed
Content-Transfer-Encoding: 7bit

%^J%^Fjuly^% http://%^P%^R2-6^%:qwertyuiopasdfghjklzxcvbnmeuioa^%.%^Fjuly_link^%/^%
```

Figure 7.1 Modèle de courriel distribué aux *bots*

– **Collecte d'adresses électroniques.** Lorsqu'une machine est infectée, un *thread* du programme explore son disque dur à la recherche d'adresses électroniques contenues dans les fichiers, probablement pour les revendre ou leur envoyer des pourriels. Tous les fichiers dont l'extension n'est pas dans une liste spéciale (avi, mp3, 7z, etc.) sont parcourus. De plus, un mécanisme de répartition de charge est utilisé pour rendre la collecte plus furtive : le programme mesure le temps pris pour lire un certain nombre d'octets et, si celui-ci est trop long, alors il s'endort pour un moment.

– **Capture de mots de passe.** *Waledac* installe la bibliothèque de capture réseau WinPcap sur les machines infectées. Le trafic réseau est alors filtré pour rechercher les mots de passe FTP, SMTP et HTTP utilisés par la machine, qui sont ensuite transmis aux contrôleurs du *botnet*. Nunnery *et al.* [111] ont montré dans leur analyse que les mots de passe SMTP servent à envoyer des pourriels, qui ont plus de chance de ne pas être reconnus comme tels par les filtres, parce qu'envoyés depuis de véritables comptes courriel.

– **Installation d'autres familles.** Le *botnet* a la capacité de déployer des programmes sur les machines infectées, typiquement d'autres familles de logiciels malveillants, contre rémunération. De faux antivirus ont ainsi été téléchargés sur certains *bots*, en même temps que des applications faisant état de façon visible d'un problème (comme des popups pornographiques), pour inciter l'utilisateur à croire le faux antivirus qui lui annonce une infection (avant de lui demander son numéro de carte de crédit pour le « désinfecter »).

– **Attaques par déni de service.** Chaque *bot* peut recevoir l'ordre de se lancer dans une attaque par déni de service. Nous n'avons jamais vu cette fonctionnalité du code utilisée

lors de notre suivi de *Waledac*.

7.2.3 Réseau

Nous allons ici décrire les aspects réseau nécessaires à la compréhension de l'attaque que nous avons implémentée contre *Waledac*, en commençant par l'architecture, puis en décrivant la gestion des listes de pairs. Afin de rester concis, nous ne parlerons pas du trafic de contrôle du *botnet*, c'est-à-dire celui par lequel les tâches sont distribuées aux machines. Une description complète du protocole peut être trouvée dans [34].

Architecture

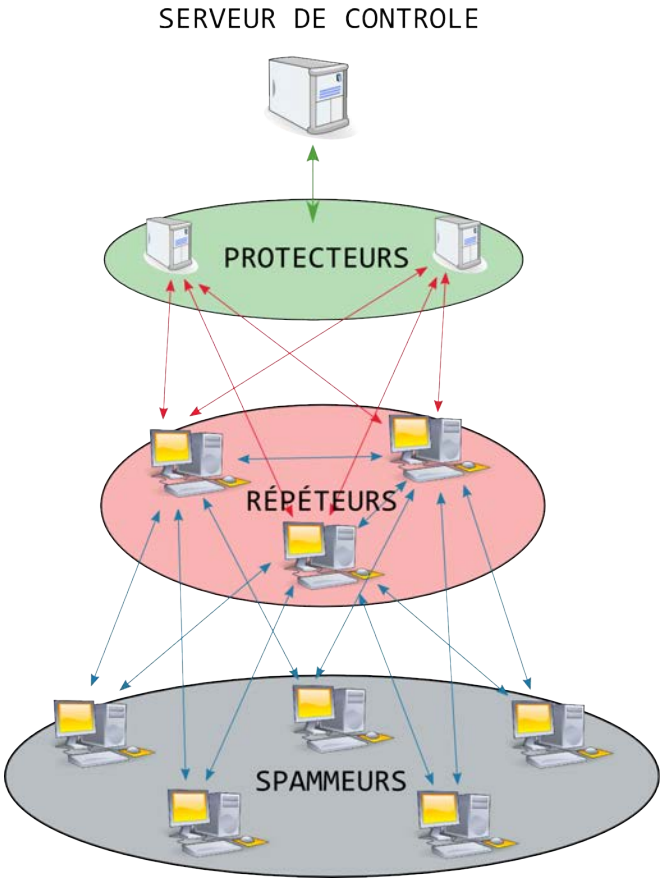


Figure 7.2 Architecture du *botnet*

L'architecture du *botnet* se compose de 4 couches, comme montré en Figure 7.2, qui sont définies comme ceci :

– **Les spammeurs :** ce sont les *bots* qui effectuent le travail de base, comme envoyer des courriels. Ce sont des machines Windows avec seulement des adresses IP privées, c'est-à-dire sans interfaces réseaux directement sur Internet. Ils ne se connaissent pas entre eux et vont chercher leurs tâches à accomplir en contactant les répéteurs dont ils stockent et mettent à jour les adresses (par un algorithme que nous verrons plus loin).

– **Les répéteurs :** ces *bots* servent de relais pour le trafic de contrôle du réseau, et également pour les requêtes HTTP et DNS pour les noms de domaines utilisés par *Waledac*. Ces machines ont une adresse IP publique sur une de leurs interfaces et sont directement joignables de l'extérieur. Elles relayent les requêtes des spammeurs vers les protecteurs et, quand il s'agit de leurs propres requêtes, elles passent également par un autre répéteur.

– **Les protecteurs :** il s'agit de serveurs Linux servant de dernier relais vers le serveur de contrôle. Contrairement aux spammeurs et répéteurs, ce sont donc des machines directement sous le contrôle des opérateurs du *botnet*, et pas des utilisateurs infectés. Il y en avait 8 durant l'année 2009.

– **Le serveur de contrôle (CC) :** l'entité qui contrôle le réseau. Nunnery *et al.* [111] ont décrit en détail l'architecture de l'infrastructure de contrôle, mais pour nos besoins il n'est pas nécessaire d'être plus précis.

Une machine nouvellement infectée par *Waledac* prendra donc soit le rôle d'un spammeur, soit celui d'un répéteur, selon qu'elle possède une adresse IP publique ou non. D'après nos observations, la proportion de spammeurs est d'environ 80% des *bots*.

Remarque 22. On peut observer que l'architecture du *botnet* n'est pas décentralisée – il y a un unique serveur de contrôle – bien que les machines infectées contactent seulement d'autres *bots*, ce qui est donc une relation pair-à-pair de leur point de vue.

Mise à jour de la liste de pairs

Les exemplaires de *Waledac* contiennent une liste de 100 à 500 répéteurs codée en dur. Cette liste est ensuite stockée dans une clé de registre nommée *RList* sous la forme d'un fichier XML, dont un exemple est donné dans la Figure 7.3. La liste contient un *timestamp* global (`<localtime>`) qui représente la dernière fois où elle a été mise à jour. Chaque répéteur (`<node>`) est défini par une adresse IP, un numéro de port, un *timestamp* local – dont le calcul sera expliqué par la suite – et un identifiant de 16 octets. Cette liste est toujours

ordonnée selon les *timestamps* locaux, de l'entrée la plus récente à la plus ancienne. Elle permet à chaque machine de connaître une partie du *botnet*, et c'est dans cette liste que celle-ci choisira un répéteur pour contacter le CC.

```
<lm>
<localtime>1244053204</localtime>
<nodes>
<node ip="a.b.c.d" port="80" time="1244053204">469abea004710c1ac0022489cef03183</node>
<node ip="e.f.g.h" port="80" time="1244053132">691775154c03424d9f12c17fdf4b640b</node>
...
</nodes>
</lm>
```

Figure 7.3 Exemple de *RList*

Il y a deux façons de mettre à jour la *RList* :

– **Par échange régulier avec les autres bots :** les machines infectées contactent régulièrement les répéteurs de leur *RList* pour faire des échanges d'information. Le choix du répéteur à contacter est aléatoire et, une fois qu'il est fait, le *bot* lui transmet un extrait de 100 répéteurs choisis aléatoirement dans sa liste. Le récepteur répond alors par un extrait de sa propre liste. Lorsque l'initiateur de l'échange est un répéteur, il met ses informations dans l'extrait envoyé, avec le *timestamp* le plus récent possible, ce qui lui permet de se faire connaître dans le *botnet*.

– **Par connexion à un site Internet :** tous les binaires *Waledac* contiennent une URL pointant vers une page spéciale maintenue par les contrôleurs du *botnet*. Quand une machine échoue à contacter dix répéteurs de sa *RList* d'affilés, elle fait une connexion vers cette URL où se trouve une liste de répéteurs actifs, mise à jour toutes les 10 minutes. Ce mécanisme permet de recréer un lien avec des *bots* qui n'ont plus d'informations à jour dans leur liste (par exemple parce qu'ils sont restés longtemps éteints).

L'algorithme d'insertion des nouvelles informations dans la liste sera décrit par la suite, il est celui qui est à la base de notre attaque.

Mise à jour de la liste des protecteurs

Les répéteurs ont besoin d'avoir une liste de protecteurs à jour afin de leur relayer le trafic de contrôle. Cette liste est échangée entre répéteurs et a la particularité d'être signée. Chaque

exemplaire de *Waledac* contient une clé publique pour vérifier la signature. Ce mécanisme empêche l'injection d'une liste de protecteurs, puisque seuls ceux qui possèdent la clé privée peuvent en créer une acceptée par les *bots*. La liste échangée contient également un *timestamp* permettant à chaque répéteur de vérifier si elle est plus récente que la sienne, et de remplacer celle-ci si la signature de la nouvelle liste est correcte.

7.3 Mise en place de l'émulation

Nous avons décrit en §6.3.3 la méthodologie expérimentale de l'émulation de réseaux de machines zombies. L'étape 3 de celle-ci consiste à reproduire l'infrastructure du véritable *botnet*, nous en présentons ici les spécificités de l'expérience avec *Waledac*.

- **Machines infectées :** une machine virtuelle Windows XP contenant un exemplaire de *Waledac* est utilisée pour tous les *bots*. Cette machine se voit assigner une adresse IP privée ou publique selon son rôle (spammeur ou répéteur). Initialement, nous modifions la *RList* de la machine pour qu'elle contienne les 500 adresses IP des répéteurs de notre réseau. Elle contient également un script de contrôle, qui nous permet par exemple de l'infecter et de la désinfecter à distance, ou bien encore de prendre des mesures et de récupérer les résultats.

- **Serveur de contrôle :** une machine virtuelle Linux s'exécutant seule sur une machine physique de notre cluster joue le rôle de CC. Grâce à notre étude préliminaire de *Waledac* [34] nous avons pu reproduire le CC dans un script capable de dialoguer avec les *bots*. Cette machine comporte 8 interfaces réseau qui jouent le rôle des protecteurs. Il faut remarquer que, comme la liste des protecteurs est signée, nous avons dû utiliser les mêmes adresses IP que celles du véritable *botnet*. De plus, nous avons fait en sorte que notre CC se comporte de façon similaire à ce que nous avions observé sur Internet, par exemple en distribuant à chaque spammeur des tâches comprenant entre 500 et 1 000 adresses de courriel, car c'est ce que faisait le véritable *botnet*. Nous avons également implémenté le mécanisme de mise à jour de la *RList* par URL décrit précédemment, en publiant toutes les 10 minutes une liste constituée des répéteurs les plus actifs sur cette URL.

- **Environnement :** diverses machines virtuelles Linux hébergent les services nécessaires au bon fonctionnement du *botnet* (DNS, SMTP, HTTP).

Ainsi, nous avons reproduit le réseau de machines zombies *Waledac* avec 2 300 spammeurs et 500 répéteurs, soit 2 800 *bots*. Cette proportion spammeurs/répéteurs est similaire à celle que nous avons observée dans le véritable *botnet*.

7.4 Expérience : prise de contrôle du *botnet*

La question de recherche à laquelle nous voulions répondre en émulant *Waledac* était de vérifier la vulnérabilité de ses communications pair-à-pair que nous avions précédemment conjecturée [34]. Pour expliquer l'attaque, il nous faut d'abord décrire l'algorithme de mise à jour des *RLists*.

7.4.1 Algorithme de mise à jour des *RLists*

Lorsqu'un *bot* reçoit un extrait de *RList* à insérer dans sa propre liste, selon les deux mécanismes décrits en §7.2.3, cet extrait a la forme montrée dans la Figure 7.4.

```
<lm>
<localtime>UpdateGlobalTS</localtime>
<nodes>
<node ip="UpdateIP1" port="80" time="UpdateLocalTS1">UpdateID1</node>
<node ip="UpdateIP2" port="80" time="UpdateLocalTS2">UpdateID2</node>
...
</nodes>
</lm>
```

Figure 7.4 Extrait de *RList* reçu par un *bot*

Il s'agit donc d'une liste de répéteurs, où $UpdateGlobalTS$ et $UpdateLocalTS_i$ sont respectivement les *timestamps* global, et locaux, de la *RList* de l'émetteur de l'extrait. Le *bot* qui reçoit ce message calcule alors pour chaque entrée i de cette liste un nouveau *timestamp* par la formule suivante, où $CurrentTS$ est le *timestamp* actuel :

$$NewTS_i = CurrentTS - |UpdateGlobalTS - UpdateLocalTS_i| \tag{7.1}$$

La valeur calculée permet au récepteur d'insérer les informations du pair i dans sa *RList*, de façon à ce qu'elle reste ordonnée de la plus récente à la plus ancienne entrée. S'il connaissait déjà ce répéteur, il met à jour sa position dans la liste avec le nouveau *timestamp*. Finalement, seules les 500 premières entrées de la nouvelle liste sont conservées.

La formule 7.1 semble à priori absconse, mais elle est en fait assez intuitive : la différence entre $UpdateGlobalTS$ et $UpdateLocalTS_i$ représente « l'âge » de l'entrée i dans la liste de l'émetteur, c'est-à-dire depuis combien de temps il n'a pas vu ce répéteur. En le soustrayant à $CurrentTS$, cela permet de calculer un *timestamp* qui prend en compte cet âge (moins l'entrée est récente pour l'émetteur, moins elle le sera pour le récepteur). Remarquons que le fait de prendre la valeur absolue de la différence empêche une attaque qui consisterait à envoyer

des mises à jour telles que le $NewTS_i$ calculé serait strictement supérieur au *CurrentTS*, créant ainsi des entrées dans le futur, qui seraient alors considérées comme les plus récentes et resteraient constamment dans la *RList*.

7.4.2 Attaque : insertion de faux bots

Principe. Comme expliqué précédemment, chaque *bot* est identifié par un ID de 16 octets dans la *RList*. En particulier, les adresses IP n'ont pas besoin d'être uniques dans cette liste. L'idée de l'attaque est alors de créer un *Super Répéteur (SR)* dont l'adresse IP sera répandue dans le réseau sous différents IDs, et qui aura de ce fait de grandes chances d'être choisi comme relais par les vrais *bots*.

Pour propager les informations de notre SR, nous utilisons le mécanisme de mise à jour des listes par échange d'extraits. La propagation de nos informations est rendue plus facile par un manque de rigueur du programme : *Waledac* ne vérifie pas si les extraits qu'il reçoit contiennent seulement 100 entrées (la taille habituelle). Il est donc possible d'envoyer un message avec 500 répéteurs et de remplir toute la *RList* du receveur, pour peu que les *timestamps* soient bien choisis. Plus précisément, les messages envoyés ont la forme décrite dans la Figure 7.5.

```
<lm>
<localtime>0</localtime>
<nodes>
<node ip="1.2.3.4" port="80" time="0">00000000000000000000000000000001</node>
<node ip="1.2.3.4" port="80" time="0">00000000000000000000000000000002</node>
...
<node ip="1.2.3.4" port="80" time="0">00000000000000000000000000000500</node>
</nodes>
</lm>
```

Figure 7.5 Message de mises à jour pour promouvoir un SR avec l'adresse IP `1.2.3.4`

Quand un *bot* reçoit ce message, la différence nulle entre *UpdateGlobalTS* et chaque $UpdateLocalTS_i$ implique que $NewTS_i$ = *CurrentTS* pour toutes les entrées. Autrement dit, chaque répéteur reçu se retrouve avec le *timestamp* le plus récent possible, et il est donc inséré en première position. Par conséquent, toutes les anciennes entrées sont remplacées, car elles sont désormais au-delà de la position 500 dans la liste.

Effets. Les conséquences de cette technique sur le *bot* cible dépendent de son rôle :

– **S'il s'agit d'un répéteur,** il y a une situation de *race condition* après qu'il ait reçu ce message. En effet, il est très probable que ce répéteur soit présent dans des listes d'autres *bots*, et que ceux-ci lui envoient des mises à jour avec des vrais répéteurs. Si un certain temps s'est écoulé depuis notre envoi, ces nouvelles informations seront plus récentes que les nôtres, et les remplaceront. Ceci implique que nous devons continuer à envoyer à notre cible nos messages de mises à jour, tant que tout le *botnet* n'est pas sous notre contrôle.

– **S'il s'agit d'un spammeur,** le résultat de l'attaque est beaucoup plus net : comme le spammeur n'est pas joignable directement (c'est lui qui contacte les répéteurs et pas l'inverse), il va être totalement isolé du reste du réseau et ne passera plus que par notre SR. Néanmoins, il faut qu'on ait été contacté par le spammeur pour avoir pu y délivrer notre mise à jour (ce qui implique d'avoir infecté un certain nombre de répéteurs qui lui auront alors donné nos informations). Ainsi, le contrôle des répéteurs représente le point crucial de l'attaque.

7.4.3 Implémentation

Pour l'implémentation de l'attaque contre notre *botnet* émulé, nous avons utilisé trois entités distinctes, tel qu'indiqué dans la Figure 7.6 :

– **1 attaquant**, qui envoie à des répéteurs cibles – dont le nombre sera notre variable expérimentale – l'extrait de liste décrit précédemment. Celui-ci contient les adresses IP des SR, et il est renvoyé toutes les minutes aux cibles afin d'empêcher le remplacement de nos informations. L'attaquant ne joue donc pas le rôle d'un *bot*, il est une entité externe.

– **3 super-répéteurs (SR)**, dont les informations vont être propagées dans le *botnet*, et qui relayent tout le trafic de contrôle vers une seule machine, décrite ci-dessous. Ils répondent également aux demandes de mises à jour avec la liste utilisée par l'attaquant.

– **1 serveur de contrôle blanc**, par opposition au vrai CC, que nous appellerons *noir*, qui va réceptionner et répondre au trafic relayé par les SR. Si nous ne répondions pas aux *bots*, ceux-ci utiliseraient alors le mécanisme de mise à jour de leur liste par URL, et obtiendraient les informations de véritables répéteurs. Pour garder le contrôle des *bots*, nous distribuons donc une tâche bénigne en utilisant le fait que chaque spammeur reçoit habituellement l'adresse d'un serveur SMTP lui permettant de s'assurer de sa capacité à envoyer des courriels. Notre CC blanc leur donne une adresse de serveur SMTP injoignable, ce qui leur fait croire qu'ils ne sont pas capables d'envoyer des courriels et les rend inutiles.

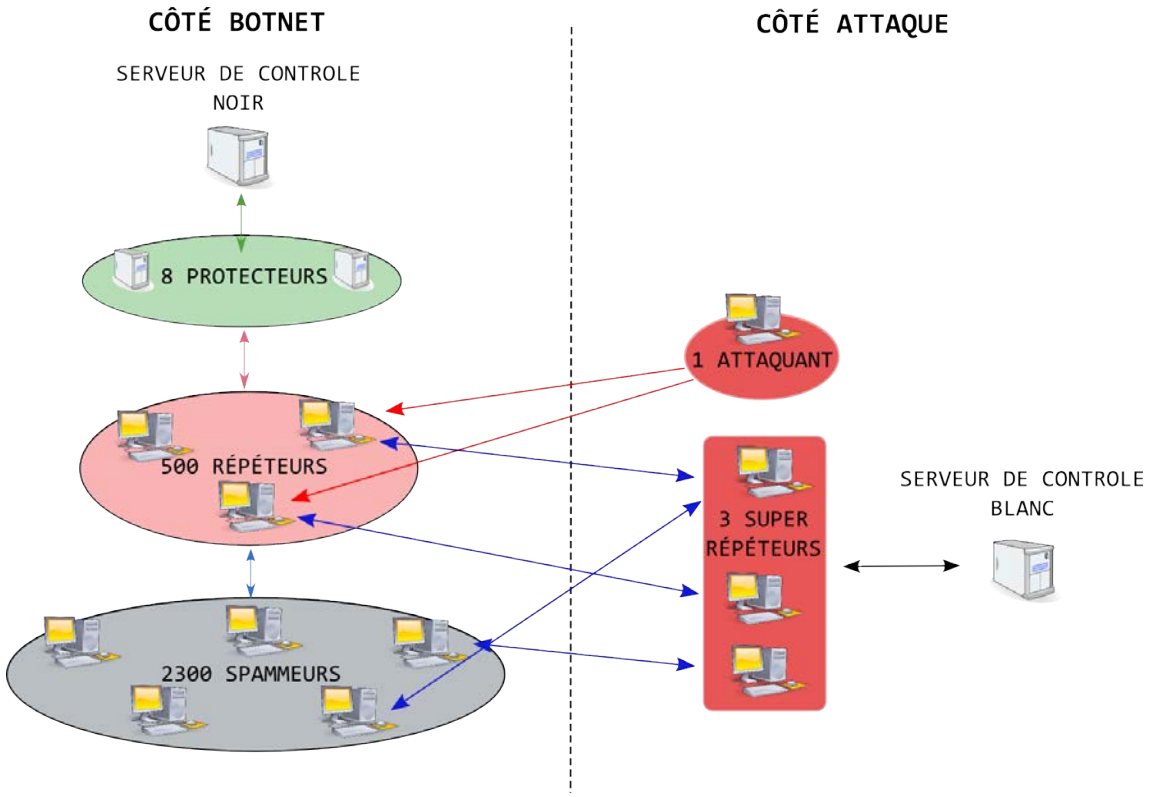


Figure 7.6 Mise en place de l'attaque contre le *botnet*

7.5 Résultats

Le but de notre expérience est de rendre compte de l'efficacité de notre attaque contre le *botnet*. Pour ce faire, nous présenterons d'abord les différentes mesures que nous avons mises en place, puis nous décrirons une observation expérimentale que nous avons pu faire sur l'utilisation de la cryptographie, pour finalement présenter les résultats de l'attaque.

7.5.1 Mesures

Afin de rendre compte de l'activité du *botnet* et de l'efficacité de l'attaque, nous avons mis en place les mesures suivantes :

– *Charge du processeur sur les CC chaque 30 secondes :* cette mesure permet d'estimer la charge des deux serveurs de contrôle, blanc et noir, lorsqu'ils gèrent les requêtes des *bots*.

– *Nombre de messages NOTIFY reçus par les CC chaque minute :* un message NOTIFY est le deuxième message échangé entre un *bot* et un CC lorsqu'ils entament un dialogue [34]. Le nombre de ces messages est un indicateur robuste du nombre de machines qui commu-

niquent avec chaque CC car les demandes de connexion avortées ne sont ainsi pas prises en compte. Autrement dit, il s'agit d'une mesure de *l'influence* de chacun des CC.

– *Nombre de courriels reçus chaque minute :* le but principal de *Waledac* étant d'envoyer des courriels, en mesurant leur nombre cela nous donne l'état de santé du *botnet*. Pour réaliser la mesure, notre CC noir distribue des adresses de courriel avec un domaine pour lequel notre serveur DNS retourne l'adresse de notre serveur SMTP, qui recevra donc tous les courriels envoyés.

– *Nombre de spammeurs qui contactent notre serveur SMTP chaque minute :* c'est une autre mesure de l'activité d'envoi de courriels, qui s'effectue elle au niveau réseau. La pertinence de cette mesure sera discutée par la suite.

– *Pourcentage de SR dans les RLists des répéteurs à chaque mise à jour :* l'objectif de notre attaque étant de remplacer les répéteurs par nos SR dans les listes de pairs, le pourcentage des SR dans ces listes représente la progression de l'attaque. Nous nous focalisons sur les *RLists* des répéteurs, car c'est leur contrôle qui est la clé de l'attaque, tel qu'expliqué précédemment. Le script qui s'exécute sur chaque *bot* prend une copie de la liste à chaque fois qu'elle est modifiée, et ces copies sont analysées à la fin de l'expérience.

7.5.2 Observation sur l'utilisation de la cryptographie

Avant de présenter les résultats de l'attaque, nous allons décrire ici une observation annexe, qui constitue à notre avis une preuve de l'intérêt de notre approche pour étudier les *botnets*.

Usage de la cryptographie par *Waledac*. Le protocole de contrôle du réseau, que nous avons décrit dans [34], permet de donner à chaque *bot* une clé cryptographique unique pour chiffrer ses communications avec le CC, c'est-à-dire une clé de session. De ce fait, nous avons été très surpris de nous apercevoir que dans le véritable *botnet* **tous les *bots* recevaient exactement la même clé de session**. Les conséquences de ce choix sont importantes : le répéteur qui sert de relais à un *bot* peut alors déchiffrer ses communications, et même les modifier. Cela nous a permis de collecter un grand nombre d'informations en observant simplement le trafic passant par nos répéteurs infiltrés dans le vrai *botnet*.

Notre première hypothèse pour expliquer ce choix était l'incompétence des contrôleurs de *Waledac*, ce qui nous a même fait écrire un article avec un titre plutôt condescendant [34]. Peu de temps après avoir fait cette observation, le CC s'est mis à distribuer une clé de session

différente *toutes les 10 minutes*. Comme tous les *bots* recevaient toujours la même clé pendant cette période, cela n'empêchait pas le déchiffrement du trafic, il suffisait à notre répéteur de redemander au CC la nouvelle clé lorsqu'il y avait un changement. Nous avions alors la plus grande peine à comprendre ce nouveau choix : si les contrôleurs étaient conscients du problème de distribuer la même clé de session, pourquoi ne pas simplement donner une clé unique à chaque *bot* ?

Point de vue des contrôleurs. En construisant notre propre *botnet*, la réponse nous est rapidement apparue de façon claire. Pour ce faire, nous avons tenté, par curiosité, de mettre en place un système de gestion de clés de session rigoureuse, c'est-à-dire en distribuant une clé unique à chaque *bot*. Il est très rapidement apparu que c'était impossible à réaliser, du fait de la charge énorme que cela entraîne pour le CC. En effet, les machines infectées par *Waledac* sont très bavardes : elles communiquent fréquemment avec le CC et demandent des nouvelles tâches dès qu'elles en ont terminé une. Par conséquent, en attribuant une clé unique à chaque *bot*, cela force le CC a être constamment en train de chiffrer de nouveaux messages, ce qui est coûteux en ressources. À l'inverse, avec une seule clé de session, chaque message n'a besoin d'être chiffré qu'une seule fois par le CC, ce qui diminue le nombre d'opérations cryptographiques. Pour comprendre l'importance du phénomène, il faut noter qu'un *bot* et le CC communiquent par un dialogue composé de plusieurs messages [34], et que la majorité des réponses du CC sont en fait exactement les mêmes à chaque nouveau dialogue (*acknowledgement*, ordre générique, etc.). De ce fait, avec la stratégie des clés différentes ces messages génériques doivent être chiffrés pour chaque *bot*, tandis qu'avec une seule clé de session une seule opération de chiffrement est nécessaire, ce qui soulage le CC.

Pour valider cette observation, nous avons mesuré sur le CC noir la charge CPU toutes les 30 secondes pendant 4 heures avec chacune des deux stratégies. Le résultat est montré dans la Figure 7.7. Ainsi, quand une clé de session différente est donnée à chaque *bot*, la charge CPU du serveur de contrôle est en moyenne à 60%, avec des pics autour de 90%. D'un autre côté, quand une même clé est utilisée pour tous les *bots*, la charge CPU oscille autour de 10%. Étant donné que notre réseau ne comprend « que » 3 000 machines, on peut aisément imaginer que le phénomène est amplifié avec plus de 50 000 *bots*, comme c'était le cas du véritable *botnet*.

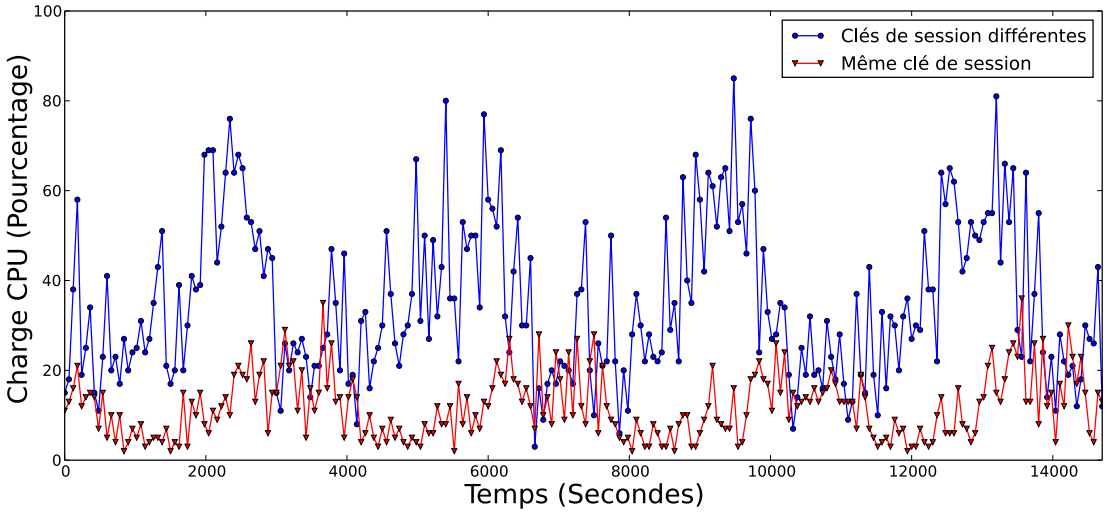


Figure 7.7 Comparaison de la charge CPU du CC selon la stratégie cryptographique

On peut conclure que le choix d'utiliser une même clé de session pour tous les *bots* a été fait en connaissance de cause, afin d'éviter que le CC ne soit surchargé de calculs cryptographiques. Cette décision est évidente lorsqu'on se met à la place des contrôleurs, mais bien plus difficile à comprendre lorsqu'on observe la situation du point de vue d'un *bot*.

Remarque 23. Cette observation n'implique pas qu'il aurait été impossible pour *Waledac* d'utiliser correctement la cryptographie, mais cela aurait nécessité une remise à plat de son protocole, par exemple pour diminuer le nombre de messages nécessaires à la demande d'une tâche par un *bot*.

7.5.3 Résultats de l'attaque

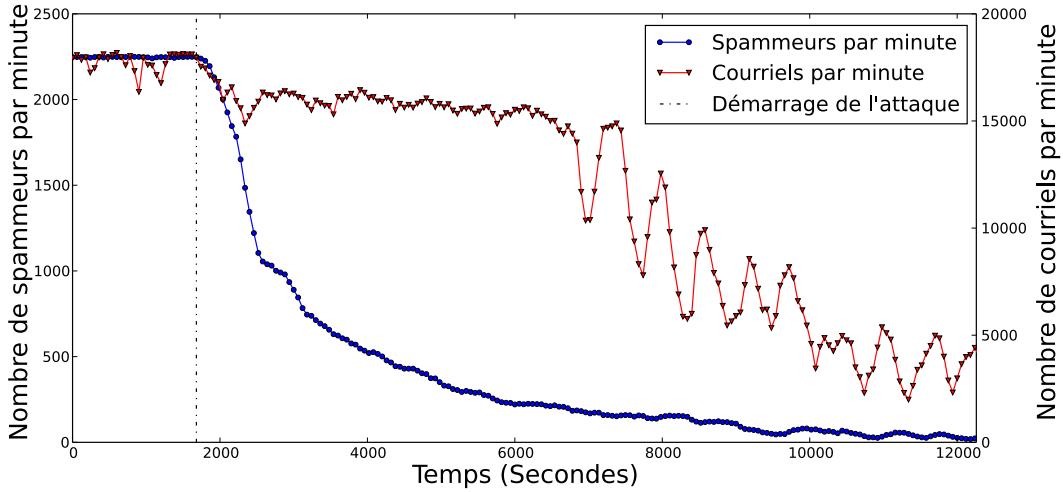
La variable expérimentale de notre expérience est le *nombre de répéteurs ciblés directement*, c'est-à-dire le nombre de répéteurs auxquels l'attaquant envoie l'extrait de liste. Le reste des *bots* est touché de façon indirecte, par échange d'information avec les cibles directes. En effet, attaquer directement tous les répéteurs n'aurait pas été réaliste, car il est peu probable d'avoir assez de ressources pour le faire sur le vrai *botnet* qui comporte plusieurs milliers de répéteurs. De ce fait, nous avons réalisé des attaques avec 100, 25, 10 et 5 cibles directes. Notre objectif est à la fois de valider que l'attaque fonctionne, et de déterminer l'impact du nombre de cibles directes sur son efficacité.

Pour chaque résultat présenté ici, nous avons réalisé au moins 5 fois l'expérience, en nous assurant que les écarts types des mesures – présentées en Annexe B – étaient satisfaisants, et

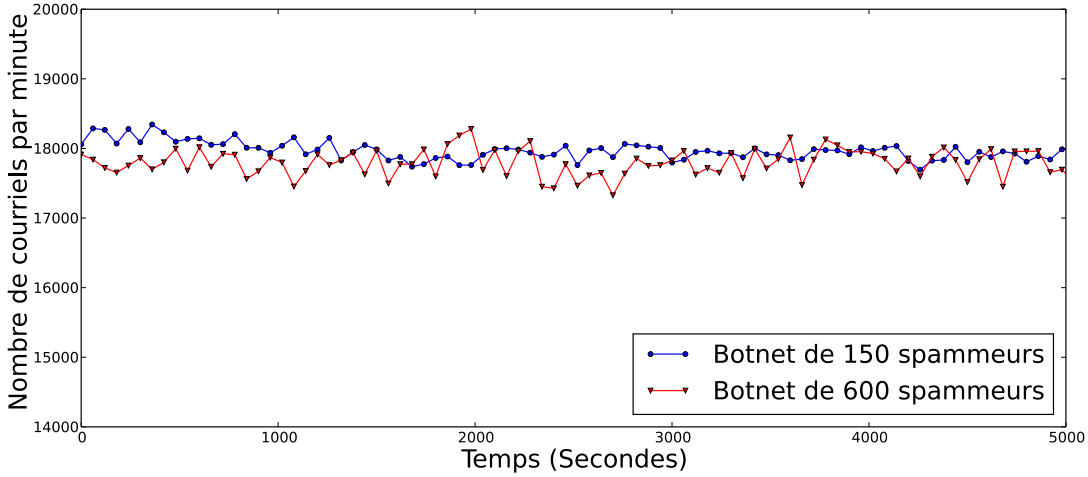
nous présentons la moyenne des expériences. Avant de lancer chaque attaque, nous attendons que le *botnet* atteigne un état de fonctionnement normal.

Conséquences sur l'envoi de courriels

Lors d'une première expérience, nous avons mesuré l'effet d'une attaque à 25 cibles directes sur la capacité d'envoi de courriels du *botnet*. Comme l'indique la courbe rouge de la Figure 7.8(a), moins de 3 heures après le début de l'attaque le nombre de courriels reçus par le serveur SMTP chaque minute passe sous la barre des 5 000, alors qu'il était à plus de 18 000 avant l'attaque, ce qui représente **une diminution de près de 80% du nombre de courriels**.



(a) Effets d'une attaque à 25 cibles sur l'envoi de courriels



(b) Expériences sans attaque sur des *botnets* de 150 et 600 spammeurs

Figure 7.8 Mesure de la capacité du *botnet* à envoyer des courriels

Au-delà de la réussite de l'attaque, cette courbe présente trois aspects intéressants :

- **Le *botnet* continue à envoyer environ 4 000 courriels par minute, en dépit de notre attaque.** Autrement dit, certains spammeurs ne contactent pas nos SR, et il y a donc des répéteurs qui continuent de distribuer des informations valides. Le *botnet* fait ainsi preuve de résilience et ne passe pas complètement sous notre contrôle, ce qui est une conséquence de la *race condition* au niveau des répéteurs décrite précédemment, que nous ne gagnons pas complètement avec ce nombre de cibles directes.

- **Un plateau apparaît juste après que l'attaque ait commencé.** Ce comportement, qui dure près de 80 minutes, semble suggérer que notre attaque n'a pas réellement d'effets pendant cette fenêtre de temps. Pour valider ceci, nous avons mesuré le nombre de spammeurs qui contactent le CC par minute, en supposant qu'il resterait constant durant cette période. En fait, comme le montre la courbe bleue de la Figure 7.8(a), le nombre de spammeurs en contact avec le CC diminue constamment, ce n'est donc pas l'explication de notre plateau. Remarquons que le nombre de spammeurs est initialement autour de 2 300, ce qui est bien la population de notre *botnet*.
  Il faut alors comprendre pourquoi, malgré la diminution constante des contacts entre spammeurs et CC, le nombre de courriels envoyés par le *botnet* reste à peu près le même. Notre hypothèse fut alors que notre serveur SMTP ne recevait en fait pas tous les courriels envoyés, certains devaient être rejetés ou perdus. Pour valider cette hypothèse, nous avons fait une nouvelle expérience en mesurant sans attaque le nombre de courriels envoyés par minute avec un *botnet* de 150 spammeurs, et un autre de 600 spammeurs. La Figure 7.8(b) montre les résultats, et on peut observer que le nombre de courriels est similaire dans les deux *botnets*. Cela confirme notre hypothèse : notre serveur SMTP étant incapable de gérer tous les courriels, cela créé une limite dans le nombre qu'il peut observer, d'où le plateau.

- **Le nombre de courriels reçus suit un comportement cyclique d'une période d'environ 10 minutes après le plateau.** Ceci indique que les spammeurs restant dans le *botnet* travaillent puis redemandent une tâche de façon synchronisée. Nous pensons que ce phénomène est une conséquence de la manière dont le *botnet* a été démarré – par vagues successives comme expliqué en §6.3.4 –, et du fait que ce démarrage a eu lieu peu de temps avant l'attaque (moins d'une heure), ce qui n'a pas laissé le temps aux spammeurs de se désynchroniser par la réalisation de tâches plus ou moins lourdes. Ainsi, il nous parait peu probable que de tels cycles se produisent dans le véritable *botnet*.

Nous pouvons conclure de ces premières expériences qu'en raison de l'incapacité de notre

serveur SMTP à gérer l'ensemble des courriels qui lui sont envoyés, le nombre de courriels reçus n'est pas une bonne mesure. En particulier, cela veut dire que le nombre de courriels du *botnet* en activité de la Figure 7.8(a) est sous-estimé. À l'inverse, le nombre de demandes de connexion reçues par le serveur SMTP de la part des spammeurs est une mesure robuste, et c'est donc celle-là que nous utiliserons par la suite pour rendre compte de la capacité du réseau à envoyer des courriels. Une autre possibilité, pour continuer à utiliser le nombre de courriels comme mesure, aurait été de déployer différents serveurs SMTP et de répartir la charge.

La Figure 7.9 présente les effets de l'attaque sur le nombre de spammeurs qui contactent le serveur SMTP chaque minute, avec 100, 25, 10 et 5 cibles directes. On peut observer qu'**avec 100 cibles cela prend un peu plus d'une heure pour mettre le *botnet* dans un état quasi inactif**, c'est-à-dire avec moins de 50 contacts spammeurs-SMTP par minute, tandis qu'avec 5 cibles cela prend 2 fois et demie plus de temps pour atteindre le même état. L'écart type de l'expérience est présenté en Annexe B.

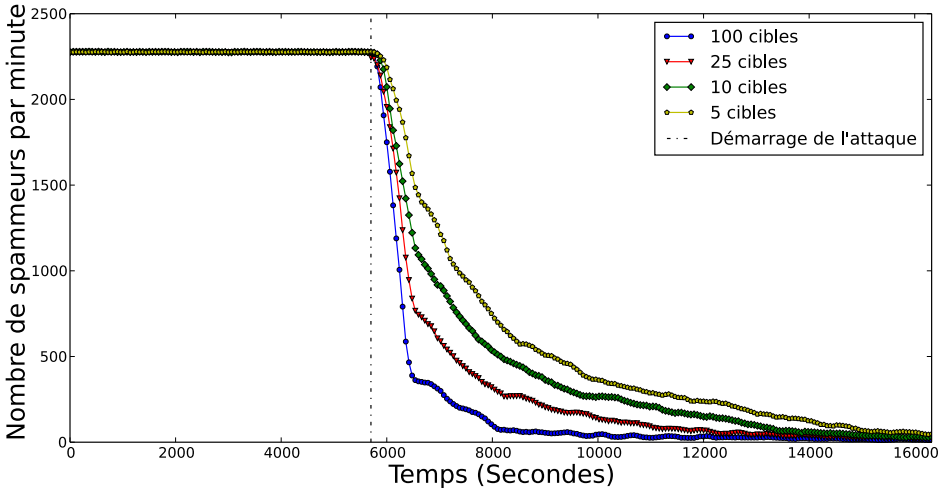


Figure 7.9 Contacts des spammeurs avec le serveur SMTP

Conséquences sur l'influence du CC noir

La Figure 7.10 présente le nombre de messages NOTIFY reçus par le CC noir par minute, avant et après des attaques à 100, 25, 10 et 5 cibles directes. On y observe une forte décroissance suite à l'attaque, par exemple avec 100 cibles cela prend moins d'une heure pour passer d'environ 300 messages NOTIFY par minute à 30. On remarque également que moins le nombre de cibles est élevé, plus le CC noir garde de l'influence dans le *botnet* et continue

à distribuer des ordres, sans surprise. Ensuite, la résilience du botnet précédemment décrite est encore visible : entre 20 et 100 contacts par minute – en fonction du nombre de cibles – sont faits entre les *bots* et le CC noir après l'attaque.

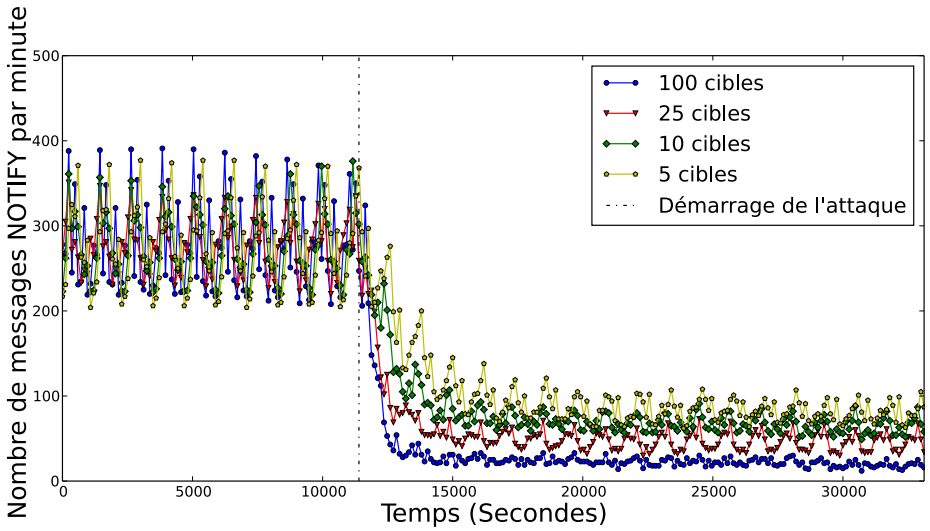


Figure 7.10 Nombre de demandes de tâches reçus par le CC noir

Finalement, la Figure 7.10 montre aussi un effet cyclique dans la distribution des ordres, ce qui s'explique à notre avis de la même manière que pour le nombre de courriels en §7.5.3, c'est-à-dire par le démarrage récent des machines par groupes. L'écart type de l'expérience est présenté en Annexe B.

La Figure 7.11 montre le nombre de messages NOTIFY reçus par les deux CC lors d'une attaque avec 100 cibles. On peut y observer la prise de contrôle très rapide du CC blanc, avec légèrement moins de messages que le CC noir avant l'attaque, puisque celui-ci garde le contrôle sur un faible nombre de *bots*. Ceci démontre clairement l'efficacité de l'attaque.

Conséquences sur les listes de pairs

La Figure 7.12 montre le pourcentage d'entrées qui sont des SR dans les listes de pairs *des répéteurs*. On peut observer une croissance rapide, et une stabilisation à un palier, qui montre que l'on n'a pas un contrôle total des listes de pair. La hauteur du palier dépend du nombre de cibles directes, car les listes de celles-ci sont plus faciles à infecter que celles des autres *bots*. Malgré cela, on contrôle avec seulement 5 cibles directes plus de 60% des entrées dans les listes de pairs, ce qui démontre l'efficacité de l'attaque. L'écart type de l'expérience est présenté en Annexe B.

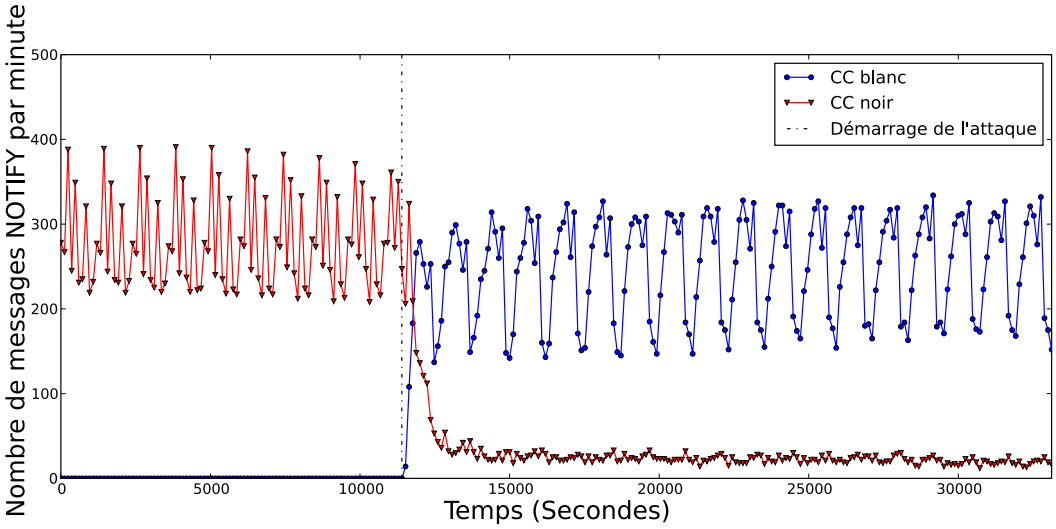


Figure 7.11 Nombre de demandes de tâches reçues par les CC noir et blanc lors d'une attaque avec 100 cibles

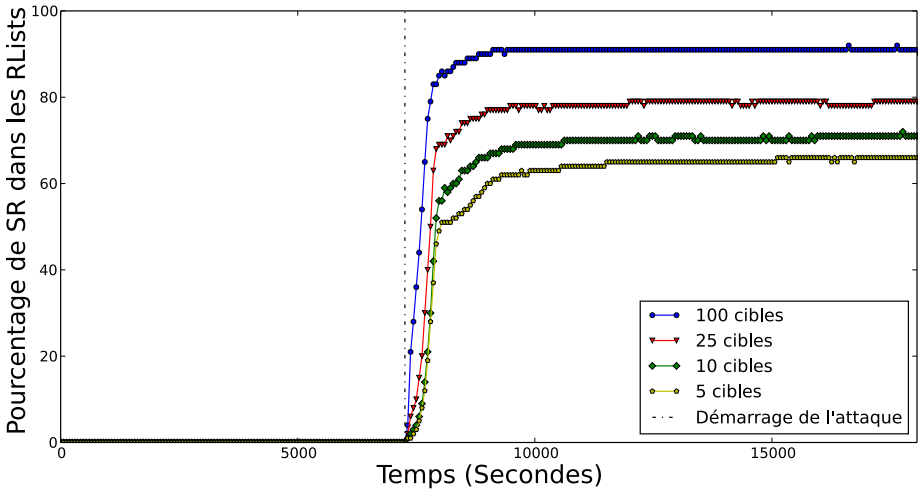


Figure 7.12 Part des SR dans les *RLists* des répéteurs

7.6 Résumé des contributions

Lors de ce cas d'étude, nous avons réalisé une série de contributions que nous résumons ci-dessous.

– Nous avons validé la vulnérabilité de *Waledac* à une attaque permettant la prise de contrôle de son *botnet*. Nous avons montré en particulier que celle-ci amène une baisse importante (i) de la capacité du *botnet* à envoyer des courriels, (ii) de l'influence de son CC et (iii) du

nombre de vrai *bots* dans les listes de pairs. Ces différentes mesures nous ont été rendues possibles par notre contrôle de l'environnement d'expérience.

– Nous avons exploré le nombre de cibles directes nécessaires à l'attaque. Bien que l'efficacité de l'attaque augmente avec le nombre de cibles directes, ce qui n'est pas une surprise, il est intéressant d'observer qu'un faible nombre de cibles permet quand même une prise de contrôle assez rapide et quasi totale. Étant donné qu'un faible nombre de cibles directes à l'avantage de rendre l'attaque plus furtive, il peut être intéressant de réfléchir à un nombre optimal de cibles en fonction de la taille du *botnet*. Ce type de raisonnement serait impossible à faire en étudiant le véritable *botnet* sur Internet.

– Nous avons développé un environnement d'expérience mature et adapté à des expériences de cette taille. En particulier nous avons mis de côté le nombre de courriels comme mesure, car il nécessitait de fortes ressources pour pouvoir être comptabilisé correctement, pour le remplacer par une mesure équivalente, mais moins coûteuse.

– Finalement, nous avons fait une observation qui n'est pas reliée à l'attaque, mais qui apporte un éclairage nouveau sur *Waledac* : le fait de partager une clé de session cryptographique entre les *bots* n'était pas un « mauvais » choix de la part des opérateurs du réseau, il s'agissait simplement d'un compromis pour assurer ses performances. Cette observation démontre l'intérêt de l'émulation comme méthode d'étude des *botnets*, car c'est en prenant la place des opérateurs que nous avons pu la faire.

7.7 Limitations et voies de recherche futures

Plusieurs limitations sont présentes dans notre expérience, et chacune peut donner lieu à une nouvelle piste de recherche :

– **Augmentation de l'échelle.** Nous avons construit un *botnet* d'environ 3 000 machines infectées, en respectant les proportions mesurées sur Internet entre les différentes catégories de *bots*. Ce nombre peut paraître à première vue modeste en comparaison des plusieurs dizaines de milliers de *bots* des véritables *botnets*, mais il nous permet de travailler à un ordre de grandeur similaire à ceux-ci, ce qui était précédemment impossible. Partant de cela, il y a peut-être une façon de s'assurer qu'une observation dans notre environnement est toujours valide lorsque la taille du réseau augmente. Autrement dit, existe-t-il un nombre de machines à partir duquel les résultats d'une émulation sont transposables avec confiance

sur Internet ? La réponse dépend probablement du *botnet* ciblé, mais il y a aussi sans doute des critères généraux à satisfaire pour pouvoir construire un tel raisonnement.

– **Réalisme de l'environnement.** Le fait d'utiliser de véritables machines infectées permet d'atteindre un niveau de réalisme difficile à égaler par d'autres méthodes, mais il reste encore des lacunes importantes dans notre environnement :

  – En premier lieu, notre réseau est très homogène, c'est-à-dire que pour deux machines infectées choisies aléatoirement, la latence entre les deux est sensiblement toujours la même, et très faible. Cela pourrait entraîner des biais dans nos mesures, par rapport au véritable *botnet*. Une solution serait donc d'introduire artificiellement de la latence dans les communications, selon un modèle à définir.

  – Deuxièmement, le comportement de l'utilisateur n'est pas émulé, en particulier nos machines ne s'éteignent jamais. De ce fait, nous n'avons pas de *diurnal behavior*, c'est-à-dire des variations de populations en fonction de l'heure de la journée. Une solution serait de créer un modèle de population dans lequel une fraction seulement des machines seraient allumées en fonction de l'heure. De plus, il faudrait également créer des changements à l'intérieur d'une population donnée par un cycle extinction-allumage régulier des machines. Finalement, nos machines restent constamment infectées, alors qu'en réalité il y a une course constante entre infection et désinfection. Un tel phénomène est difficile à modéliser en général, mais une stratégie telle que la diffusion d'un *patch* pourrait être introduite, par exemple en désinfectant un groupe de machines dans un court laps de temps.

  – Troisièmement, nous soupçonnons que la cause des comportements cycliques dans notre réseau – comme dans les Figures 7.8(a) et 7.10 – est la méthode que nous avons adoptée pour le démarrage du *botnet*. La détermination d'une méthode réaliste de démarrage du réseau reste un problème de recherche en soi, étant donné le manque de données sur le sujet.

  Du fait que l'attaque contre *Waledac* dure quelques heures à peine, ces lacunes ont un impact faible dans ce cas, mais pour toute expérience plus longue il sera nécessaire de les prendre en compte.

– **Avoir un adversaire.** Notre *botnet* n'a pas de contrôleur qui puisse le défendre. Il serait

intéressant d'émuler le comportement d'un « défenseur » du réseau, par exemple en s'inspirant de modèles de la théorie des jeux. Cela impliquerait de lister les « coups » jouables par un tel défenseur, et de déterminer quels sont ceux qui sont adaptés pour répondre à notre attaque.

7.8 Conclusion

Nous avons montré grâce à l'émulation que le *botnet Waledac* est vulnérable à une attaque qui permet de prendre son contrôle. Ainsi, nous avons pu répéter l'attaque de nombreuses fois, ce qui nous permit de montrer qu'il suffisait de cibler directement un petit nombre de *bots* pour contrôler la majorité du réseau. De plus, nous avons pu comprendre que le choix, à priori absurde, d'utiliser la même clé cryptographique pour chiffrer toutes les communications a été fait pour favoriser les performances, ce qui aurait été difficile à deviner par une autre approche.

CHAPITRE 8

CONCLUSION

8.1 Synthèse des travaux et des contributions

Protections de logiciels malveillants. Dans le chapitre 2, nous avons défini formellement les programmes en langage machine x86 à partir d'une machine abstraite et d'un langage assembleur simplifié. Nous avons pu alors expliciter les difficultés de l'analyse de ces programmes : le code auto modifiant, la complexité du jeu d'instruction, et l'absence d'informations de haut niveau. Pour finir, nous avons montré l'indécidabilité du désassemblage d'un programme machine, présenté les travaux sur le sujet, puis discuté l'absence de représentation complète de la sémantique du langage machine x86.

Dans le chapitre 3, nous avons construit un cadre formel autour de l'analyse dynamique, qui est notre méthode de choix pour étudier les protections de logiciels malveillants. En particulier, nous avons précisé ce que sont un traceur et une trace d'exécution, cette dernière étant l'objet de notre analyse. Ensuite, nous avons présenté notre implémentation pour tracer un programme, qui a servi de base à toutes nos expérimentations. Finalement, nous avons fait une revue des applications de l'analyse dynamique, ainsi que des travaux sur la couverture de code.

Après ces deux premiers chapitres pédagogiques, nous avons présenté dans le chapitre 4 une première réalisation originale par la cartographie expérimentale des protections de logiciels malveillants. Pour ce faire, nous avons tout d'abord défini la notion de protection de façon semi-formelle en partant de celle d'attaquant. Ensuite, nous avons émis l'hypothèse qu'un certain modèle de protection – basé sur le code auto modifiant, et la distinction entre couches de code de protection et de charge utile – est particulièrement prévalent dans les logiciels malveillants. Après avoir montré qu'aucun travail existant ne permettait de tester la validité de notre hypothèse, nous avons présenté les résultats de l'expérience que nous avons construite pour le faire. Ainsi, de nombreuses mesures réalisées sur près de 600 exemplaires de logiciels malveillants nous ont permis de dégager un modèle de protection prévalent parmi les logiciels malveillants.

Dans le chapitre 5, nous avons présenté une méthode d'identification des implémentations

cryptographiques dans les protections de logiciels malveillants. Pour ce faire, nous avons construit une abstraction robuste afin d'extraire les possibles implémentations cryptographiques d'une trace d'exécution. À partir de cela, nous avons récupéré leurs paramètres, dont nous avons pu tester la relation entrées-sorties pour identifier les implémentations cryptographiques. Nous avons validé expérimentalement notre approche en l'implémentant dans l'outil *Aligot*, qui nous a tout d'abord permis d'identifier avec succès des implémentations des algorithmes TEA et RC4 dans des programmes protégés, au contraire de tous les outils existants. En particulier, nous avons montré que deux familles de logiciels malveillants ont fait la même erreur en implémentant l'algorithme TEA. De plus, nous avons identifié des implémentations des algorithmes MD5 et AES, que cela soit dans des exemples synthétiques ou dans des logiciels malveillants protégés. Encore une fois, aucun des outils existants ne peut en faire autant. Finalement, nous avons pu aussi identifier l'opération de base de l'algorithme RSA, ainsi que des combinaisons et modifications de divers algorithmes cryptographiques.

Réseaux de machines zombies. Dans le chapitre 6, nous avons abordé l'étude des réseaux de machines zombies de façon globale, c'est-à-dire en prenant en compte le réseau dans son ensemble. Nous avons commencé par décrire les approches existantes, en montrant que l'émulation a l'avantage de combiner rigueur scientifique et éthique, au contraire de la simulation et de l'étude dans la nature. Ensuite, nous avons décrit l'implémentation de notre environnement d'émulation de *botnets*, grâce auquel nous avons pu construire un réseau de 3 000 machines infectées, dans un environnement complètement contrôlé et non connecté à Internet. C'est la première fois, à notre connaissance, qu'un environnement d'émulation à cette échelle est construit.

Le chapitre 7 a présenté un cas concret d'émulation, avec le *botnet Waledac*, dont nous voulions vérifier la vulnérabilité à une attaque. Nous avons d'abord introduit ce logiciel malveillant, et son protocole de communication, pour ensuite montrer comment nous avons reproduit son *botnet* dans notre environnement. Ensuite, nous avons présenté les résultats de notre expérience, qui mettent notamment en avant le faible nombre de cibles nécessaires au succès de l'attaque. En particulier, nous avons mis en place de nombreuses mesures expérimentales, pour la plupart impossibles à réaliser avec les autres méthodes d'analyse de *botnets*. De plus, nous avons découvert que le choix de distribuer une même clé cryptographique aux machines infectées n'était pas une grossière erreur des opérateurs de *Waledac*, mais plutôt un compromis pour optimiser les performances du réseau.

8.2 Perspectives

Les trois réalisations originales de cette thèse – la cartographie des protections, l'identification des fonctions cryptographiques, et l'émulation de *Waledac* – ne sont que des premiers pas, et elles ouvrent chacune des voies de recherche prometteuses. Ces suites à notre recherche ont été décrites en détail dans les chapitres concernés, et nous les résumons ici.

8.2.1 Cartographie des protections de logiciels malveillants

Notre outil d'analyse des protections de logiciels malveillants pourrait être utilisé pour attaquer les questions de recherche suivantes :

- **L'étude ciblée des protections des familles de logiciels malveillants à longue durée de vie.** Il serait en particulier intéressant de déterminer si différentes protections ont été utilisées – ce qui pourrait s'expliquer par l'impact des produits de sécurité –, ou si au contraire un unique modèle les a protégées durant leur existence.

- **La création de signatures de protecteurs.** L'état de l'art étant plutôt sommaire dans ce domaine – des outils statiques avec des signatures sur le code binaire –, il serait appréciable de pouvoir identifier et classifier automatiquement les protections avec des caractéristiques plus fines. Les méthodes habituelles de classification des programmes – par exemple basées sur le graphe de flot de contrôle – ne peuvent répondre à ce problème, du fait que les programmes protégés sont spécialement construits pour brouiller les pistes. Par contre, les caractéristiques que nous avons mesurées dans notre expérience sont adaptées à cette tâche.

- **La mesure de la véritable importance des protecteurs du domaine public.** Il faudrait notamment identifier les cas où ceux-ci sont combinés avec d'autres protections. Nous pourrions aussi déterminer l'importance des protections commerciales dans les logiciels malveillants, et valider l'observation faite dans d'autres travaux selon laquelle celle-ci a diminué ces dernières années.

8.2.2 Identification des fonctions cryptographiques

Dans le cadre de l'identification de la cryptographie, il serait intéressant d'**importer automatiquement dans notre base de référence tous les possibles algorithmes cryptographiques** rencontrés dans les traces d'exécution, afin de pouvoir ensuite les reconnaître s'ils sont réutilisés. Le problème est ici de reconstruire un programme et de l'exécuter sur de nouveaux arguments à partir d'une trace d'exécution. Ensuite, **le principe de notre**

comparaison entrée-sortie pourrait être utilisé pour n'importe quel algorithme déterministe, par exemple pour les algorithmes de compression.

Sur le plan théorique, remarquons que le fait d'avoir une abstraction – c'est-à-dire de regrouper un ensemble d'instructions dans une structure commune – facilite la compréhension humaine d'un programme informatique, et permet de développer des raisonnements complexes, comme celui qui nous a permis de partir des boucles pour identifier des fonctions cryptographiques. **La construction automatique de l'abstraction la plus adaptée pour analyser un programme donné** serait une capacité utile pour l'analyse des protections de logiciels malveillants.

8.2.3 Réseaux de machines zombies

La mise en place de notre environnement d'expérimentation avec les *botnets* ouvre la voie à de nouvelles pistes de recherche :

- **Augmenter l'échelle de nos observations :** au-delà de la problématique technique de l'ajout de machines supplémentaires dans notre environnement, il serait intéressant d'explorer les conditions qui assurent que les observations faites à une échelle réduite restent valides lorsque le nombre de machines augmente. Ainsi, nous pourrions garder la taille de notre environnement pour explorer des réseaux de taille réelle supérieure.

- **Améliorer le réalisme de l'environnement :** cela pourrait d'abord se faire en introduisant de la charge réseau artificielle, mais aussi en simulant le comportement de l'utilisateur. Cela implique de mettre en place un modèle de population prenant en compte l'extinction des machines pendant la nuit, mais aussi les allumages-extinctions qui ont lieu pendant la journée, et finalement le cycle infection-désinfection. Le démarrage du *botnet* constitue également un problème de recherche important pour augmenter le réalisme de l'émulation.

- **Avoir un adversaire :** il serait intéressant de simuler la réaction des opérateurs du *botnet* lorsque nous interagissons avec lui. Il faudrait alors créer un modèle simulant cet opérateur.

RÉFÉRENCES

[1] AGRAWAL, H. et HORGAN, J. R. (1990). Dynamic program slicing. *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI)*.

[2] AHO, A. V., LAM, M. S., SETHI, R. et ULLMAN, J. D. (2007). *Compilers : principles, techniques, and tools*, vol. 1009. Pearson/Addison Wesley.

[3] AJJAN, A. (2013). Russian ransomware takes advantage of windows powershell. `http://nakedsecurity.sophos.com/2013/03/05/russian-ransomware-windows-powershell`. Consulté le 27 Mars 2013.

[4] AURIEMMA, L. (2013). Signsrch tool. `http://aluigi.altervista.org/mytoolz.htm`. Consulté le 26 Mars 2013.

[5] AVTEST (2012). AVTest : The independent IT-security institute. `http://www.av-test.org/fr/page-daccueil/`. Consulté le 11 Mars 2013.

[6] BALAKRISHNAN, G., GRUIAN, R., REPS, T. et TEITELBAUM, T. (2005). Codesurfer/x86—a platform for analyzing x86 executables. *Proceedings of the 14th International Conference on Compiler Construction (CC)*.

[7] BALAKRISHNAN, G., REPS, T., MELSKI, D. et TEITELBAUM, T. (2008). WYSINWYX : what you see is not what you execute. *Verified Software : Theories, Tools, Experiments*, 202–213.

[8] BARFORD, P. et BLODGETT, M. (2007). Toward botnet mesocosms. *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets*.

[9] BAYER, U., HABIBI, I., BALZAROTTI, D., KIRDA, E. et KRUEGEL, C. (2009). A view on current malware behaviors. *Proceedings of the 2009 USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*.

[10] BAYER, U., MOSER, A., KRUEGEL, C. et KIRDA, E. (2012). Anubis : Analyzing unknown binaries. `http://anubis.iseclab.org`. Consulté le 7 Mars 2013.

[11] BEAUCAMPS, P., GNAEDIG, I. et MARION, J.-Y. (2010). Behavior abstraction in malware analysis. *Proceedings of the 1st International Conference on Runtime Verification (RV)*.

[12] BELLARD, F. (2005). Qemu, a fast and portable dynamic translator. *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX)*.

[13] BIGGERSTAFF, T. J., MITBANDER, B. G. et WEBSTER, D. (1993). The concept assignment problem in program understanding. *Proceedings of the 15th IEEE International Conference on Software Engineering (ICSE)*.

[14] BOLDEWIN, F. (2013). Peacomm.c Cracking the nutshell. `http://www.reconstructer.org/papers/Peacomm.C-Crackingthenutshell.zip`. Consulté le 26 Mars 2013.

[15] BONZINI, P. (2003). GNU lightning. `http://www.gnu.org/software/lightning/`. Consulté le 20 Février 2013.

[16] BRANCO, R. R., BARBOSA, G. N. et NETO, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-VM technologies. *Black Hat USA 2012*.

[17] BRAVERMAN, M. (2006). Windows malicious software removal tool : Progress made, trends observed. *Microsoft Antimalware Team Whitepaper*, 10.

[18] BROSCH, T. et MORGENSTERN, M. (2006). Runtime packers : the hidden problem. *Black Hat USA 2006*.

[19] BRUENING, D., DUESTERWALD, E. et AMARASINGHE, S. (2001). Design and implementation of a dynamic optimization framework for windows. *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*.

[20] BRUMLEY, D. et JAGER, I. (2012). The BAP handbook. `http://bap.ece.cmu.edu/doc/bap.pdf`. Consulté le 7 Mars 2013.

[21] BRUMLEY, D., JAGER, I., AVGERINOS, T. et SCHWARTZ, E. (2011). BAP : a binary analysis platform. *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*.

[22] BUCK, B. et HOLLINGSWORTH, J. K. (2000). An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14, 317–329.

[23] BUSTAMANTE, P. (2008). Packer (r)evolution. `http://research.pandasecurity.com/packer-revolution/`. Consulté le 06 Mai 2013.

[24] CABALLERO, J., JOHNSON, N., MCCAMANT, S. et SONG, D. (2009). Binary code extraction and interface identification for security applications. *Proceedings of the 17th ISOC Network and Distributed System Security Symposium (NDSS)*.

[25] CABALLERO, J., POOSANKAM, P., KREIBICH, C. et SONG, D. (2009). Dispatcher : Enabling active botnet infiltration using automatic protocol reverse-engineering. *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*.

[26] CABALLERO, J., YIN, H., LIANG, Z. et SONG, D. (2007). Polyglot : Automatic extraction of protocol message format using dynamic binary analysis. *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*.

[27] CADAR, C., DUNBAR, D. et ENGLER, D. (2008). Klee : unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*.

[28] CAI, H., SHAO, Z. et VAYNBERG, A. (2007). Certified self-modifying code. *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*.

[29] CALVET, J. (2012). Aligot : Cryptographic function identification in obfuscated programs - code. `https://code.google.com/p/aligot/`. Consulté le 01 Avril 2013.

[30] CALVET, J. (2012). Cryptographic function identification in obfuscated binary programs. *Reverse Engineering Conference 2012 (REcon)*.

[31] CALVET, J. (2012). Cryptographic function identification in obfuscated binary programs. *Hackito Ergo Sum 2012 (HES)*.

[32] CALVET, J. et BUREAU, P.-M. (2009). Analyse en profondeur de waledac et de son réseau. *MISC magazine*, 45.

[33] CALVET, J. et BUREAU, P.-M. (2010). Understanding swizzor's obfuscation scheme. *Reverse Engineering Conference 2010 (REcon)*.

[34] CALVET, J., DAVIS, C. R. et BUREAU, P.-M. (2009). Malware authors don't learn, and that's good ! *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE)*.

[35] CALVET, J., DAVIS, C. R., FERNANDEZ, J. M., GUIZANI, W., KACZMAREK, M., MARION, J.-Y., ST-ONGE, P.-L. *ET AL.* (2010). Isolated virtualised clusters : testbeds for high-risk security experimentation and training. *Proceedings of the 3rd International Conference on Cyber Security Experimentation and Test (CSET)*.

[36] CALVET, J., DAVIS, C. R., FERNANDEZ, J. M., MARION, J.-Y., ST-ONGE, P.-L., GUIZANI, W., BUREAU, P.-M. et SOMAYAJI, A. (2010). The case for in-the-lab botnet experimentation : creating and taking down a 3000-node botnet. *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*.

[37] CALVET, J., FERNANDEZ, J. M., BUREAU, P.-M., MARION, J.-Y. *ET AL.* (2010). Large-scale malware experiments : Why, how, and so what ? *Proceedings of the 2010 Virus Bulletin Conference (VB)*.

[38] CALVET, J., FERNANDEZ, J. M. et MARION, J.-Y. (2012). Aligot : Cryptographic function identification in obfuscated binary programs. *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*.

[39] CIFUENTES, C. et SENDALL, S. (1998). Specifying the semantics of machine instructions. *Proceedings of the 6th IEEE International Workshop on Program Comprehension (IWPC)*.

[40] CIFUENTES, C., VAN EMMERIK, M., RAMSEY, N. et LEWIS, B. (2002). Experience in the design, implementation and use of a retargetable static binary translation framework. Rapport technique, Sun Microsystems Inc.

[41] ÉCOLE POLYTECHNIQUE DE MONTRÉAL (2011). Laboratoire de sécurité des systèmes d'information. `http://secsi.polymtl.ca/`. Consulté le 21 Mai 2013.

[42] COLLBERG, C. S. et THOMBORSON, C. (2002). Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering*, 28, 735–746.

[43] COLLINS, R. (2003). Undocumented opcodes : SALC. `http://www.rcollins.org/secrets/opcodes/SALC.html`. Consulté le 20 Février 2013.

[44] COMMUNITY, P. S. (2013). Polar SSL library Web site. `http://polarssl.org`. Consulté le 26 Mars 2013.

[45] COMPARETTI, P. M., SALVANESCHI, G., KIRDA, E., KOLBITSCH, C., KRUEGEL, C. et ZANERO, S. (2010). Identifying dormant functionality in malware programs. *Proceedings of the 31th IEEE Symposium on Security and Privacy (SP)*.

[46] COMPARETTI, P. M., WONDRACEK, G., KRUEGEL, C. et KIRDA, E. (2009). Prospex : Protocol specification extraction. *Proceedings of the 30th IEEE Symposium on Security and Privacy (SP)*.

[47] COOGAN, K. et DEBRAY, S. (2011). Equational reasoning on x86 assembly code. *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*.

[48] DAEMEN, J. et RIJMEN, V. (2002). *The design of Rijndael : AES–the advanced encryption standard*. Springer-Verlag.

[49] DAGON, D., GU, G., LEE, C. P. et LEE, W. (2007). A taxonomy of botnet structures. *Proceedings of the 23rd Anual Computer Security Applications Conference (ACSAC)*.

[50] DAVIS, C. R., FERNANDEZ, J. M. et NEVILLE, S. (2009). Optimising sybil attacks against p2p-based botnets. *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE)*.

[51] DAVIS, C. R., FERNANDEZ, J. M., NEVILLE, S. et MCHUGH, J. (2008). Sybil attacks as a mitigation strategy against the storm botnet. *Proceedings of the 3rd International Conference on Malicious and Unwanted Software (MALWARE)*.

[52] DEGENBAEV, U. (2012). *Formal Specification of the x86 Instruction Set Architecture*. Thèse de doctorat, Universitat des saarlandes.

[53] DINABURG, A., ROYAL, P., SHARIF, M. et LEE, W. (2008). Ether : malware analysis via hardware virtualization extensions. *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*.

[54] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J. et LEE, W. (2011). Virtuoso : Narrowing the semantic gap in virtual machine introspection. *Proceedings of the 32th IEEE Symposium on Security and Privacy (SP)*.

[55] DULLIEN, T. et PORST, S. (2009). REIL : A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest 2009*.

[56] EAGLE, C. (2008). *The IDA Pro Book : The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press.

[57] EMULAB (2013). Network emulation testbed home. `http://www.emulab.net/`. Consulté le 14 Avril 2013.

[58] ERNST, M. D. (2003). Static and dynamic analysis : Synergy and duality. *Proceedings of the 1st ICSE Workshop on Dynamic Analysis (WODA)*.

[59] ESET (2013). Virus radar. `http://www.virusradar.com/en`. Consulté le 23 Mai 2013.

[60] FALCON, F. et RIVA, N. (2012). Dynamic binary instrumentation frameworks : I know you're there spying on me. *Reverse Engineering Conference 2012 (REcon)*.

[61] FALLIERE, N. (2009). On moving register. `http://0x5a4d.blogspot.ca/2009/12/on-moving-register.html`. Consulté le 20 Février 2013.

[62] FALLIÈRE, N. (2010). Reversing trojan.mebroot's obfuscation. *Reverse Engineering Conference 2010 (REcon)*.

[63] FALLIERE, N. (2011). Sality : Story of a peer-to-peer viral network. Rapport technique, Symantec Corporation.

[64] FALLIERE, N., FITZGERALD, P. et CHIEN, E. (2009). Inside the jaws of trojan.clampi. Rapport technique, Symantec Corporation.

[65] FALLIERE, N., MURCHU, L. O. et CHIEN, E. (2011). Stuxnet dossier. Rapport technique, Symantec Corporation.

[66] FERRIE, P. (2008). Anti-unpacker tricks–part one. *Virus Bulletin Journal*, 4.

[67] GODEFROID, P., LEVIN, M. Y., MOLNAR, D. *ET AL.* (2008). Automated whitebox fuzz testing. *Proceedings of the 16th ISOC Network and Distributed System Security Symposium (NDSS)*.

[68] GOSTEV, A. (2009). The neverending story. `http://www.securelist.com/en/weblog?weblogid=208187654`. Consulté le 6 Juin 2013.

[69] GRÖBERT, F., WILLEMS, C. et HOLZ, T. (2011). Automated identification of cryptographic primitives in binary programs. *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*.

[70] GUILFANOV, I. (2006). FindCrypt2. `http://www.hexblog.com/?p=28`. Consulté le 31 Mars 2013.

[71] HALDERMAN, J., SCHOEN, S., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J., FELDMAN, A., APPELBAUM, J. et FELTEN, E. (2009). Lest we remember : cold-boot attacks on encryption keys. *Comm. of the ACM*, 52, 91–98.

[72] HANEL, A. (2013). Malware analysis search. `http://www.google.com/cse/home?cx=011750002002865445766%3Apc60zx1rliu`. Consulté le 27 Mai 2013.

[73] HENSON, S. *ET AL.* (2013). OpenSSL library. `http://openssl.org`. Consulté le 26 Mars 2013.

[74] HEX-RAYS (2013). IDA pro disassembler. `https://www.hex-rays.com/products/ida/index.shtml`. Consulté le 6 Juin 2013.

[75] HOPCROFT, J. E., MOTWANI, R. et ULLMAN, J. D. (2007). *Introduction to Automata Theory, Languages, And Computation*. Addison-Wesley.

[76] HUNGENBERG, T. et ECKERT, M. (2010). INetSim. `http://www.inetsim.org/index.html`. Consulté le 10 Mai 2013.

[77] IEEE (2011). IEEE software taggant system for exposing malware creators. `http://standards.ieee.org/news/2011/icsg_software.html`. Consulté le 06 Mai 2013.

[78] INTEL (2012). Software development emulator. `http://software.intel.com/en-us/articles/intel-software-development-emulator`. Consulté le 1 Mars 2013.

[79] INTEL (2012). XED2 user guide. `http://www.cs.virginia.edu/kim/publicity/pin/docs/20751/Xed/html/`. Consulté le 3 Mars 2013.

[80] INTEL (2013). Intel 64 and IA-32 Architectures Software Developer's Manuals.

[81] JACKSON, A. W., LAPSLEY, D., JONES, C., ZATKO, M., GOLUBITSKY, C. et STRAYER, W. T. (2009). SLINGbot : A system for live investigation of next generation botnets. *Proceedings of the Cybersecurity Applications & Technology Conference For Homeland Security (CATCH)*.

[82] JONES, N. D. (1997). *Computability and complexity : from a programming perspective*, vol. 21. The MIT Press.

[83] KASPERSKY (2013). 2012 by the numbers : Kaspersky Lab now detects 200,000 new malicious programs every day. `http://www.kaspersky.com/about/news/virus/2012/2012_by_the_numbers_Kaspersky_Lab_now_detects_200000_new_malicious_programs_every_day`. Consulté le 12 Mars 2013.

[84] KOBAYASHI, M. (1984). Dynamic characteristics of loops. *IEEE Trans. on Computers*, 100, 125–132.

[85] KOIVUNEN, T. (2013). SigBuster. Outil privé.

[86] KOLBITSCH, C., HOLZ, T., KRUEGEL, C. et KIRDA, E. (2010). Inspector gadget : Automated extraction of proprietary gadgets from malware binaries. *Proceedings of the 31th IEEE Symposium on Security and Privacy (SP)*.

[87] LEVIN, I. O. (2013). Draft crypto analyzer DRACA. `http://www.literatecode.com/draca`. Consulté le 26 Mars 2013.

[88] LIANG, Z., YIN, H. et SONG, D. (2008). HookFinder : Identifying and understanding malware hooking behaviors. *Department of Electrical and Computing Engineering*, 41.

[89] LIM, J. et REPS, T. (2008). A system for generating static analyzers for machine instructions. *Proceedings of the 17th International Conference on Compiler Construction (CC)*.

[90] LIN, Z., JIANG, X., XU, D. et ZHANG, X. (2008). Automatic protocol format reverse engineering through context-aware monitored execution. *Proceedings of the 15th ISOC Annual Network and Distributed System Security Symposium (NDSS)*.

[91] LIN, Z., ZHANG, X. et XU, D. (2010). Automatic reverse engineering of data structures from binary execution. *Proceedings of the 17th ISOC Network and Distributed System Security Symposium (NDSS)*.

[92] LINDHOLM, T. et YELLIN, F. (1999). *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc.

[93] LITZENBERGER, D. (2011). PyCrypto - the python cryptography toolkit.

[94] LORIA (2011). High Security Laboratory. `http://lhs.loria.fr/`. Consulté le 21 Mai 2013.

[95] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J. et HAZELWOOD, K. (2005). Pin : building customized program analysis tools with dynamic instrumentation. *Proceedings of the 26th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*.

[96] LUTZ, N. (2008). *Towards revealing attacker's intent by automatically decrypting network traffic*. Mémoire de maîtrise, ETH Zürich, Switzerland.

[97] MAARTMANN-MOE, C., THORKILDSEN, S. et ARNES, A. (2009). The persistence of memory : Forensic identification and extraction of cryptographic keys. *Digital Investigation*, 6, S132–S140.

[98] MANDIANT (2013). APT1 : Exposing one of china's cyber espionage units. `http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf`. Consulté le 12 Mars 2013.

[99] MARTIGNONI, L., CHRISTODORESCU, M. et JHA, S. (2007). Omniunpack : Fast, generic, and safe unpacking of malware. *Proceedings of the 23rd Anual Computer Security Applications Conference (ACSAC)*.

[100] MATENAAR, F., WICHMANN, A., LEDER, F. et GERHARDS-PADILLA, E. (2012). CIS : The crypto intelligence system for automatic detection and localization of cryptographic functions in current malware. *Proceedings of the 7th International Conference on Malicious and Unwanted Software (MALWARE)*.

[101] MICROSOFT (2010). Deactivating botnets to create a safer, more trusted internet. `http://www.microsoft.com/mscorp/twc/endtoendtrust/vision/botnet.aspx`. Consulté le 17 Avril 2013.

[102] MICROSOFT (2013). Malware families cleaned by the malicious software removal tool. `http://www.microsoft.com/security/pc-security/malware-families.aspx`. Consulté le 24 Mai 2013.

[103] MICROSOFT (2013). Microsoft collaborates with industry to disrupt conficker worm. `http://www.microsoft.com/en-us/news/press/2009/feb09/02-12ConfickerPR.aspx`. Consulté le 6 Juin 2013.

[104] MIRKOVIC, J., BENZEL, T. V., FABER, T., BRADEN, R., WROCLAWSKI, J. T. et SCHWAB, S. (2010). The DETER project : Advancing the science of cyber security experimentation and test. *Proceedings of the 2010 IEEE International Conference on Technologies for Homeland Security (HST)*.

[105] MONTGOMERY, P. (1985). Modular multiplication without trial division. *Mathematics of Computation*, 44, 519–521.

[106] MORGENSTERN, M. et PILZ, H. (2010). Useful and useless statistics about viruses and anti-virus programs. *Proceedings of the 2010 CARO Workshop*.

[107] MOSER, A., KRUEGEL, C. et KIRDA, E. (2007). Exploring multiple execution paths for malware analysis. *Proceedings of the 28th IEEE Symposium on Security and Privacy (SP)*.

[108] NETHERCOTE, N. et SEWARD, J. (2007). Valgrind : a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42, 89–100.

[109] NIELSON, F., NIELSON, H. R. et HANKIN, C. (2004). *Principles of program analysis*. Springer.

[110] NIELSON, H. et NIELSON, F. (2007). *Semantics with applications : an appetizer*. Springer.

[111] NUNNERY, C., SINCLAIR, G. et KANG, B. B. (2010). Tumbling down the rabbit hole : exploring the idiosyncrasies of botmaster systems in a multi-tier botnet infrastructure. *Proceedings of the 2010 USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*.

[112] O MURCHU, L. (2013). Trojan.silentbanker decryption. `http://www.symantec.com/connect/blogs/trojansilentbanker-decryption`. Consulté le 26 Mars 2013.

[113] OBERHUMER, M., MOLNÁR, L. et REISER, J. F. (2004). UPX : Ultimate Packer for eXecutables. `http://upx.sourceforge.net/`. Consulté le 20 Février 2013.

[114] PALEARI, R., MARTIGNONI, L., FRESI ROGLIA, G. et BRUSCHI, D. (2010). N-version disassembly : differential testing of x86 disassemblers. *Proceedings of the 19th ACM International Symposium on Software Testing and Analysis (ISSTA)*.

[115] RASCAGNERES, P. (2013). Malware.lu : base de logiciels malveillants. `http://malware.lu/`. Consulté le 10 Mai 2013.

[116] REYNAUD, D. (2010). *Analyse de codes auto-modifiants pour la sécurité logicielle*. Thèse de doctorat, Nancy-Université.

[117] RIVEST, R. (1992). RFC 1321 : The MD5 message-digest algorithm. *Internet Activities Board*, 143.

[118] RIVEST, R. (2013). RC4 source code. `http://cypherpunks.venona.com/date/1994/09/msg00304.html`. Consulté le 26 Mars 2013.

[119] RIVEST, R., SHAMIR, A. et ADLEMAN, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Comm. of the ACM*, 21, 120–126.

[120] ROSSOW, C., ANDRIESSE, D., WERNER, T., STONE-GROSS, B., PLOHMANN, D., DIETRICH, C. J. et BOS, H. (2013). P2PWNED : Modeling and evaluating the resilience of peer-to-peer botnets. *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)*.

[121] ROSSOW, C., DIETRICH, C. et BOS, H. (2012). Large-scale analysis of malware downloaders. *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[122] ROUNDY, K. A. et MILLER, B. P. (2012). Binary-code obfuscations in prevalent packer tools. `http://ftp.cs.wisc.edu/par-distr-sys/papers/Roundy12Packers.pdf`. Consulté le 18 Juin 2013.

[123] ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R. et LEE, W. (2006). Polyunpack : Automating the hidden-code extraction of unpack-executing malware. *Proceedings of the 22nd Anual Computer Security Applications Conference (ACSAC)*.

[124] SARKAR, S., SEWELL, P., NARDELLI, F. Z., OWENS, S., RIDGE, T., BRAIBANT, T., MYREEN, M. O. et ALGLAVE, J. (2009). The semantics of x86-CC multiprocessor machine code. *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*. vol. 44, 379–391.

[125] SCHWARTZ, E. J., AVGERINOS, T. et BRUMLEY, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *Proceedings of the 31th IEEE Symposium on Security and Privacy (SP)*.

[126] SCHWARZ, B., DEBRAY, S. et ANDREWS, G. (2002). Disassembly of executable code revisited. *Proceedings of the 9th IEEE Working Conference on Reverse Engineering (WCRE)*.

[127] SHANNON, C. E. (1949). Communication theory of secrecy systems. *Bell System Technical Journal*, 28, 656–715.

[128] SLON (2013). Russian TEA assembly code. `http://www.xakep.ru/post/22086/default.asp`. Consulté le 26 Mars 2013.

[129] SLOWINSKA, A., STANCESCU, T. et BOS, H. (2011). Howard : a dynamic excavator for reverse engineering data structures. *Proceedings of the 18th ISOC Network and Distributed System Security Symposium (NDSS)*.

[130] SNAKER, QWERTON et JIBZ (2013). PEiD. `http://www.peid.info`. Consulté le 13 Mars 2013.

[131] SNAKER, QWERTON et JIBZ (2013). PEiD krypto analyzer KANAL. `http://www.peid.info`. Consulté le 26 Mars 2013.

[132] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P. et SAXENA, P. (2012). BitBlaze : Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu`. Consulté le 7 Mars 2013.

[133] STARFORCETECHNOLOGIES (2013). AsProtect packer. `http://www.aspack.com/asprotect.html`. Consulté le 26 Mars 2013.

[134] STEWART, N. (2008). Inside the storm : Protocols and encryption of the Storm botnet. *Black Hat USA 2008*.

[135] STOCK, B., GOBEL, J., ENGELBERTH, M., FREILING, F. C. et HOLZ, T. (2009). Walowdac-analysis of a peer-to-peer botnet. *Proceedings of the 5th European Conference on Computer Network Defense (EC2ND)*.

[136] STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDLOWSKI, M., KEMMERER, R., KRUEGEL, C. et VIGNA, G. (2009). Your botnet is my botnet : analysis of a botnet takeover. *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*.

[137] STONE-GROSS, B., HOLZ, T., STRINGHINI, G. et VIGNA, G. (2011). The underground economy of spam : A botmaster's perspective of coordinating large-scale spam campaigns. *Proceedings of the 2011 USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*.

[138] STRELITZIA (2013). ExeForger : SignsImitator, 1.0.40.10. `http://www.at4re.com/download.php?view.18`. Consulté le 26 Mars 2013.

[139] TUBELLA, J. et GONZÁLEZ, A. (1998). Control speculation in multithreaded processors through dynamic loop detection. *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*.

[140] VAN EMMERIK, M. *ET AL.* (2002). Boomerang decompiler homepage. `http://boomerang.sourceforge.net/`. Consulté le 20 Février 2013.

[141] VAN RUITENBEEK, E. et SANDERS, W. H. (2008). Modeling peer-to-peer botnets. *Proceedings of the 5th IEEE International Conference on Quantitative Evaluation of Systems (QEST)*.

[142] VERISIGN (2013). Silentbanker analysis. `http://www.verisign.com/static/043671.pdf`. Consulté le 26 Mars 2013.

[143] VILLENEUVE, N., DEIBERT, R. et ROHOZINSKI, R. (2010). Koobface : Inside a crimeware network. Rapport technique, Munk School of Global Affairs.

[144] VMWARE (2013). VMware vSphere ESX and ESXi info center. `http://www.vmware.com/ca/en/products/datacenter-virtualization/vsphere/esxi-and-esx/overview.html`. Consulté le 6 Juin 2013.

[145] VON NEUMANN, J. (1993). First Draft of a Report on the EDVAC. *Annals of the History of Computing, IEEE*, 15, 27–75.

[146] VUKSAN, M. et PERICIN, T. (2010). File analysis and unpacking in the age of 40m new samples per year. *Proceedings of the 2010 CARO Workshop*.

[147] WANG, C., HILL, J., KNIGHT, J. et DAVIDSON, J. (2000). Software tamper resistance : Obstructing static analysis of programs. Rapport technique CS-2000-12, University of Virginia.

[148] WANG, Z., JIANG, X., CUI, W., WANG, X. et GRACE, M. (2009). ReFormat : Automatic reverse engineering of encrypted messages. *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*.

[149] WANG, Z. J. (2010). Virtual machine protection technology and av industry. *Proceedings of the 2010 CARO Workshop*.

[150] WARTELL, R., ZHOU, Y., HAMLEN, K., KANTARCIOGLU, M. et THURAISINGHAM, B. (2011). Differentiating code from data in x86 binaries. *Proceedings of the 2011 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*.

[151] WHEELER, D. et NEEDHAM, R. (1995). TEA, a tiny encryption algorithm. *Proceedings of the 2nd International Workshop on Fast Software Encryption (FSE)*.

[152] WICHERSKI, G. *ET AL.* (2008). Stormfucker : Owning the storm botnet. *25th Chaos Communication Congress (CCC)*.

[153] XCAT (2013). Extreme cloud administration toolkit (xCAT). `http://xcat.sourceforge.net/`. Consulté le 15 Avril 2013.

[154] YARA (2013). A malware identification and classification tool. `https://code.google.com/p/yara-project/`. Consulté le 13 Mars 2013.

[155] ZAKORZHEVSKY, V. (2013). A new version of sality at large. `http://www.securelist.com/en/blog/180/A_new_version_of_Sality_at_large`. Consulté le 26 Mars 2013.

[156] ZAYTSEV, V., KARNIK, A. et PHILLIPS, J. (2010). Parasitics : The next generation. *Proceedings of the 19th EICAR Annual Conference*.

[157] ZHAO, R., GU, D., LI, J. et YU, R. (2011). Detection and analysis of cryptographic data inside software. *Information Security*, 7001, 182–196.

ANNEXE A

Liste de mnémoniques x86 standards

ADC, ADD, AND, CALL, CDQ, CLD, CMP, CMPS,
CMPXCHG, DEC, DIV, FSUB, IDIV, IMUL, INC, JA,
JAE, JB, JBE, JG, JGE, JL, JLE, JMP, JNB, JNO, JNS, JNZ,
JO, JP, JS, JZ, LEA, LEAVE, LOOP, MOV, MOVS, MOVSB,
MOVSD, MOVSW, MOVSX, MOVZX, MUL , NEG, NOP, NOT, OR,
POP, POPA, PUSH, PUSHA, PUSHF, RET, RETN, ROL, ROR,
SAR, SBB, SETB, SETL, SETLE, SETNL, SETNLE, SETNZ, SETZ,
SHL, SHR, SHRD, STOS, STOSB, STOSD, STOSW, SUB, TEST, XADD,
XCHG, XOR

ANNEXE B

Écarts types des expériences avec Waledac

Dans cette annexe nous présentons les écarts types pour les trois mesures de l'efficacité de notre attaque contre Waledac.

#	Déviation standard pour le nombre de spammeurs qui contactent le serveur SMTP chaque 5 minute après le début de lattaque																					
5	3.8	12.9	27.7	64.4	20.5	58.5	26.9	39.8	12.1	5.3	14.1	18.1	6.1	13.4	26.5	24.5	17.9	16.3	24.4	13.8	15.9	12.1
10	2.6	48.4	64.7	5.6	41.4	16.2	21.4	5.7	16.0	11.3	10.9	19.1	13.1	20.5	17.7	16.0	13.5	13.1	5.8	2.8	3.7	15.0
25	3.0	60.5	61.5	55.8	31.4	46.0	38.0	34.6	19.3	13.6	15.1	7.2	9.7	12.0	10.2	6.7	2.4	8.4	10.8	11.7	6.0	11.1
100	8.9	125.8	127.0	9.8	20.3	17.7	14.5	6.4	20.6	6.9	9.6	7.2	15.7	6.5	11.9	7.4	9.4	4.6	8.8	10.0	7.6	4.1

#	Déviation standard pour le nombre de messages NOTIFY reçus par le CC noir chaque 5 minute après le début de lattaque																					
5	11.5	83.5	14.5	86.3	16.3	84.3	19.8	57.3	9.5	39.6	20.7	28.4	8.1	24.2	14.3	20.8	15.8	12.8	16.8	26.6	13.5	22.3
10	50.2	121.0	45.1	123.3	46.4	121.5	37.5	70.1	17.4	41.2	18.8	29.9	11.1	23.5	13.8	24.8	11.9	28.2	11.8	25.4	11.4	22.4
25	64.2	92.9	63.4	95.1	62.2	95.2	38.4	26.0	16.9	23.8	17.5	19.8	15.3	23.4	15.5	21.8	12.2	16.3	13.6	19.5	13.2	17.6
100	97.2	96.6	95.2	99.3	93.8	100.8	52.7	21.3	6.1	11.5	3.8	9.0	7.5	5.5	6.2	4.5	7.4	6.9	10.1	7.0	7.5	7.7

#	Déviation standard pour le pourcentage de SR dans les listes de pairs chaque 5 minute après le début de lattaque																							
5	0.0	2.0	6.5	1.4	1.4	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
10	0.0	1.0	1.4	1.4	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
25	1.7	3.8	1.0	1.4	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	3.0	2.0	1.4	1.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

RÉSUMÉ : ANALYSE DYNAMIQUE DE LOGICIELS MALVEILLANTS

Mots clés : logiciel malveillant, analyse dynamique, virus informatique.

L'objectif de cette thèse est le développement de méthodes de compréhension des logiciels malveillants, afin d'aider l'analyste humain à mieux appréhender cette menace.

La première réalisation de cette thèse est une analyse à grande échelle et en profondeur des protections de logiciels malveillants. Plus précisément, nous avons étudié des centaines d'exemplaires de logiciels malveillants, soigneusement sélectionnés pour leur dangerosité. En mesurant de façon automatique un ensemble de caractéristiques originales, nous avons pu alors montrer l'existence d'un modèle de protection particulièrement prévalent dans ces programmes, qui est basé sur l'auto modification du code et sur une limite stricte entre code de protection et code utile.

Ensuite, nous avons développé une méthode d'identification d'implémentations cryptographiques adaptée aux programmes en langage machine protégés. Nous avons validé notre approche en identifiant de nombreuses implémentations d'algorithmes cryptographiques – dont la majorité sont complètement invisibles pour les outils existants –, et ceci en particulier dans des protections singulièrement obscures de logiciels malveillants.

Finalement, nous avons développé ce qui est, à notre connaissance, le premier environnement d'émulation de réseaux de machines infectées avec plusieurs milliers de machines. Grâce à cela, nous avons montré que l'exploitation d'une vulnérabilité du protocole pair-à-pair du réseau *Waledac* permet de prendre son contrôle.

ABSTRACT : DYNAMIC ANALYSIS OF MALICIOUS SOFTWARE

Keywords : malicious software, dynamic analysis, computer virus.

The main goal of this thesis is the development of malware analysis methods to help human analysts better comprehend the threat it represents.

The first achievement in this thesis is the large-scale and in-depth analysis of malware protection techniques. In particular, we have studied hundreds of malware samples, carefully selected according to their threat level. By automatically measuring a set of original characteristics, we have been able to demonstrate the existence of a particularly prevalent model of protection in these programmes that is based on self-modifying code and on a strict delimitation between protection code and payload code.

Then, we have developed an identification method for cryptographic implementations adapted to protected machine language programmes. We have validated our approach by identifying several implementations of cryptographic algorithms —the majority unidentified by existing tools— and this even in particularly obscure malware protection schemes.

Finally, we have developed what is, to our knowledge, the first emulation environment for botnets involving several thousands of machines. Thanks to this, we were able to validate the viability of the use of a vulnerability in the peer-to-peer protocol in the *Waledac* botnet to take over this network.