Étude du métamorphisme viral : modélisation, conception et détection

Jean-Marie Borello

N° d'ordre : ANNÉE 2011

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : INFORMATIQUE

École doctorale MATISSE

présentée par

Jean-Marie Borello

préparée à l'unité de recherche EA 4039 (SSIR)
Sécurité des Systèmes d'Information et Réseaux
SUPÉLEC

Étude du métamorphisme viral : modélisation, conception et détection

Thèse soutenue à Supélec
le 1er avril 2011
devant le jury composé de :

Thomas Jensen
Directeur de recherche à l'INRIA / président

Guillaume Bonfante
Maître de conférence à l'Institut National Polytechnique de Lorraine / rapporteur

Jean-Marc Steyaert
Professeur à l'école Polytechnique / rapporteur

Marc Dacier
Senior director in charge of the Collaborative Advanced Research Department (CARD) within Symantec Research Labs / examinateur

Loïc Duflot
Docteur à l'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) / examinateur

Frédéric Valette
Ingénieur au centre DGA Maîtrise de l'Information de la Direction Générale de l'Armement / invité

Ludovic Mé
Professeur à Supélec / directeur de thèse

Éric Filiol
Professeur à l'ESIEA / co-directeur de thèse

Sommaire

Table des figures

Liste des tableaux

Liste des programmes et des algorithmes

Remerciements

Introduction

À l'origine réservés aux infrastructures critiques pour lesquelles l'échange et la rapidité d'accès aux informations constituent une composante essentielle, les systèmes d'informations se sont répandus avec le développement de l'informatique personnelle. Aujourd'hui, l'omniprésence de ces systèmes dans des domaines aussi variés que la santé, les télécommunications, les transports et les organisations gouvernementales, les rend particulièrement critiques. Leur interconnexion ainsi que la rapidité d'échange des informations qu'ils véhiculent permettent une dématérialisation des actions qui en font une proie toute désignée pour des utilisateurs mal intentionnés. La moindre défaillance de ces systèmes peut alors directement impacter la stabilité d'un pays, aussi bien sur les plans économiques, énergétiques, financiers que politiques. À titre d'exemple, la plus lourde attaque en dénis de service distribué (« *Distributed Denial of Service* » ou DDoS) jamais observée en Europe a eu lieue le 27 avril 2007 [110]. Cette attaque, ciblant l'Estonie, a paralysé pendant plusieurs jours son administration, ses banques ainsi qu'une bonne partie de ses médias. Cet exemple illustre l'enjeu que représente la sécurité des systèmes d'information.

Conformément à la définition des critères de l'« *Information Technology Security Evaluation Criteria* » (ITSEC) [86], garantir la sécurité des systèmes d'information consiste à assurer trois propriétés sur les informations manipulées :

- la confidentialité, les informations ne doivent pas être révélées aux utilisateurs non autorisés ;
- l'intégrité, les informations ne doivent pas être modifiés par des utilisateurs non autorisés ou par suite d'erreurs ;
- la disponibilité, les informations doivent être accessibles à tout utilisateur autorisé, en toute circonstances.

Les actions menées par des utilisateurs malveillants dans le but de porter atteinte à la sécurité d'un système sont communément désignées sous le terme d'attaques informatique. Ces attaques, peuvent être conduites de deux manières :

- les attaques dites *manuelles* sont menées par un être humain qui, par le biais d'un ensemble de programmes s'exécutant sous son identité, lui permettent de porter atteinte au système ;
- la seconde manière de conduire une attaque consiste à l'automatiser. Dans ce cas, l'attaque est menée de manière autonome par des logiciels désignés

sous l'expression de codes malveillants.
L'expression code malveillant provient de la traduction littérale du terme anglophone « *malware* », néologisme obtenu par la contraction du terme « *malicious* » signifiant malveillant [1] et du terme « *software* » désignant un logiciel. Cette parenthèse étymologique nous permet de soulever le problème de la terminologie liée au domaine des codes malveillants. Bien que plusieurs organismes tels que le « *Computer Antivirus Research's Organization* » (CARO), le « *Common Malware Enumeration initiative* » (CME) et l'« *European Institute for Computer Antiviral Research* » (EICAR) tentent d'harmoniser le vocabulaire associé au domaine de la lutte contre les codes malveillants, certaines définitions ne sont pas encore unanimement admises. Même en cas de consensus, trouver des équivalents français précis et explicites aux termes anglophones consacrés est parfois une tâche difficile. Tout au long de ce mémoire, un soin particulier sera apporté à la définition des termes employés. L'emploi des termes anglais sera réservé au cas de non existence d'équivalents français officiels ou lorsque la définition française ne nous semble pas suffisamment précise et explicite.

Le panorama des codes malveillants apparait aujourd'hui riche et varié comme en témoignent les derniers rapports de sécurité de Microsoft (« *Microsoft Security Intelligence Report* » (SIR)) [123, 124, 125, 126]. Ce panorama actuel reflète l'évolution rapide des codes malveillants depuis les débuts de l'ère informatique, en réponse aux évolutions des outils employés pour les détecter.

Les premiers travaux en lien avec les codes malveillants traitent d'un mécanisme fondamental intervenant dans l'évolution biologique : l'auto-reproduction. C'est ainsi que les travaux précurseurs de von Neumann [169, 170] et de Burks [23], portant sur la théorie des automates cellulaires et visant à modéliser de manière simplifiée l'auto-reproduction, introduisent une base théorique pour les premiers codes malveillants connus. À partir de ces résultats, confortés par la théorie des fonctions récursives de Kleene [96], et notamment par le théorème de récursion, preuve est alors faite qu'il est possible de concevoir des programmes dont l'exécution produit leur propre code.

En application de ces travaux théoriques, les premières implémentations de programmes auto-reproducteurs à caractère malveillant, historiquement désignées sous l'appellation « virus informatiques » par Cohen [43], sont apparues dans les années 80. En même temps ont été initiées les premières recherches académiques traitant de ce sujet avec les travaux de Kraus [102] en 1980 et ceux de Cohen [43] en 1986. Parmi les premiers virus découverts, les plus célèbres sont Elk Cloner tournant sous AppleDOS 3.3, apparu en 1983, ainsi que le virus d'amorce de disque Brain, découvert en 1986 [82].

Peu à peu, ces premiers programmes ont progressivement laissé la place à d'autres types de codes malveillants. L'utilisation des réseaux comme moyen de propagation est apparu avec le programme Xerox conçu en 1981. Bien que

1. Le terme malicieux est parfois employé comme équivalent français. Dans ce cas, il correspond à sa définition première selon le dictionnaire de la langue française d'Émile Littré [116] à savoir, une inclinaison à malfaire. Pour lever toute ambiguïté, nous utiliserons uniquement comme traduction le terme « malveillant ».

son origine semble plus accidentelle que volontaire, ce programme constitue le premier ver connu. L'auto-reproduction se fait alors de machine en machine par le biais du réseau et non plus par infection d'autres programmes comme pour le cas des virus. Le ver MORRIS [156] est le premier code malveillant publique à se propager à travers l'Internet. Le début des années 2000 est marqué par l'émergence des vers à propagation rapide tels que CODE RED en 2001 [27] et SLAMMER en 2003 [127]. Avec l'essor de l'Internet, ces programmes se sont particulièrement développés pour toucher un maximum de machines le plus rapidement possible. Par exemple, la deuxième version de CODE RED a réussi à infecter, en moins de 24 heures, plus de 350 000 serveurs dans le monde [194]. La vitesse de propagation de SLAMMER a été évaluée au double de celle de CODE RED.

Le nombre de machines connectées à Internet a aussi fortement influencé l'évolution de la menace. Sont alors apparus des logiciels espions (« *spyware* ») [130], des réseaux de zombies (« *botnets* ») [77], des logiciels visant à extorquer de l'argent (« *ransomware* ») [19, 70].

Les codes malveillants représentent aujourd'hui une menace sérieuse pesant sur l'informatique moderne. Symantec évalue à 5 millions le nombre de machines compromises impliquées dans des réseaux de zombies pour l'année 2008. Panda Security estime à 10 millions le nombre de machines infectées par des codes malveillants en 2009. D'un point de vue économique, les dégâts imputés à cette menace ont été évalués 9,3 milliards d'euros en Europe entre 2007 et 2008 [128] .

Avec l'avènement des premiers codes malveillants, sont aussi apparus les premières variations de ces codes, désignées sous le nom de variantes (de codes malveillants). Pour notre propos, une variante est *un programme qui partage une quantité de code suffisante avec un programme d'origine P, appelé programme souche, pour être considéré comme apparenté à P*. Cette forme de diversification a pour objectif de contourner les systèmes de détection et principalement ceux à base de signatures. Si, à l'origine, la génération de variantes résultait essentiellement de modifications manuelles, la situation actuelle est tout autre. En effet, ces dernières années témoignent d'une augmentation significative du nombre de variantes de codes malveillants connus d'après Microsoft : au cours de la seconde moitié de l'année 2008, 95 millions de fichiers à caractère malveillant ont été répertoriés [125]. Six mois plus tard, ce nombre s'élevait alors à 116 millions [124] pour finalement atteindre 126 millions à la fin de l'année 2009 [126]. Devant cette augmentation, la lutte contre la prolifération des variantes est devenue un enjeu majeur.

Afin de produire de nouvelles variantes de codes malveillants à partir d'une souche originale, les attaquants ont recours à des techniques de mutation de code. Ces techniques de mutation peuvent être réalisées soit avant la mise en « service » du code malveillant, soit après. Dans le premier cas, on parle de génération « hors-ligne » (« *off-line* »). Les variantes sont alors obtenues par des outils (« *kits* ») de génération automatique de codes malveillants ou alors par des outils de protection de code (« *packers* »). Dans le second cas, la mutation a lieu à chaque réplication dans le but de produire un code différent. On

parle alors de génération « en-ligne » (« *in-line* »). De notre point de vue, le cas des mutations « en-ligne » , c'est-à-dire celui des codes auto-reproducteurs mutants, représente une plus grande menace. Reconsidérons à titre d'exemple le cas de CODE RED. Supposons que nous disposions de n variantes de ce même ver produites « hors-ligne » . Le nombre de variantes à détecter correspond alors, parmi les n générées, aux m versions qui ont effectivement été « mises en service » (en l'occurrence, celles qui ont été introduites sur l'Internet). Supposons maintenant que ce ver soit capable de muter lors de chaque réplication. Dans ce cas, en moins de 24 heures, ce sont 350 000 variantes qui sont obtenues à partir d'une souche initiale introduite en un seul nœud du réseau.

Les codes capables de modifier leurs formes peuvent être classés en codes chiffrés, polymorphes, et enfin métamorphes [158]. Nous les présentons de manière informelle, par ordre d'apparition. Cet ordre correspond aussi à celui de la complexité des techniques mises en œuvre :

– le **chiffrement**. Il s'agit de la première technique historiquement employée afin de contourner la détection dite par signature. Cette détection par signature consiste à identifier un motif au sein d'un programme. Un motif peut se définir de diverses façons, des formes les plus simples comme une expression régulière sur la syntaxe d'un programme jusqu'à des formes complexes décrivant le comportement d'un programme. Dans le cas du chiffrement de code, la technique de détection ciblée est celle par signature statique portant sur le binaire d'un programme. Le chiffrement de code comporte au minimum trois parties distinctes : une routine de déchiffrement D, une charge « utile » U et enfin, une routine de chiffrement E. La figure 1 présente le fonctionnement d'un code chiffré auto-reproducteur simple [2] note C. Avant son exécution (1), le programme C se présente

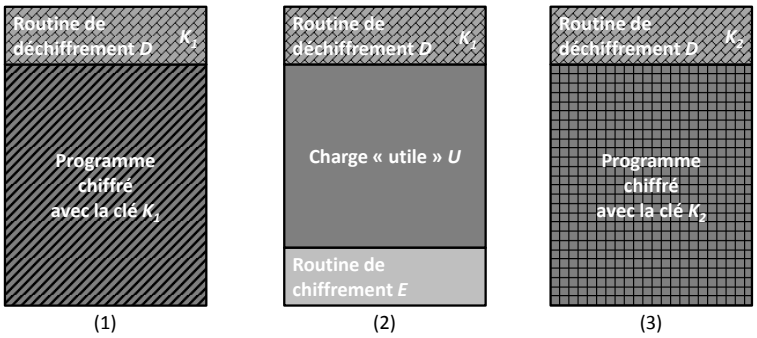


FIGURE 1 – Fonctionnement d'un code auto-reproducteur chiffré.

sous la forme d'une routine de déchiffrement D contenant la clé K_1 per-

2. On remarquera que le processus de réplication d'un code chiffré, tel que présenté en figure 1, peut être plus complexe. En effet, il est possible de l'imbriquer en cascade afin d'obtenir des codes plus élaborés.

mettant de déchiffrer le reste du programme. Après exécution de D (2), C est alors intégralement déchiffré, et peut alors exécuter directement le reste de son code, c'est-à-dire la charge « utile » U et la routine de chiffrement E. Une fois U, la routine E génère une nouvelle clé K_2. Cette clé est ensuite utilisée pour chiffrer U et E avant d'être insérée dans la routine de chiffrement D. Une nouvelle instance du code chiffré notée C' est alors obtenue (3). Chaque instance étant chiffrée avec une clé générée aléatoirement, le corps du programme chiffré ne présente alors pas de motif syntaxique permettant sa détection ;

– le **polymorphisme**. Bien que le corps d'un programme chiffré change constamment de formes à chaque réplication, la routine en charge du déchiffrement demeure constante, d'une copie à l'autre. Aussi, cette routine de déchiffrement constitue naturellement un motif possible de détection portant sur l'image statique du programme C. C'est afin d'éviter la pré-

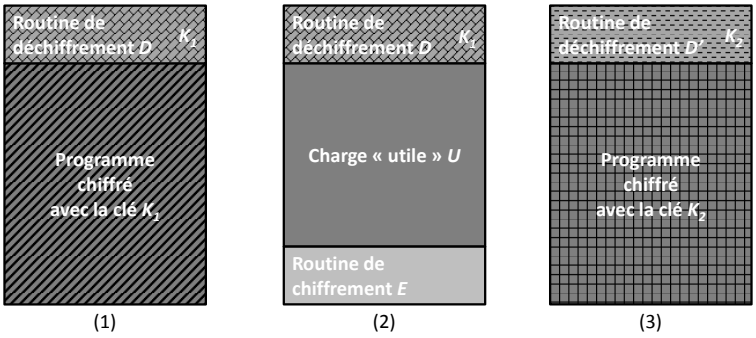


FIGURE 2 – Illustration du fonctionnement d'un code polymorphe simple.

sence d'un tel motif de détection que le polymorphisme est né. Il consiste à faire évoluer la syntaxe de la routine de déchiffrement D au moyen de techniques de mutations de code. Le fonctionnement d'un code polymorphe est illustré en figure 2. Le processus de réplication d'apparente à celui d'un code auto-reproducteur chiffré (étapes (1) et (2)). Sauf que cette fois ci, une nouvelle routine de déchiffrement D' est générée de sorte que D' et D soit syntaxiquement différentes.

– le **métamorphisme**. Le polymorphisme peut être déjoué par émulation de la routine de déchiffrement. De manière simplifiée, l'émulation consiste à imiter l'exécution d'un programme au moyen d'un processeur (« *Central Processor Unit* » ou CPU) logiciel. Ainsi, l'émulation permet effectivement de déchiffrer la charge « utile » U d'un programme polymorphe P. Une fois déchiffré, U constitue un motif de détection présent en mémoire [3]. C'est afin d'éviter la présence d'un motif syntaxique constant, à tout mo-

3. On remarquera que l'émulation permet aussi de déchiffrer en mémoire un code auto-reproducteur chiffré, ce qui autorise le même type de détection que dans le cas du polymorphisme.

> ment de son exécution, que le métamorphisme est né. Historiquement, le métamorphisme consiste à faire muter l'intégralité du code s'exécutant et non pas uniquement la routine de déchiffrement comme dans le cas du polymorphisme. C'est pour cette raison que le métamorphisme est parfois considéré comme du « polymorphisme de corps » [159].

Ces définitions, issues des observations de codes existants, restent empiriques et nécessitent une étude plus approfondie que nous nous proposons de mener dans le cas du métamorphisme en défendant la thèse suivante : il est possible de créer des codes métamorphes dont la détection statique est *prouvée difficile*, et pour lesquels les anti-virus actuels ne peuvent fournir une détection à la fois *fiable* et *pertinente*. Nous proposons toutefois dans ce mémoire une approche de détection dynamique exploitant le fait que toutes les variantes d'une même souche métamorphe présentent un fort degré de similarité dans leurs comportements.

Afin d'étayer cette thèse, ce mémoire s'organise de la manière suivante :

Le chapitre 1 présente un état de l'art sur le métamorphisme. Partant de l'étymologie de ce terme, nous proposons une première définition informelle du métamorphisme. Cette première définition permet d'aborder les aspects essentiels qui sont développés dans la suite de ce chapitre : les formalismes des codes malveillants permettant d'aboutir aux différentes définitions existantes du métamorphisme, les techniques de mutation de code ainsi que le fonctionnement global de tels programmes, et enfin les approches de détection employées.

La suite du mémoire s'organise en deux parties. Dans une première partie, nous présentons la conception d'un moteur générique de métamorphisme fondée sur une approche d'obscurcissement de code (en anglais « *obfuscation* ») à résilience prouvée dans le cadre de l'analyse statique. Cette première partie se compose des chapitres 2 et 3, qui présentent la conception, l'implémentation et l'utilisation de notre moteur de métamorphisme.

Le chapitre 2 propose une approche d'obscurcissement de code utilisable dans le cadre de codes malveillants métamorphes. Cette approche s'appuie sur un modèle théorique permettant de prouver l'efficacité des transformations utilisées dans le cadre de l'analyse statique de programmes. Cette hypothèse d'analyse statique est ensuite expérimentalement vérifiée sur un cas réel. La contribution de ce chapitre consiste à montrer que des techniques d'obscurcissement de code avancée peuvent effectivement être employées pour des codes malveillants métamorphes.

Le chapitre 3 décrit l'implémentation d'un moteur générique de métamorphisme s'appuyant sur le modèle théorique du chapitre précédent. Ce moteur est appliqué sur un code malveillant connu afin d'en obtenir une version métamorphe. Le programme résultant est soumis à un panel d'outils de détection représentatif de l'état de l'art industriel. Les résultats obtenus permettent de mettre à jour les techniques de détection employées par les antivirus « grand public » tout en montrant l'efficacité de notre moteur de métamorphisme. La

contribution de ce chapitre est double. D'une part, nous montrons expérimentalement que notre approche syntaxique est efficace par rapport aux outils de détection anti-viraux actuels. D'autre part, cette approche permet une évaluation « boîte noire » des techniques observables de détection employées par ces anti-virus.

Dans une seconde partie nous présentons, une approche de détection dynamique s'appuyant sur la complexité de Kolmogorov pour mesurer la similarité entre profils comportementaux. Cette seconde partie comprend les chapitres 4 et 5, qui exposent notre approche de détection dynamique de codes malveillants fondée sur la complexité de Kolmogorov.

Le chapitre 4 introduit une nouvelle mesure de similarité fondée sur la complexité de Kolmogorov. Les différentes approches concernant la quantification de l'information sont abordées, à savoir la théorie de Shannon ainsi que l'approche algorithmique de Kolmogorov. Deux mesures de similarités y sont présentées d'un point de vue théorique. La première, la distance normalisée de compression (« *Normalized Compression Distance* » ou NCD), déjà largement utilisée dans le domaine de la classification [41]. La seconde, le degré d'inclusion mutuelle par compression (« *Compression based Mutual Inclusion Degree* » ou CMID) est une nouvelle mesure que nous proposons. Elle permet d'évaluer le degré d'inclusion en termes d'information entre deux objets. La contribution de ce chapitre est de démontrer dans quelles conditions sur le compresseur utilisé cette mesure correspond effectivement à la notion de degré d'inclusion [191].

Le chapitre 5 propose une approche de détection dynamique adaptée non seulement aux codes métamorphes, mais aussi aux codes malveillants, indépendamment des transformations syntaxiques qu'ils emploient. Notre prototype s'appuie sur les deux mesures de similarité présentées au chapitre précédent. Le principe de détection repose sur la similarité comportementale entre programmes. Cet prototype est finalement évalué sur des variantes de codes malveillants. Ce chapitre présente une contribution dans le domaine de la détection de codes malveillants en proposant une approche fondée sur la similarité comportementale obtenue au moyen d'un algorithme de compression sans perte.

Enfin, ce mémoire se conclut en dressant un bilan du travail réalisé et de ses contributions, tout en proposant des perspectives de recherches ouvertes par ce travail.

Chapitre 1

État de l'art

Le métamorphisme est issu de l'évolution des codes malveillants dans le but d'échapper aux outils de détection auxquels ils furent confrontés. Ce terme, d'origine grecque, se compose du préfixe `méta` (μετά) signifiant « au-delà, après », ainsi que du suffixe `morph` (μορφή) signifiant « forme ». En accord avec ses racines étymologiques, le métamorphisme peut, en première approche, se définir comme une *technique d'auto-reproduction durant de laquelle le programme métamorphe en cours d'exécution produit une réplique mutée de son propre code dans le but d'éviter d'être détecté par un autre programme, l'anti-virus*. Bien qu'approximative, cette première définition met en évidence trois aspects du métamorphisme : les formalismes liés aux programmes auto-reproducteurs, les techniques de modification (mutation) de code, et enfin les approches de détection.

Cet état de l'art contient donc trois parties. Dans un premier temps, nous présentons les différents modèles de l'auto-reproduction qui ont permis d'aboutir aux définitions du métamorphisme. L'avantage de ces formalismes est double. D'une part, ils permettent d'apporter des définitions précises au métamorphisme. D'autre part, ces modèles mathématiques permettent de définir la complexité du problème de la détection de tels codes. Dans un second temps, nous abordons le métamorphisme d'un point de vue plus pratique en considérant les techniques intervenant dans le cadre de la mutation de code. Finalement dans un dernier temps, nous étudierons les différentes approches pratiques de détection des codes métamorphes.

1.1 Vers un métamorphisme formel

Cette section présente les différents formalismes utilisés pour décrire les codes malveillants évolutifs. Partant de la définition historique d'un virus, nous exposons les modèles successifs qui ont permis d'aboutir aux définitions formelles du métamorphisme.

1.1.1 Définition historique d'un virus

Les travaux de Kraus [102, 103] sont les premiers à définir la notion de code auto-reproducteur évolutif au début des années 80. Toutefois, l'étude de l'auto-reproduction y est menée indépendamment du caractère malveillant dont va faire l'objet cette technique à travers l'apparition des premiers virus informatiques. C'est pourquoi Kraus n'oriente pas ces travaux en termes de détection. Il faut attendre la thèse de Cohen pour obtenir la première étude à la fois théorique et pratique de la notion de virus informatique évolutif [43]. Dans ces travaux, Cohen s'attachent à déterminer la complexité de la détection virale en s'appuyant sur le formalisme des machines de Turing [137]. Ces machines sont définies par la donnée de trois éléments :

- un **ruban** (de calcul) composé d'une infinité de cellules. Chaque cellule contient un symbole parmi un alphabet fini, dit alphabet de bande. Parmi ces symboles, le symbole vide joue un rôle particulier. Il s'agit du symbole contenu dans chaque cellule du ruban de calcul lorsqu'aucun programme n'est fourni à la machine de Turing ;
- une **tête de lecture** se déplaçant sur le ruban et en charge de l'acquisition (lecture) et de la restitution (écriture) des symboles. Cette tête de lecture se voit autoriser deux mouvements : avancer ou reculer d'une cellule ;
- un **automate déterministe à états fini (« *Determinist Finite State Machine* » ou DFA)** qui comprend un ensemble fini d'états internes (dits états de la machine de Turing) et qui à tout symbole lu par la tête de lecture et à tout état interne associe un nouvel état, un nouveau symbole (écrit par la tête de lecture) et un mouvement de la tête de lecture.

1.1.1.1 Le modèle viral de Cohen

Afin d'introduire la définition virale proposée par Cohen [43], nous adoptons par la suite son formalisme décrivant une machine de Turing M comme un quintuplet $(S_M, I_M, \$_M, \square_M, P_M)$ avec :

- S_M, un ensemble fini d'états possibles de la machine M ;
- I_M, un ensemble fini de symboles correspondant à l'alphabet de bande de M ;
- $\$_M : \mathbb{N} \to S_M$, une fonction temporelle d'état qui à tout instant associe l'état interne de M ;
- $\square_M : \mathbb{N} \times \mathbb{N} \to I_M$, une fonction temporelle de bande qui à tout instant et à tout index de cellule associe le symbole contenu dans cette cellule ;
- $P_M : \mathbb{N} \to \mathbb{N}$, une fonction temporelle de cellule qui à tout instant associe l'index de la cellule devant laquelle se trouve la tête de lecture ;

L'originalité de l'approche de Cohen réside dans la définition d'un ensemble viral $\mathcal{V}_c$ comme couple constitué d'un environnement d'exécution (une machine de Turing M) et d'un ensemble de programmes viraux (V) s'exécutant dans cet environnent d'exécution. Un tel programme présente alors un caractère viral uniquement pour l'architecture cible et demeure inerte pour toute autre architecture. À titre d'exemple, il suffit de considérer un virus compilé ou assemblé

pour une architecture de type x86. Ce programme ne peut alors s'exécuter directement sur une autre architecture (de type ARM par exemple).

Nous introduisons les notations requisent pour la définition d'un ensemble viral selon Cohen. Nous notons $\mathcal{M}$ l'ensemble des machines de Turing. Dans toute cette section, la notation I_M^* désigne l'ensemble des mots sur l'alphabet I_M, c'est-à-dire l'ensemble des programmes possibles pour une machine M. Pour tout v de I_M^*, la notation $|v|$ désigne la taille du programme v.

Définition 1. *(Virus selon Cohen [43]). Un ensemble viral $\mathcal{V}_c$ est défini de la façon suivante :*

$$\begin{aligned}\forall M\forall V(M,V)\in\mathcal{V}_c \Leftrightarrow &[V\subset I_M^*]\ et\ [M\in\mathcal{M}]\ et\ [\forall v\in V[\forall t\forall j\\ &[1.P_M(t)=j\ et\\ &\ 2.\$_M(t)=\$(0)\ et\\ &\ 3.(\square_M(t,j),\ldots,\square_M(t,j+|v|-1))=v]\\ \Rightarrow&[\exists v'\in V[\exists t'>t[\exists j'\\ &[4.[[(j'+|v'|\leq j]\ ou\ [(j+|v|)\leq j']]et\\ &\ 5.(\square_M(t',j'),\ldots,\square_M(t',j'+|v'|-1))=v'\ et\\ &\ 6.[\exists t'',[t<t''<t']\ et\ [P_M(t'')\in j',\ldots,j'+|v'|-1]]\\ &]]]]]\end{aligned}$$

Cette définition traduit le fait que le couple (M,V) formé d'une machine de Turing M et d'un ensemble de programme V pour la machine M est un ensemble viral noté $\mathcal{V}_c$ si et seulement si tout programme v de V est tel que si à un instant donné t :

- 1. la tête de lecture pointe sur la cellule d'index j ;
- 2. la machine est dans son état initial noté $\$(0)$;
- 3. les cellules à partir de j contiennent la séquence de symboles constituant le code du programme v ;

alors il existe un autre instant t' postérieur à t pour lequel un autre programme v' de V vérifie :

- 4 et 5. v' est écrit soit avant soit après v ;
- 6. la tête de lecture finit par être positionnée au début du programme v'.

Afin de faciliter la lecture, nous adopterons la notation abrégée, proposée par Cohen, de la définition 1 : $\forall M\forall V,(M,V)\in\mathcal{V}_c$ si et seulement si $V\subset I_M^*$ et $\forall v\in V,[v\overset{M}{\Rightarrow}V]$

1.1.1.2 Résultats de Cohen : indécidabilité de la détection et de l'évolution virale

Parmi les résultats obtenus dans ces travaux [43], les deux principaux problèmes abordés sont celui l'indécidabilité de la détection virale et celui de l'évolution virale. Le premier problème consiste à savoir s'il est possible de définir un algorithme (une machine de Turing) permettant de détecter un quelconque ensemble viral (au sens de Cohen). Le second problème consiste à savoir s'il est

possible de définir un algorithme permettant de déterminer si un programme correspond à une forme mutée d'un autre programme. Les résultats obtenus correspondent aux deux théorèmes suivants :

Théorème 1. *(Indécidabilité de la détection virale [43]). Il n'existe pas de machine de Turing D comprenant un état particulier s tel que pour toute machine de Turing M et pour tout ensemble de programmes V de M, le calcul de D s'arrête à un instant t pour lequel, D est dans l'état s si et seulement si le couple (M, V) est un ensemble viral.*

En d'autres termes, il n'est pas possible de définir un algorithme générique permettant de répondre à la première question. La détection virale se doit donc d'être approximative.

Théorème 2. *(Indécidabilité de l'évolution virale [43]). Il n'existe pas de machine de Turing D avec un état particulier noté s tel que pour tout ensemble viral (M, V) et pour tous programmes v et v' de V, le calcul de D s'arrête à un instant t pour lequel, D est dans l'état s si et seulement si $v \overset{M}{\Rightarrow} \{v'\}$.*

Comme pour le problème de la détection virale, il n'est pas possible de définir un algorithme générique permettant de répondre à la deuxième question. Seule une détection approximative des codes évolutifs est possible.

Les travaux de Cohen constituent une première approche formelle de la notion de virus. Le modèle générique adopté apporte les deux premiers résultats négatifs concernant la détection des codes malveillants. Le deuxième résultat implique d'ores et déjà que la détection des codes auto-reproducteurs métamorphes est impossible.

1.1.2 Fonctions récursives et codes maveillants

Peu de temps après les travaux de Cohen, Adleman propose un modèle plus abstrait des codes malveillants s'appuyant sur un autre formalisme de la théorie de la calculabilité [1]. L'utilisation des fonctions partielles récursives lui permet entre autres de décrire différents types de codes malveillants. Outre le fait que ce nouveau formalisme autorise des résultats plus précis, il permet aussi de se familiariser avec les notations utilisées par la suite. La classification virale d'Adleman permet d'introduire les travaux de Zuo *et al.* [195, 196], présentés en section 1.1.2.3, qui aboutissent à une première définition formelle du métamorphisme dans un cadre viral.

Les fonctions partielles récursives introduites par Kleene [96] correspondent à un formalisme plus abstrait que celui des machines de Turing. Pour autant, ce formalisme ne perd pas en généralité puisque la classe des fonctions partielles récursives correspondent exactement aux fonctions calculées par les machines de Turing [148]. Afin de présenter les travaux d'Adleman, nous introduisons au préalable les notations nécessaires.

Classiquement, l'ensemble des entiers naturel est désigné par $\mathbb{N}$. L'ensemble des séquences finies d'entiers naturels est noté S. Pour tout $(s_1, s_2, \ldots, s_n) \in$

S^n, la notation $\langle s_1, s_2, \ldots, s_n \rangle$ représente une fonction injective calculable et à valeurs dans $\mathbb{N}$ dont l'inverse est également calculable. Il peut, par exemple, s'agir d'une numérotation de Gödel [1] [72]. Afin de faciliter la lisibilité, pour toute fonction partielle $f : \mathbb{N} \mapsto \mathbb{N}$, nous notons $f(s_1, s_2, \ldots, s_n)$ au lieu de $f(\langle s_1, s_2, \ldots, s_n \rangle)$. Pour toute séquence $p = (i_1, i_2, \ldots, i_n) \in S$, le remplacement du k^{e} élément par une fonction v calculable sur $\mathbb{N}$ sera noté $p[v(\underline{i_k})]$ (c'est-à-dire $p[v(\underline{i_k})] = (i_1, i_2, \ldots, v(i_k), \ldots, i_n)$).

La notion d'environnement d'exécution est représentée sous la forme d'un couple constitué d'un ensemble de données d et d'un ensemble de programmes p. Pour toute numérotation de Gödel des fonctions partielles récursives $\{\phi_i\}$, $\phi_P(d, p)$ désignera la fonction partielle calculée par le programme P (en numérotation de Gödel des programmes) sur le couple (d, p).

1.1.2.1 Le modèle viral d'Adleman

La définition virale proposée par Adleman est plus restrictive que celle de Cohen. Elle impose au comportement viral une action parmi les trois que sont la *nuisance*, l'*imitation* et l'*infection*.

Définition 2. *(Virus selon Adleman [1]). Une fonction récursive totale v est considérée comme virale par rapport à $\{\phi_i\}$, si pour tout environnement (d, p), elle présente au moins l'un des 3 comportements suivants :*

- ***la nuisance**, qui correspond à l'exécution de la charge virale à proprement parlé indépendamment de la fonctionnalité initiale du programme infecté, $\forall (i, j) \in \mathbb{N}^2, \phi_{v(i)}(d, p) = \phi_{v(j)}(d, p)$. Ce comportement traduit le caractère malveillant ;*
- ***l'imitation**, quand le programme infecté se comporte comme avant l'infection, $\forall i \in \mathbb{N}, \phi_{v(i)}(d, p) = \phi_i(d, p)$;*
- ***l'infection**, qui permet à un virus d'assurer sa propagation au sain du système en modifiant (infectant) d'autres programmes, $\forall i \in \mathbb{N}, \phi_{v(i)}(d, p) = \langle d', \epsilon_v(p'_1), \ldots, \epsilon_v(p'_n) \rangle$ avec $\phi_i(d, p) = \langle d', p' \rangle$, $p' = \langle p'_1, \ldots, p'_n \rangle$ et ϵ_v est une fonction de sélection calculable définie par*

$$\epsilon_v(i) = \begin{cases} i \text{ (le programme } i \text{ est conservé tel quel)} \\ ou \\ v(i) \text{ (le programme } i \text{ est infecté par } v). \end{cases}$$

Fort de cette définition, Adleman propose alors deux caractéristiques sur les programmes viraux :

Définition 3. *(Virus pathogènes et contagieux [1]). Pour toute numérotation de Gödel des fonctions partielles récursives $\{\phi_i\}$, pour tout virus v par rapport à $\{\phi_i\}$, pour tout $i \in \mathbb{N}$, $v(i)$ est dit :*

1. La numérotation de Gödel s'appuie sur la factorisation en nombres premiers. Un entier est assigné à chaque symbole du langage. À toute séquence d'entiers $x_1 x_2 x_3 \ldots x_n$ est alors associé l'entier égal au produit des n premiers nombres premiers élevés à la puissance de l'entier correspondant dans la séquence, soit $2^{x_1} \times 3^{x_2} \times 5^{x_3} \times \cdots \times p_n^{x_n}$ où p_n désigne le n^{e} nombre premier.

- ***pathogène*** *s'il existe* $(d, p) \in S$ *pour lequel* $v(i)$ *n'infecte pas et n'imite pas ;*
- ***contagieux*** *s'il existe* $(d, p) \in S$ *pour lequel* $v(i)$ *infecte.*

Dans la définition précédente d'Adleman, on remarquera que le programme $v(i)$ est *pathogène* s'il existe un environnement (d, p) pour lequel il est uniquement *nuisible*. En effet, tout virus au sens d'Adleman comprend l'un des trois comportements déjà présentés en définition 2 : la *nuisance*, l'*imitation* et l'*infection*. Maintenant, dire que $v(i)$ est *pathogène* équivaut, par définition, à dire qu'il existe (d, p) pour lequel $v(i)$ n'*infecte* pas et n'*imite* pas. Dans ce cas, $v(i)$ est alors nécessairement *nuisible*.

À partir des caractères *pathogènes* ou *contagieux* d'un programme, Adleman répartit l'ensemble des virus sous la forme d'une union disjointe en quatre catégories :

Définition 4. *(Classification virale d'Adleman [1]). Pour toute numérotation de Gödel des fonctions partielles récursives* $\{\phi_i\}$*, pour tout virus* v *par rapport à* $\{\phi_i\}$*, pour tout* $i \in \mathbb{N}$*,* $v(i)$ *est :*

- ***bénin*** *s'il n'est ni pathogène, ni contagieux ;*
- ***épéin*** *(un cheval de Troie) s'il est pathogène mais non contagieux ;*
- ***disséminateur*** *(«* dropper *») s'il n'est pas pathogène mais contagieux ;*
- ***virulent*** *s'il est a la fois pathogène et contagieux.*

En d'autres termes, un virus *bénin imite* tout programme sur lequel il est appliqué. Un *cheval de Troie* est un virus qui n'infecte pas d'autres programmes mais apporte une fonctionnalité supplémentaire (malveillante) pour un environnement particulier. Un virus est un *disséminateur* s'il infecte uniquement pour un environnement donné. Dans le cas général, les virus sont qualifiés de *virulents*.

1.1.2.2 Résultats d'Adleman : complexité de la détection et de l'évolution virale

Cette définition permet de compléter et d'affiner les résultats obtenus par Cohen en considérant d'autres types de codes malveillants. En effet, les deux théorèmes suivants montrent que, selon ce nouveau modèle, la détection de l'ensemble des virus au sens d'Adleman est décidable.

Théorème 3. *(Complexité de la détection virale selon Adleman [1]). L'ensemble* $V = \{v | \phi_v$ *est un virus (au sens d'Adleman)*$\}$ *est* $\prod_2$*-complet*[2]*.*

Bien que décidable, la détection d'un virus au sens d'Adleman est hors de portée pratique. Intéressons-nous maintenant au cas des programmes évolutifs.

2. Soit $i \geq 1$, un langage L est dans $\prod_i^P$ s'il existe un polynôme q et un problème A dans P tel que $w \in L$ si et seulement si $\forall u_1 \in \{0,1\}^{q(|x|)} \exists u_2 \in \{0,1\}^{q(|x|)} \dots Q_i u_i \in \{0,1\}^{q(|x|)} < w, u_1, u_2, \dots, u_i > \in A$, avec Q_i désignant $\exists$ ou $\forall$ selon si i est respectivement pair ou impair. On remarquera que $\prod_1^P$ =coNP.

Théorème 4. *(Complexité de l'évolution virale selon Adleman [1]). L'ensemble des infection de v défini par $\{i|\exists j, i = v(j)\}$ est $\sum_1$-complet*[3].

Ce théorème exprime le fait que déterminer si, pour un virus v et un programme i donnés, il existe un programme j tel que i soit une version infectée de j par le virus v, est un problème NP-complet.

1.1.2.3 Le modèle viral de Zuo *et al.*

Zuo *et al.* [195, 196] ont repris et complété le formalisme d'Aldeman afin prendre en compte d'autres types de virus parmi lesquels, les virus résidents, compagnons, furtifs, polymorphes, métamorphes, etc. Nous présentons ici uniquement les définitions utiles à la description des codes malveillants évolutifs, c'est-à-dire modifiant leur forme à chaque réplication. Le concept central de leur modélisation est la notion de *noyau* (« *kernel* ») qui caractérise entièrement un virus. Un tel noyau se présente sous la forme d'un quadruplet (T, I, D, S) composé des prédicats récursifs T et I, ainsi que des fonctions récursives D et S. Les prédicats T (« *Trigger* ») et I (« *Infect* ») représentent respectivement une condition de *déclenchement* et une condition d'*infection*. Les fonction récursives D (« *Dammage* ») et S (« *Select* ») désignent respectivement une fonction de *nuisance* et une fonction de *sélection*. De plus, il est supposé que les prédicats T et I vérifient les deux conditions suivantes : l'ensemble des couples (d, p) pour lesquels T et I sont simultanément faux est infini, et l'ensemble des couples (d, p) pour lesquels ces prédicats sont simultanément vrais est vide.

1.1.2.3.a Virus non-résident

La première définition proposée, qui correspond à celle d'un virus non résident, permet de se familiariser avec les notations employées.

Définition 5. *(Virus non-résident [195]). La fonction récursive totale v de noyau (T, I, D, S) est un virus non-résident si pour tout programme x et tout environnement (d, p),*

$$\phi_{v(x)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_x(d, p[v(\underline{S(p)})]), & si\ I(d,p) \\ \phi_x(d,p), & sinon \end{cases} \tag{1.1}$$

Le premier cas correspond au *déclenchement* de la charge finale ; si l'environnement d'exécution (d, p) satisfait le prédicat T alors le virus v exécute la fonction de *nuisance* D sur son environnement. Le dernier cas correspond à celui où les deux prédicats T et I sont simultanément faux, c'est-à-dire qu'il n'y a ni *déclenchement* de la charge D, ni *infection*. Dans ce cas, v imite le programme *infecté* x (l'exécution de v est similaire à celle de x). Le deuxième

3. Soit $i \geq 1$, un langage L est dans $\sum_i^P$ s'il existe un polynôme q et un problème A dans P tel que $w \in L$ si et seulement si $\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \ldots Q_i u_i \in \{0,1\}^{q(|x|)} < w, u_1, u_2, \ldots, u_i > \in A$, avec Q_i désignant $\forall$ ou $\exists$ selon si i est respectivement pair ou impair. On remarquera que $\sum_1^P$ =NP.

cas définit un virus non-résident, qui *sélectionne* un programme $S(p)$, puis le substitue par sa version *infectée* et enfin, lance l'exécution du programme x sur ce nouvel environnement. On remarquera qu'il existe deux moments propices à l'*infection* : avant ou après l'exécution du programme infecté x. Le choix fait dans la définition précédente correspond bien à une *infection* avant exécution $\phi_x(d, p[v(\underline{S(p)})])$. Le cas contraire s'écrirait sous la forme $\phi_x(d, p)[v(\underline{S(p)})]$.

1.1.2.3.b Virus polymorphes

À partir de cette définition initiale d'un virus non-résident et de la notion de *noyau* viral, Zuo *et al.* proposent d'autres types de virus. Ils précisent ainsi la notion de mutation de code à travers une première version du polymorphisme, le polymorphisme à deux formes.

Définition 6. *(Virus polymorphe à deux formes [195]). La paire (v, v') constituée de deux fonctions récursives totales v et v' de même noyau (T, I, D, S) est un virus polymorphe à deux formes si pour tout x et tout environnement (d, p),*

$$\phi_{v(x)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_x(d, p[v'(\underline{S(p)})]), & si\ I(d,p) \\ \phi_x(d,p), & sinon \end{cases} \tag{1.2}$$

et

$$\phi_{v'(x)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_x(d, p[v(\underline{S(p)})]), & si\ I(d,p) \\ \phi_x(d,p), & sinon \end{cases} \tag{1.3}$$

Pour chaque forme, lors de l'*infection* (satisfaction du prédicat I) l'autre forme est utilisée pour *infecter* le programme sélectionné. Ce résultat peut s'étendre à un nombre fini de formes. Zuo *et al.* démontrent aussi l'existence théorique de virus polymorphes à une infinité de formes (voir [195]). La seule différence avec le cas d'un virus non-résident tient dans l'introduction d'un index de forme n.

Définition 7. *(Virus polymorphe à infinité de formes [195]). La fonction récursive totale $v(n, x)$ de noyau (T, I, D, S) est un virus polymorphe à infinité de formes si pour tout (n, x) et tout environnement (d, p),*

$$\phi_{v(n,x)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_x(d, p[v(n+1, \underline{S(p)})]), & si\ I(d,p) \\ \phi_x(d,p), & sinon \end{cases} \tag{1.4}$$

Cette définition complète celle d'Adleman afin de prendre en compte le cas des mutations. On remarquera qu'un tel virus polymorphe conserve le même *noyau* quelle que soit la génération considérée indexée par n. Dans ce cas, la complexité de détection est donnée par le théorème suivant.

Théorème 5. *(Complexité des virus à noyaux constants [195]). L'ensemble des virus présentant le même noyau, tel que défini par de Zuo* et al.*, est $\prod_2$-complet.*

Les résultats obtenus ici n'imposent pas de limitation dans la taille des programmes. Dans le cas où les programmes sont de tailles finies, Spinellis a démontré qu'il est possible de construire des codes viraux polymorphes pour lesquels la détection est prouvée NP-complète [157]. Ce résultat est obtenu par réduction du problème SAT [92] à celui de la détection d'un virus polymorphe de taille finie.

Les virus polymorphes présentant un *noyau* constant, l'étape suivante dans la modélisation des codes évolutifs s'est naturellement orientée vers la formalisation des virus possédant des *noyaux* différents. C'est ainsi qu'est née la première définition formelle du métamorphisme.

1.1.2.3.c Virus métamorphes

Comme dans le cas du polymorphisme, Zuo *et al.* proposent une définition d'un virus métamorphe à deux formes. Cette définition reprend celle des virus polymorphes à deux formes mais avec cette fois-ci deux *noyaux* distincts.

Définition 8. *(Virus métamorphe à deux formes [196]). La paire (v, v') constituée de deux fonctions récursives totales v et v' de noyaux respectifs (T, I, D, S) et (T', I', D', S') est un virus métamorphe à deux formes si pour tout x et tout environnement (d, p),*

$$\phi_{v(x)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_x(d, p[v'(\underline{S(p)})]), & si\ I(d,p) \\ \phi_x(d,p), & sinon \end{cases} \tag{1.5}$$

et

$$\phi_{v'(x)}(d,p) = \begin{cases} D'(d,p), & si\ T'(d,p) \\ \phi_x(d, p[v(\underline{S'(p)})]), & si\ I'(d,p) \\ \phi_x(d,p), & sinon \end{cases} \tag{1.6}$$

Un tel virus métamorphe est composé de deux formes v et v' ainsi que de deux *noyaux* distincts (T, I, D, S) et (T', I', D', S'). Il est possible d'étendre la définition d'un virus métamorphe à deux formes au cas d'un nombre fini de formes. Il suffit pour cela de considérer autant de noyaux qu'il existe de formes virales distinctes. Étonnamment Zuo *et al.* ne se sont pas intéressés à l'existence éventuelle de virus métamorphes à infinité de formes. Nous proposons une définition de tels virus dont nous démontrons l'existence en annexe A.

La difficulté de la détection des virus à noyaux variables n'est par contre pas définie dans les travaux de Zuo *et al.* Dans la section suivante, nous exposons un autre formalisme permettant d'affiner le modèle de mutation employé par les codes malveillants évolutifs et donc, la difficulté de leur détection.

1.1.3 Grammaires formelles et métamorphisme

La première modélisation des mutations de code au moyen de grammaires formelles est due à Qozah [141]. Ce formalisme applicable au polymorphisme

ainsi qu'au métamorphisme autorise une hiérarchisation plus précise du problème de la détection des codes évolutifs en fonction du type de grammaires qui régissent leurs mutations.

1.1.3.1 Mutation de code et grammaires formelles

Une grammaire permet de formaliser la syntaxe d'un langage, c'est-à-dire l'ensemble des mots admissibles sur un alphabet donné. Dans notre contexte, chaque mot peut s'interpréter comme une forme mutée d'un même code malveillant évolutif.

Définition 9. *(Grammaire formelle [137]). Une grammaire formelle est un quadruplet* (N, T, S, R) *avec :*

- N *un ensemble de symboles non-terminaux ;*
- T *un ensemble de symboles terminaux vérifiant* $N \cap T = \emptyset$ *;*
- $S \in N$ *le symbole de départ ;*
- R *un ensemble de règles de réécritures de la forme* $R \subseteq (T \cup N)^* \times (T \cup N)^*$ *tel que* $(u, v) \in R \Rightarrow u \in T^*$.

La figure 1.1 présente un exemple de grammaire formelle G composée de deux symboles non terminaux (A et B), de trois symboles terminaux (a, b et c), de son symbole de départ S et d'un ensemble de règles de réécriture R.

$$G = (\{A, B\}, \{a, b, c\}, S, R), \text{ avec } R = \begin{cases} S ::= aA|bA|c, \\ A ::= aA|B, \\ B ::= bB|c. \end{cases}$$

FIGURE 1.1 – Exemple de grammaire formelle.

La notion de grammaire formelle permet de définir le langage formel engendré par à cette grammaire comme l'ensemble des mots qui peuvent être obtenus, à partir du symbole initial S, au moyen de l'ensemble des règles de réécriture R. Pour une grammaire donnée G, nous noterons $L(G)$ le langage formel engendré par G. Par exemple, la grammaire présentée en figure 1.1 peut générer le mot *aabbc* grâce à la dérivation suivante : $S \overset{R}{\Rightarrow} aA \overset{R}{\Rightarrow} aaA \overset{R}{\Rightarrow} aaB \overset{R}{\Rightarrow} aabB \overset{R}{\Rightarrow} aabbB \overset{R}{\Rightarrow} aabbc$. Le langage défini par cette grammaire est $L(G) = \{(a|b)(a)^*(b)^*c, c\}$.

Dans le cadre de mutation de code, une grammaire formelle G décrit la syntaxe de ces mutations qui ne sont autres que les mots du langage $L(G)$. Détecter une mutation se ramène alors à un problème classique de la théorie langage, celui de la décision d'un langage.

Définition 10. *(Décision d'un langage [90]). Soit* $G = (N, T, S, R)$ *une grammaire et* $x \in T^*$ *une chaîne, le problème de décision du langage* $L(G)$ *consiste à déterminer si* $x \in L(G)$.

Pour illustrer cette définition, reprenons l'exemple de la figure 1.1. Le mot *aaabbbc* appartient bien au langage de G car il peut être obtenu par la dérivation suivante : $S \overset{R}{\Rightarrow} aA \overset{R}{\Rightarrow} aaA \overset{R}{\Rightarrow} aaaA \overset{R}{\Rightarrow} aaaB \overset{R}{\Rightarrow} aaabB \overset{R}{\Rightarrow} aaabbB \overset{R}{\Rightarrow} aaabbbB \overset{R}{\Rightarrow} aaabbbc$. Par contre le mot *aabbca* n'appartient pas à ce langage car tout mot de $L(G)$ se termine nécessairement par un c.

Considérons maintenant une routine de (dé)chiffrement d'un code malveillant polymorphe au sens de la définition informelle donnée en introduction telle que présentée en figure 1.2.

$G = (\{A, B\}, \{a, b, c, x, y\}, S, R)$ avec,

$$T = \begin{cases} a\texttt{="nop",} \\ b\texttt{="sub edx,0",} \\ c\texttt{="push ebx"+"pop ebx",} \\ x\texttt{="xor [edi],al",} \\ y\texttt{="inc al",} \\ e\texttt{=""}. \end{cases} \quad \text{et } R = \begin{cases} S ::= aS|bS|cS|xA, \\ A ::= aA|bA|cA|yB, \\ B ::= aB|bB|cB|e. \end{cases}$$

FIGURE 1.2 – Exemple de grammaire permettant de générer la routine de déchiffrement d'un code malveillant polymorphe inspiré de [141].

Une dérivation possible est $S \overset{R}{\Rightarrow} aS \overset{R}{\Rightarrow} acS \overset{R}{\Rightarrow} acxA \overset{R}{\Rightarrow} acxcA \overset{R}{\Rightarrow} acxcyB \overset{R}{\Rightarrow} acxcybB \overset{R}{\Rightarrow} acxcybe$. Elle correspond au code du programme 1 dans le tableau 1.1. Le code du programme 2 correspond quant à lui à la dérivation suivante : $S \overset{R}{\Rightarrow} cS \overset{R}{\Rightarrow} cbS \overset{R}{\Rightarrow} abxA \overset{R}{\Rightarrow} cbxaA \overset{R}{\Rightarrow} cbxayB \overset{R}{\Rightarrow} cbxayaB \overset{R}{\Rightarrow} cbxayae$. Dans ce tableau, seul le code en gras correspond à du code « utile », c'est-à-dire effectuant le (dé)chiffrement du corps du programme, le reste représente du code « inutile » (appelé aussi « code mort » , en anglais « *garbage* » ou encore « *junk code* »). Cet extrait de code correspond au calcul d'un `OU` exclusif (`XOR`) entre l'octet pointé par le registre `edi` et l'octet contenu dans le registre `al`, registre lui même incrémenté.

Pogramme 1	**Pogramme 2**
`nop`	`push ebx`
`push ebx`	`pop ebx`
`pop ebx`	`sub edx, 0`
`xor [edi], al`	**`xor [edi], al`**
`inc al`	`nop`
`sub edx, 0`	**`inc al`**
	`nop`

TABLE 1.1 – Exemple de routines polymorphes obtenues par la grammaire G de la figure 1.2.

Quelle que soit la dérivation considérée, les symboles x et y correspondant respectivement aux instructions `xor [edi],al` et `inc al` apparaissent né-

cessairement dans le mot final. Une telle routine polymorphe est facilement détectable au moyen d'un automate déterministe donné en figure 1.3.

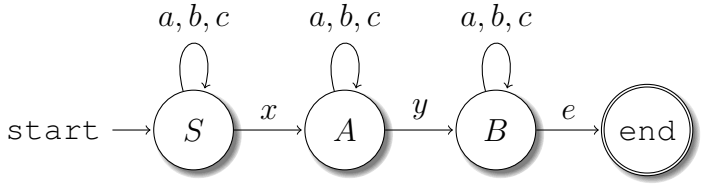


FIGURE 1.3 – Automate permettant de détecter la routine polymorphe générée par la grammaire G de la figure 1.2

Si pour ce type de grammaire le problème de la décision du langage apparait trivial, qu'en est-il dans le cas général ?

Afin de répondre à cette question, nous nous référons aux travaux de Chomsky [29, 30] qui répartissent les grammaires formelles en quatre catégories distinctes. Nous illustrons cette classification au moyen d'une grammaire $G = (N, T, S, R)$ avec $N = \{A, B, C\}$ et $T = \{a, b, c\}$ dont seules les règles de réécritures varient en fonction du type de grammaire considérée :

- les **grammaires de type 0**, dites ***libres***, avec des règles de production de la forme $x ::= y$, $y \in (N \cup T)^*$. Ces grammaires ne sont soumises à aucune contrainte particulière. Un exemple de règles de réécriture correspondant à une grammaire libre est fourni ci-après :

$$R = \begin{cases} S ::= aBBA, \\ AB ::= BA|Sc|C, \\ cBc ::= aaa|ab|c, \\ A ::= CBC, \\ B ::= cBca|cb|aB, \\ C ::= a|c. \end{cases}$$

  On remarquera que ce type de grammaire autorise entre autre la diminution de la taille des mots qu'elle engendre ;
- les **grammaires de type 1**, dites ***contextuelles***, pour lesquelles la seule contrainte porte sur la taille des mots qui ne peut pas diminuer. Cette classe représente tous les langages naturels. Un exemple de règles de réécriture correspondant à une grammaire contextuelle est fourni ci-après :

$$R = \begin{cases} S ::= aBBA, \\ AB ::= BA|Sc, \\ cBc ::= aaa, \\ A ::= CBC, \\ B ::= cBca|cb|aB, \\ C ::= a|c. \end{cases}$$

- les **grammaires de type 2**, dites ***non-contextuelles***, comprennent toutes les grammaires dont les règles de production sont de la forme $X ::= y$ où

X désigne un unique symbole non-terminal et y désigne un élément de $(N \cup T)^*$. Ces grammaires sont notamment utilisées pour décrire la syntaxe des langages de programmation. Un exemple de règles de réécriture correspondant à une grammaire non-contextuelle est fourni ci-après :

$$R = \begin{cases} S ::= aBBA, \\ A ::= CBC, \\ B ::= cBca|cb|aB, \\ C ::= a|c. \end{cases}$$

– les **grammaires de type 3**, dites ***régulières***, possèdent des règles de réécritures de la forme $X ::= x$ ou $X ::= xY$ avec $(X, Y) \in N^2$ et $x \in T^*$. Un exemple de règles de réécriture correspondant à un grammaire régulière est fourni ci-après :

$$R = \begin{cases} S ::= aS|aA, \\ A ::= Bc, \\ B ::= Bc|c. \end{cases}$$

Reconsidérons alors le problème de la décision d'un langage. Ce problème présente différents niveaux de difficulté en fonction du type de grammaire envisagé.

Théorème 6. *(Difficulté du problème de décision d'un langage [30]). Le problème de décision d'un langage est :*
- *indécidable pour les grammaires de type 0 ;*
- *NP-complet pour les grammaires de type 1 et 2 ;*
- *polynomial pour les grammaires de type 3.*

1.1.3.2 Le métamorphisme selon Filiol

S'appuyant sur les résultats de Chomsky, Filiol [64] propose de définir le métamorphisme au moyen de grammaires formelles. Si dans la définition du Zuo *et al.*, l'évolution d'un noyau viral métamorphe n'est pas explicité, ce nouveau modèle permet de caractériser comment les règles de mutations évoluent à chaque réplication.

Définition 11. *(Virus métamorphe selon Filiol [64]). Soient $G_1 = (N, T, S, R)$ et $G_2 = (N', T', S', R')$ deux grammaires. T' désigne un ensemble de grammaires formelles. S' est la grammaire de départ notée G_1 et R' est un ensemble de règles de réécriture respectivement à $(N' \cup T')^*$. Un virus métamorphe est alors décrit par G_2 et chacune de ses formes (mutations) est un mot de $L(L(G_2))$.*

En reprenant les notations de la définition 11, G_2 est une méta-grammaire, c'est-à-dire une grammaire produisant d'autres grammaires. Plus précisément, chaque mot G de $L(G_2)$ est une grammaire formelle, pour laquelle chaque mot v du langage engendré $L(G)$ est une mutation du virus représenté par la grammaire G_2. Détecter un tel virus revient à définir si un programme quelconque correspond à une des formes engendrées par $L(G_2)$. Autrement dit, le problème de la détection se ramène dans ce cas à celui de la décision du langage $L(L(G_2))$.

On remarquera que la définition du métamorphisme viral selon Filiol décrit uniquement les mécanismes régissant les mutations de code contrairement au modèle de Zuo *et al.* qui considère l'*infection* et l'*imitation* conformément au modèle viral d'Adleman. La définition 11 du métamorphisme apparaît comme la plus générale puisqu'elle autorise d'autres types de codes auto-reproducteurs tels que les vers, qui n'*infectent* pas et n'*imitent* pas d'autres programmes.

Afin d'illustrer ces résultats, Filiol présente une preuve de concept (« *Proof Of Concept* » ou POC) abstraite de virus métamorphe, dénommé `POC_PBMOT`. Cet exemple s'appuie sur le problème du mot pour proposer un modèle de virus dont la détection est indécidable. Le problème du mot, formalisé par Post, est connu pour être indécidable [138], au même titre que le problème de l'arrêt de Turing [161]. Ce problème consiste à déterminer si deux mots r et s, de tailles finies sur un alphabet donné, sont équivalents pour un système de réécriture R, c'est-à-dire si le mot s peut être obtenu à partir du mot r au moyen des règles R et réciproquement.

Actuellement, les virus métamorphes connus ne semblent pas conçus à partir de grammaires formelles complexes (autres que de type 3) mais plutôt au moyen de techniques empiriques de mutation de codes [158, chapitre 7]. Cet aspect est conforté par l'exemple du code malveillant Win32.MetaPHOR [4] connu pour être le virus métamorphe le plus abouti [64, 158] dont les règles réécritures sont fournies en annexe B. Ces mutations de code correspondent à des transformations syntaxiques généralement simples comme nous le verrons en section 1.2. Les travaux Zbitskiy [189] confirment ce constat, en modélisant les transformations de code utilisées actuellement par des programmes polymorphes et métamorphes.

1.1.4 Bilan des formalismes et discussion sur le métamorphisme

À partir de la définition la plus générale des codes auto-reproducteurs évolutifs fournie par Cohen, nous avons présenté les différents formalismes conduisant aux définitions du métamorphisme. À chaque modèle proposé, nous avons associé la difficulté de la détection virale. Ainsi, selon le modèle de Cohen qui représente les codes auto-reproducteurs évolutifs au moyen de machines de Turing, la détection des ces programmes est un problème indécidable. Selon le modèle d'Adleman, s'appuyant sur les fonctions partielles récursives pour affiner la définition de Cohen, déterminer l'ensemble des infections d'un virus v est NP-complet. En ce qui concerne le métamorphisme, Zuo *et al.* en proposent une première définition formelle pour laquelle ils ne démontrent pas la complexité de détection. Ils montrent cependant que, selon leur modèle issu de celui d'Adleman, la détection des codes viraux à noyau fixe, c'est-à-dire polymorphes, est Π_2-complète. Finalement, la définition de Filiol formalise les mutations de code au moyen de grammaires formelles. La détection des programmes évolutifs dépend alors du type de grammaire formelle considérée. La détection d'un

4. Ce virus est présenté en section 1.2.7.2

code polymorphe est alors prouvée : indécidable pour les grammaires libres, NP-complète pour les grammaires contextuelles et non-contextuelles, et enfin polynomiale pour les grammaires régulières. Pour cette dernière définition, la complexité des codes métamorphes reste toutefois à définir.

Au terme de cette section exposant les différents modèles du métamorphisme, nous constatons qu'il existe à ce jour deux façon de l'envisager :

- la première correspond à une définition empirique, telle que celle donnée en introduction, issue d'observations de codes malveillants métamorphes réels. Dans ce cas, le métamorphisme est considéré comme du polymorphisme de corps [158], dans lequel l'intégralité du nouveau programme produit évolue sans modification dynamique de son propre code (déchiffrement/chiffrement). Cette définition est partagée par de nombreux travaux parmi lesquels [31, 54, 106, 139, 159, 173, 174, 179, 180, 184, 190]. D'un point de vue formel, les règles de mutation correspondent alors à une grammaire G pour laquelle une variante métamorphe v est un mot de $L(G)$;
- la deuxième manière d'envisager le métamorphisme correspond à une vision plus théorique telle que proposée par Zuo *et al.* [196] ainsi que Filiol [64]. Dans ce cas, ce sont les règles de mutation elles-mêmes qui mutent lors de chaque réplication du code métamorphe. Ces règles sont représentées par une grammaire G pour laquelle chaque variante v est alors un mot de $L(L(G))$.

On remarquera qu'une variante d'un code métamorphe est un mot v issu de : $L(G)$ pour la première définition et de $L(L(G))$ pour la seconde. Rien n'empêche alors de poursuivre cette abstraction sur les langages formels, en envisageant par exemple la mutation de méta-grammaires, c'est-à-dire le cas où $v \in L(L(L(G)))$, et ainsi de suite. C'est pourquoi nous proposons d'unifier les définitions existantes des codes métamorphes en introduisant une hiérarchie au sein du métamorphisme.

Définition 12. *(Hiérarchie du métamorphisme). Un code auto-reproducteur est dit métamorphe d'ordre n si ses variantes v correspondent aux mots du langage $L^n(G)$ où G désigne une grammaire formelle et la notation $L^n(G)$ est définie par :*

$$\begin{cases} L^0(G) = L(G), \\ \forall n \in \mathbb{N}, n \neq 0, L^n(G) = L(L^{n-1}(G)). \end{cases}$$

Ainsi la définition de Ször [158] correspond à un métamorphisme d'ordre 0 alors que celles de Zuo *et al.* et de Filiol correspondent à un métamorphisme d'ordre 1. Aujourd'hui, seule la complexité de détection du métamorphisme d'ordre 0 est connue.

Perspective 1 (Étude de la hiérarchie du métamorphisme).

L'étude des ordres non nuls du métamorphisme reste à conduire aussi bien sur le plan théorique que pratique.

Les grammaires formelles constituent un modèle théorique efficace pour décrire la complexité des mutations de code intervenant dans le cadre du métamorphisme. Toutefois, leur utilisation dans la pratique est compliquée par la contrainte sémantique du code qu'elles génèrent : les programmes (mots) produits doivent conserver la même fonctionnalité. Ainsi, même pour le métamorphisme d'ordre 0, la description des mutations de code au moyen de grammaires formelles peut s'avérer une tâche fastidieuse, comme le montrent les travaux de Zbitskiy [189] ainsi que les règles de réécritures employées par le virus METAPHOR (voir annexe B). Dans ce cas, comment sont construites ces mutations dans la pratique ? C'est afin de répondre à cette question que nous envisageons ces mutations en termes d'obscurcissement de code dans la section qui suit.

1.2 Techniques de mutation par obscurcissement de code

Les mutations employées par les codes évolutifs visent à modifier la forme d'un programme tout en lui garantissant la même fonctionnalité. Ces modifications complexifient la structure du programme dans le but de le rendre plus difficile à détecter. L'étude de ces fonctions de transformation constitue une discipline à part entière dénommée obscurcissement de code [5], que nous nous proposons d'étudier dans cette section.

1.2.1 Qu'est-ce que l'obscurcissement de code ?

L'obscurcissement de code est issu d'un besoin de protection de la propriété intellectuelle dans le cadre du déploiement d'algorithmes. En effet, l'utilisation de plus en plus fréquente de langages de haut niveau tels que Java ou encore ceux compatibles avec le « *framework* » .NET [6] conduit à la production de fichiers intermédiaires (fichiers .class en java et « *assembly .NET* » pour .NET) contenant du « *bytecode* ». Le terme « *bytecode* » désigne un langage intermédiaire qui n'est pas directement exécutable sur une architecture matérielle. Ce langage binaire est interprété par un programme particulier, appelé machine virtuelle, pour permettre d'abstraire l'exécution de l'architecture matérielle. Indépendamment du code qu'ils contiennent, ces fichiers intermédiaires présentent aussi toutes les informations initialement contenues dans les sources d'origine telles que les classes, les variables, les fonctions et méthodes, ainsi que divers symboles, etc. Ces informations rendent alors la compréhension des programmes beaucoup plus simple que celle des fichiers exécutables contenant uniquement du code machine.

5. Le terme anglais correspondant « *obfuscation* » se retrouve fréquemment dans la littérature spécialisée. Ne possédant pas d'équivalent officiel à ce terme, nous adoptons la recommandation de l'office québécois de la langue française (OQLF) pour désigner cette discipline par l'expression « obscurcissement de code ».

6. Les principaux langages de programmation compatibles avec le « *framework* » .NET sont : ADA, APL, C#, C++, Cobol, Eiffel, Fortran, Haskell, ML, J#, JScript, Mercury, Oberon, Objective Caml, Oz, Pascal, Perl, Python, Scheme, Smalltalk, Visual Basic, etc.

C'est dans le but de protéger ces informations contenues dans les fichiers intermédiaires que s'est développée l'obscurcissement de code. Son objectif consiste à rendre le programme transformé le plus incompréhensible possible. De nombreux travaux traitent de cette problématique mais avec des objectifs différents. Des travaux théoriques tentent de définir les résultats possibles concernant l'obscurcissement de code [25, 13, 119, 8, 75]. D'autres travaux adoptent une approche plus pratique afin de proposer à la fois des techniques de transformation de code ainsi que leurs critères d'évaluation associés [44, 45, 46]. Finalement, des approches sont proposées pour le tatouage (« *watermarking* ») [47] ainsi que la protection des logiciels [115, 117, 135, 185]. Nous présentons ici uniquement les principaux résultats et approches en rapport avec le métamorphisme.

La définition la plus générale de l'obscurcissement de code est présentée dans les travaux de Collberg *et al.*

Définition 13. *(obscurcissement de code [44, 46]). Soit $\mathcal{T} : \mathcal{P} \to \mathcal{P}$ une fonction transformant un programme P en un programme P'. $\mathcal{T}$ est une transformation d'obscurcissement de code si $\mathcal{T}(P)$ possède le même comportement observable que P, sachant que :*

- *si le programme P ne se termine pas ou s'il se termine avec une erreur, alors le programme $\mathcal{T}(P)$ peut éventuellement se terminer ou non ;*
- *dans le cas contraire (cas où le programme P se termine), le programme $\mathcal{T}(P)$ se termine aussi en fournissant la même sortie que P.*

Partant de cette définition, l'ensemble des codes évolutifs, obtenus par mutations successives à partir d'une souche originale V_0, peut alors se représenter de manière simplifiée comme $\{V_{i,i\in\mathbb{N}}$ tels que $V_{i+1} = \mathcal{T}(V_i)\}$ [7]. Bien qu'introduisant le principe de base de l'obscurcissement de code, la définition 13 ne précise pas la propriété recherchée « d'opacité du programme » transformé. C'est pourquoi nous présentons maintenant les différents formalismes et résultats concernant ces transformations. Dès lors, la question qui nous intéresse est de définir quelles sont les possibilités offertes par l'obscurcissement de code.

1.2.2 Principaux résultats théoriques sur l'obscurcissement de code

Un obscurcisseur de code peut être vu comme une machine de Turing probabiliste à temps polynomial (« *Probabilistic Polynomial time Turing machine* » ou PPT) [137]. Une telle machine prend en entrée un programme initial P pour produire en sortie un programme obscurci $P' = \mathcal{O}(P)$ vérifiant les trois propriétés suivantes :

- l'**équivalence fonctionnelle**, qui signifie que pour les mêmes entrées, si P fournit une sortie alors P' fournit la même sortie ;

7. On remarquera que ce modèle de réplication avec évolution du code n'est pas réaliste. En effet, une transformation d'obscurcissement de code implique généralement une augmentation de la taille du programme sur laquelle elle est appliquée. Cet aspect est détaillé en section 1.2.7 qui expose le fonctionnement d'un code métamorphe.

- l'**accroissement polynomial**, qui traduit le fait que les complexités temporelles et spatiales de P' sont polynomiales par rapport à celles de P. Cette propriété correspond plus à une contrainte pratique que doivent respecter les obscurcisseurs de code pour ne pas trop dégrader les performances du programme original ;
- la propriété de **« boîte noire virtuelle »**, qui consiste à empêcher un attaquant d'obtenir l'algorithme décrit par la programme original P à partir du programme obscurci P'.

L'existence d'obscurcisseurs de code respectant les trois propriétés précédentes représenterait une avancée considérable, non seulement en termes de protection des logiciels, mais aussi dans le domaine de la cryptographie avec la résolution du problème ouvert du chiffrement homomorphique à clés publiques [147] ainsi que celui de la transformation d'un algorithme de chiffrement symétrique en chiffrement asymétrique [52]. Pour plus de détails sur les applications possibles de l'obscurcissement de code voir les travaux de Barak *et al.* [8]. Nous limiterons notre propos au domaine de la protection logiciel en lien avec notre problématique du métamorphisme.

1.2.2.1 Impossibilité de l'obscurcissement de code dans le cas général

Nous introduisons maintenant les notations nécessaires à l'établissement des principaux résultats. Pour tout programme A, $|A|$ désigne la taille de A. La notation $A(1^t)$ désigne le résultat de l'algorithme A pour une exécution de durée t. On dira que le programme S dispose d'un accès par oracle au programme P, ce que l'on notera S^P, lorsque S fournit uniquement la valeur de sortie de P pour une entrée donnée. En d'autres termes, l'accès par oracle à un programme signifie que seuls les couples entrées/sorties sont observables. La première définition formelle de l'obscurcissement de code est celle proposée par Barak *et al.* [8].

Définition 14. *(TM obscurcisseur [8]). Un algorithme probabiliste $\mathcal{O}$ est un obscurcisseur de machine de Turing (TM obscurcisseur) si :*

- *(équivalence fonctionnelle) pour tout $M \in \mathcal{M}$, $\mathcal{O}(M)$ décrit une machine de Turing qui calcule la même fonction que M ;*
- *(accroissement polynomial) il existe un polynôme p tel que pour tout $M \in \mathcal{M}$, $|\mathcal{O}(M)| \leq p(|M|)$, et si M s'arrête après t itérations pour une entrée x, alors $\mathcal{O}(M)$ s'arrête en $p(t)$ étapes pour x ;*
- *(« boîte noire virtuelle ») pour toute PPT A, il existe une PPT S telle que pour tout $M \in \mathcal{M}$,*

$$Pr[A(\mathcal{O}(M))] \approx Pr[S^M(1^{|M|})]$$

Si les deux premières contraintes (équivalence fonctionnelle et l'accroissement polynomial) sont suffisamment explicites, la dernière propriété nécessite plus d'explications. La propriété de « boîte noire virtuelle » traduit le fait que la distribution des sorties obtenues par un attaquant (algorithme) A agissant sur le programme obscurci $\mathcal{O}(M)$ est la même (à une fonction négligeable près) que celle obtenue par un simulateur S ayant un accès par oracle au programme M.

Autrement dit, les seules informations que l'on peut obtenir de $\mathcal{O}(M)$ correspondent à l'observation de la sortie produite pour une entrée donnée.

Malheureusement, le principal résultat obtenu dans [8] est la preuve de l'existence de fonctions que l'on ne peut obscurcir conformément à la définition 14.

Théorème 7. *(Impossibilité de l'obscurcissement de code « boîte noire virtuelle » [8]). Il n'existe aucun obscurcisseur de code générique satisfaisant la définition 14.*

La preuve de ce théorème repose sur la construction d'un famille de programmes $\mathcal{P}$ pour laquelle l'algorithme de tout programme P' calculant la même fonction qu'un programme P de $\mathcal{P}$ peut être reconstruit. Ainsi, il est impossible de concevoir un programme générique capable de protéger tout autre programme en empêchant de révéler plus d'informations que celles obtenues par les observations des entrées/sorties.

On remarquera que la propriété de « boîte noire virtuelle » est une hypothèse forte selon laquelle un programme obscurci ne doit pas révéler plus d'informations que celle fournie par l'observation de ses entrées/sorties. Les travaux de Goldwasser et Rothblum [76] proposent une définition moins forte de l'obscurcissement de code appelée « *meilleur obscurcissement (de code) possible* ». Cette définition de l'obscurcissement de code repose sur l'hypothèse moins restrictive que le programme obscurci ne doit pas fournir plus d'informations que tout autre programme de même taille et de fonctionnalité équivalente. En ce sens, cette approche correspond littéralement au meilleur obscurcissement possible. Ce résultat est obtenu dans le modèle de l'oracle aléatoire introduit par Fiat et Shamir [61]. Dans ce modèle, tous les membres ont accès à une fonction aléatoire publique $\mathcal{R}$ appelée oracle aléatoire. Chaque participant peut interroger cet oracle $\mathcal{R}$ en différents points. Sous ses conditions, Goldwasser et Rothblum obtiennent le principal résultat suivant :

Théorème 8. *(Impossibilité du « meilleur obscurcissement possible » [76]). Sous le modèle de l'oracle aléatoire, il n'existe pas d'obscurcisseur de code générique satisfiant la définition du « meilleur obscurcissement possible ».*

1.2.2.2 Possibilité d'obscurcissement de fonctions particulières

Le premier résultat positif concernant l'obscurcissement de code est celui de Lynn *et al.* [119] qui montre que le contrôle d'accès, tel qu'il est communément pratiqué par comparaison du haché d'un mot de passe, correspond mathématiquement à un obscurcissement d'une fonction point sous le modèle de l'oracle aléatoire. Une fonction point $P_\alpha : \{0,1\}^k \to \{0,1\}$ est définie par :

$$\begin{cases} P_\alpha(x) = 1 & \text{si } x = \alpha, \\ P_\alpha(x) = 0 & \text{sinon.} \end{cases}$$

P_α correspond bien à une fonction d'authentification par mot de passe α. En effet, l'authentification est correcte ($P_\alpha(x) = 1$), si le mot de passe x fourni correspond bien à celui attendu α, et incorrecte dans le cas contraire. Toutefois,

un programme qui implémente directement cette fonction n'est pas obscurci puisqu'il contient la donnée secrète : le mot de passe α. C'est pour obscurcir un tel programme que l'on utilise le modèle de l'oracle aléatoire employé aussi dans la conception de protocoles cryptographiques.

Théorème 9. *(Possibilité d'obscurcissement des fonctions point. [119].) Pour un oracle aléatoire* $\mathcal{R} : \{0,1\}^* \rightarrow \{0,1\}^{2k}$*, soit* $\mathcal{O}^{\mathcal{R}}(P_\alpha)$ *un programme qui contient* $r = \mathcal{R}(\alpha)$ *et qui pour l'entrée* $x \in \{0,1\}^k$ *retourne la valeur* 1 *si* $\mathcal{R}(x) = r$ *et* 0 *sinon. Le programme* $\mathcal{O}^{\mathcal{R}}(P_\alpha)$ *est un obscurcissement de la fonction point* P_α*.*

L'implémentation réelle du modèle idéal de l'oracle aléatoire est souvent réalisée au moyen de fonctions de hachage cryptographiques. La seule information sur α contenue dans le programme obscurci est alors la valeur $r = \mathcal{R}(\alpha)$ qui correspond au haché du mot de passe α. La sécurité de l'obscurcissement repose alors sur la sécurité de la fonction de hachage employée.

1.2.2.3 Possibilité d'un obscurcissement temporel

Un autre résultat positif peut être obtenu en se focalisant sur les aspects temporels de l'obscurcissement de code. Au lieu de rechercher une protection parfaite contre la rétro-conception, l'idée consiste à protéger le code efficacement durant un temps donné. C'est ainsi que le τ-obscurcissement de code a été envisagé :

Définition 15. *(τ-obscurcisseur de code [13]). Un algorithme probabiliste* $\mathcal{O}$ *est un τ-obscurcisseur de code s'il satisfait, en plus des propriétés de fonctionnalité et d'accroissement polynomiale de la définition 14, la nouvelle propriété de « boîte noire virtuelle » pour la durée* τ *:*

$$\forall P \in \mathcal{M}, Pr[A(\mathcal{O}(P), 1^{\tau \times t(\mathcal{O}(P))})] \approx Pr[S^P(1^{|P|})]$$

Cette nouvelle propriété de « boîte noire virtuelle » traduit le fait que tout ce qui peut être calculé en temps inférieur à $\tau \times t(\mathcal{O}(P))$, où $t(\mathcal{O}(P))$ désigne le temps d'obscurcissement de P, est effectivement calculable via un accès par oracle au programme P. L'avantage de cette définition moins contraignante que celle de [8] est qu'elle conduit au résultat positif suivant :

Théorème 10. *(Existence de τ-obscurcisseur de code [13]). Les τ-obscurcisseurs de code tels que présentés en définition 15 existent.*

Ce résultat se retrouve sous une autre forme dans les travaux de Zuo *et al.* [196] qui prouvent l'existence de virus dont l'exécution ainsi que la détection peut être arbitrairement longues.

1.2.3 Critères d'évaluation de l'efficacité d'un obscurcissement de code

L'obscurcissement de code constitue souvent un ultime rempart lorsque tous les autres moyens de protection mis en place se sont révélés infructueux. La

difficulté réside alors dans l'évaluation des solutions pratiques de protections apportées. C'est pourquoi des critères d'évaluation se doivent d'être définis avant la conception des transformations de code.

Collberg *et al.* proposent de caractériser l'efficacité empirique de l'obscurcissement de code [44, 46] au moyen des critères de mesures suivants :

- la **puissance**, qui désigne la complexité rajoutée au programme obscurci et qui traduit la difficulté de compréhension du code pour un analyste humain ;
- la **résilience**, qui représente la difficulté d'inversion de l'obscurcissement de code pour un outil automatique ;
- le **coût**, qui mesure l'augmentation des ressources nécessaires à l'exécution du programme obscurci par rapport au programme d'origine.

Le domaine de l'ingénierie logicielle propose diverses mesures conçues pour quantifier la qualité d'un programme. Certaines sont utilisées pour évaluer la puissance d'une transformation comme : la longueur du programme [81], la complexité cyclomatique [122], la complexité du flot de données [136] ou encore celle de la structure des données [131]. La définition de la puissance d'une transformation proposée par Collberg *et al.* est la suivante :

Définition 16. *(puissance [44]). Soit $\mathcal{T}$ une transformation d'obscurcissement de code et P un programme quelconque. Étant donné une mesure E applicable au programme P, la puissance de la transformation $\mathcal{T}$ par rapport au programme P, notée $\mathcal{T}_{pot}$ se définit par :*

$$\mathcal{T}_{pot}(P) = \frac{E\left(\mathcal{T}(P)\right)}{E(P)} - 1.$$

Une transformation $\mathcal{T}$ sera alors dite puissante si $\mathcal{T}_{pot} > 0$.

Bien que la puissance soit un bon indicateur de la difficulté de compréhension d'un code du point de vue de l'analyse humaine, elle ne permet pas de qualifier pleinement l'efficacité d'une transformation d'obscurcissement de code dans sa généralité. Afin de mesurer la difficulté, pour un outil automatique, à remonter au programme d'origine P étant donné un programme obscurci $\mathcal{T}(P)$, la notion de résilience a été définie comme une fonction de deux efforts. D'une part le temps nécessaire au développement de l'outil dédié à cette transformation inverse noté $\mathcal{T}_1$. D'autre part, le temps pris par cet outil pour inverser la transformation d'obscurcissement de code noté $\mathcal{T}_2$.

Définition 17. *(résilience [44]). Soit $\mathcal{T}$ une transformation d'obscurcissement de code et P un programme quelconque. La résilience de la transformation $\mathcal{T}$ par rapport au programme P, notée $\mathcal{T}_{res}(P)$ se définit par :*

$$\mathcal{T}_{res}(P) = R(\mathcal{T}_1, \mathcal{T}_2),$$

où R désigne une matrice d'effort représentée sur la figure 1.4.

La dernière mesure communément utilisée dans l'évaluation de l'obscurcissement de code est le coût, qui représente l'augmentation de ressources nécessaires

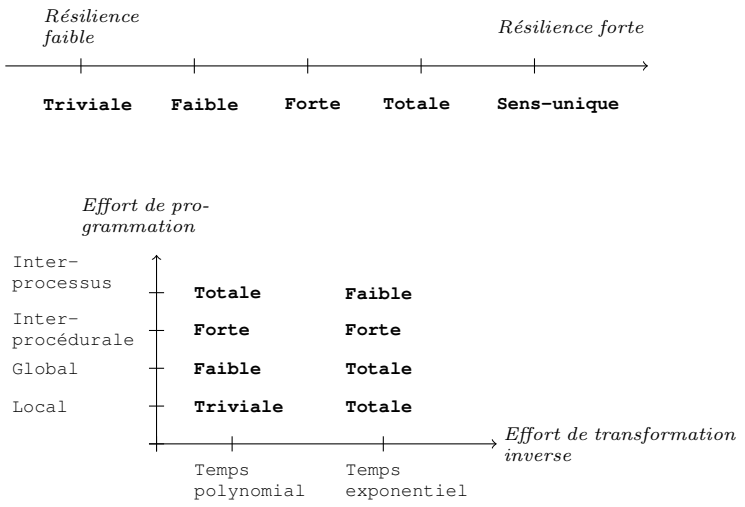


FIGURE 1.4 – Matrice d'effort R, extraite de [44], permettant l'évaluation de la résilience. L'échelle de résilience est représentée en haut de la figure.

à l'exécution du programme obscurci $\mathcal{O}(\mathcal{P})$ par rapport au programme d'origine P.

Définition 18. *(coût [44]). Soit $\mathcal{T}$ une transformation d'obscurcissement de code et P un programme quelconque. Étant donné $\mathcal{C}$ une fonction qui a tout programme associe sa complexité (temporelle ou spatiale), le coût de la transformation $\mathcal{T}$ par rapport au programme P, noté $\mathcal{T}_{cost}(P)$ se définit par :*

$$\mathcal{T}_{cost}(P) = \frac{\mathcal{C}(\mathcal{T}(P))}{\mathcal{C}(P)}.$$

Dans un contexte particulier de détection de codes malveillants, l'objectif consiste à inverser ces techniques d'obscurcissement de code afin de retrouver des caractéristiques communes à des codes malveillants connus. Aussi, dans le cadre du métamorphisme, la résilience apparait alors comme le critère à maximiser afin d'éviter d'être détecté.

1.2.4 Taxonomie des techniques de base de l'obscurcissement de code

Le but de cette section est de présenter les principales techniques employées en ingénierie logicielle afin d'obscurcir un programme. Les transformations utilisées dans le cadre des codes malveillants seront abordées en section 1.2.6. La classification proposée par Collberg *et al.* [44, 46] constitue une référence en la matière. Bien qu'elle cible les langages de haut niveau et plus précisément le langage *java*, les transformations qui y sont présentées s'appliquent dans le plupart

des cas aux autres langages. Les transformations répertoriées se répartissent en quatre catégories :

- *l'obscurcissement des symboles* tels que les noms des données et des objets ;
- *l'obscurcissement du flot de données*, c'est-à-dire la façon dont les informations transitent au sein d'un programme ;
- *l'obscurcissement du flot de contrôle*, c'est-à-dire l'ordre dans lequel les instructions d'un programme sont exécutées ;
- *l'obscurcissement préventif* qui vise à se prémunir des outils d'analyse.

Afin de présenter ces transformations de code, nous introduisons la notion de prédicat opaque qui constitue un élément clé pour la majorité des approches d'obscurcissement de code.

1.2.4.1 Constructions de prédicats opaques pour l'obscurcissement de code

Les transformations visant à modifier le flot de contrôle d'un programme impliquent une augmentation de la complexité calculatoire, ce qui influence directement le temps d'exécution du programme obscurci. Une façon d'augmenter la résilience d'une transformation d'obscurcissement de code à moindre coût réside dans l'emploi de prédicats opaques tels que définis par Collberg *et al.*

Définition 19. *(Prédicat opaque [46]). Un prédicat est dit opaque s'il est de valeur constante, connue au moment de l'obscurcissement de code, mais dont la détermination est difficile pour un outil d'analyse automatique.*

L'utilisation de tels prédicats se retrouvent dans la majorité des approches d'obscurcissement de code [44, 46, 115, 135, 175, 176]. Divers techniques ont été proposées jusqu'alors pour construire ce type de prédicats [46]. Nous présentons ici les principales.

1.2.4.1.a Prédicats opaques issus de la théorie des nombres

Certains résultats d'algèbre connus peuvent s'avérer difficiles à obtenir pour un outil d'analyse automatique. La figure 1.5 présente des exemples de prédicats opaques extraits de [4]. Ces prédicats issus de l'arithmétique sont toujours vrais. Le prédicat 1.7 expose une équation algébrique de deux variables qui n'a pas de solution entière. Les prédicats 1.8 et 1.9 reposent sur la divisibilité d'expressions algébriques. Le prédicat 1.10 stipule que la somme des nombres impairs jusqu'à $2x-1$ est égale à x^2. Le prédicat 1.11 stipule que la partie entière inférieure de la moitié de tout entier élevé au carré est pair.

Les travaux d'Arboit [4] proposent la construction de prédicats opaques paramétrés pour un nombre premier p donné, en considérant par exemple les solutions de l'équation $y^2 \equiv a[p]$ pour a un entier naturel quelconque et $p = 4x + 3$. Ces solutions se présentent sous la forme $y = \pm a^{x+1}$. La famille de prédicats opaques correspondante est alors $\left(a^{x+1}\right)^2 \equiv a[p]$. De manière similaire, les solutions de l'équation $y^2 \equiv a[p]$ avec $p = 8x + 5$, sont de la forme $y = \pm\frac{(4a)^{x+1}}{2}$. La famille de prédicats opaques associée correspond à $\left(\frac{(4a)^{x+1}}{2}\right)^2 \equiv a[p]$.

$$\forall (x,y) \in \mathbb{Z}^2, 7y^2 - 1 \neq x^2 \tag{1.7}$$
$$\forall x \in \mathbb{Z}, 3 | (x^3 - x) \tag{1.8}$$
$$\forall x \in \mathbb{Z}, 2 | x \vee 8 | (x^2 - 1) \tag{1.9}$$
$$\forall x \in \mathbb{Z}, \sum_{i=1, i \neq 0[2]}^{2x-1} i = x^2 \tag{1.10}$$
$$\forall x \in \mathbb{N}, 2 \left| \left\lfloor \frac{x^2}{2} \right\rfloor \right. \tag{1.11}$$

FIGURE 1.5 – Exemples de prédicats opaques arithmétiques toujours vrais.

1.2.4.1.b Prédicats opaques par incorporation de problèmes difficiles pour l'analyse statique

L'analyse statique est définie par Landi [107] comme *le processus visant à extraire les informations sémantiques d'un programme au moment de la compilation*. Dans le cadre de l'étude des codes malveillants, l'analyse statique consiste à déterminer certaines propriétés d'un programme malveillant sans en fixer les valeurs d'entrée et sans recourir à l'exécution du code. L'avantage de l'analyse statique, comme nous le verrons plus en détail en section 1.3.2, est de permettre la détection d'une application malveillante avant son exécution.

Un problème classiquement étudié en analyse statique est la détermination des alias. On dit qu'un alias apparait à un certain point de l'exécution d'un programme lorsque au moins deux noms existent pour désigner une même zone mémoire. Par exemple, en langage C, l'instruction `p=&v` crée un alias entre `*p` et `v`. Plusieurs versions de l'analyse statique des alias sont démontrées NP-difficiles [84] et même indécidables dans le cas des langages autorisant des branchements conditionnels, des boucles, des allocations dynamiques de mémoire ainsi que des structures récursives [107, 143]. Aussi, l'utilisation du problème de la détermination des alias peut considérablement impacter la précision de l'analyse statique.

1.2.4.1.c Prédicats opaques par utilisation de conjectures mathématiques

La création de prédicats opaques peut aussi reposer sur des conjectures mathématiques. Pour cela, il suffit de s'assurer que la conjecture est valide sur l'espace de calcul du programme qui les emploie. Ainsi, la conjecture de Collatz [80], connue aussi sous le nom de conjecture de Syracuse, est souvent utilisée pour sa simplicité de mise en œuvre. On considère pour cela la suite (S_n) définie

par :

$$S_0 \in \mathbb{N} \text{ et } \forall n \in \mathbb{N}, n \neq 0, S_{n+1} = \begin{cases} \frac{S_n}{2} & \text{si } S_n \text{ est pair,} \\ 3S_n + 1 & \text{sinon.} \end{cases}$$

La conjecture de Collatz stipule que, quelle que soit la valeur initiale n_0, la suite S_n finit par boucler sur les valeurs 1, 4, 2 à partir d'un certain rang. Cette propriété a été vérifiée expérimentalement pour n_0 variant jusqu'à $4,899 \times 10^{18}$ mais reste aujourd'hui encore à démontrer. La figure 1.1 illustre l'utilisation de cette conjecture en tant que prédicat opaque sous la forme d'une boucle se terminant quand la suite S_n atteint la valeur 1. Cette boucle se termine effectivement pour toute valeur entière de n de 32 bits choisie aléatoirement.

```
1 n = random(1,2^32);
2 do
3  if (n%2 != 0)
4   n=3*n+1;
5  else
6   n=n/2;
7 while (n>1);
```

Listing 1.1 – Exemple d'utilisation de la conjecture de Collatz comme prédicat opaque extrait de [44].

Grâce aux prédicats opaques, nous disposons maintenant de l'outillage suffisant pour présenter les principales techniques de base concernant l'obscurcissement de code répertoriées par Collberg *et al.* [44, 46].

1.2.4.2 L'obscurcissement des symboles

L'analyse des symboles d'un programme est une source d'informations directement exploitable pour la compréhension du code. Des informations comme les noms de classes pour les langages objet, noms de fonctions, de méthodes ou encore de procédures ainsi que les chaînes de caractères sont autant de sources de compréhension de la structuration et de la fonctionnalité du code étudié. Pour les langages dont la chaîne de compilation conserve les informations de nommage et notamment pour les langages interprétés ou managés, la première étape d'obscurcissement consiste à supprimer ces symboles. Il s'agit bien sûr d'une transformation à sens unique puisque l'information de nommage est définitivement perdue.

Un exemple de ce type d'obscurcissement est donné en figure 1.2 extrait de [158, chapitre 7], qui présente une version du virus MSIL/Gastropod dans laquelle les noms de classes et de méthodes obscurcis sont représentés en gras. Il s'agit d'un code écrit en « *Microsoft Intermediate Language* » (MSIL) qui représente le constructeur de la classe `.cctor` du virus. Ce virus utilise l'espace de nommage `System.Reflection.Emit` pour générer un nouveau binaire avec des noms aléatoires de tailles comprises entre 6 et 15 caractères.

```
.method private static hidebysig specialname void .cctor()
{
  ldstr "[.NET.Snail - sample CLR virus (c) whale 2004 ]"
  stsfld class System.String Ylojnc.lgxmAxA::WaclNvK
  nop
  ldc.i4.6
  ldc.i4.s 0xF
  call int32 [mscorlib]System.Environment::get_TickCount()
  nop
  newobj void nljvKpqb::.ctor(int32 len1, int32 len2, int32 seed)
  stsfld class nljvKpqb Ylojnc.lgxmAxA::XxnArefPizsour
  call int32 [mscorlib]System.Environment::get_TickCount()
  nop
  newobj void [mscorlib]System.Random::.ctor(int32)
  stsfld class [mscorlib]System.Random Ylojnc.lgxmAxA::aajqebjtoBxjf
  ret
}
```

Listing 1.2 – Illustration de l'obscurcissement des noms de classes dans le virus MSIL/Gastropod.

1.2.4.3 L'obscurcissement du flot de données

Le flot de données désigne la propagation des données au sein d'un programme. L'analyse du flot de données consiste alors à collecter des informations sur la façon dont les variables sont utilisées au sein d'un programme. Cette analyse permet notamment de déterminer la valeur des paramètres lors d'un appel à une fonction d'une interface de programmation (« *Application Programming Interface* » ou API) qui constitue un point de convergence obligatoire pour toute application. L'obscurcissement du flot de données peut se décomposer en trois sous-parties :

1. le **stockage et le codage**, qui consiste à changer la représentation et la façon d'utiliser les variables au sein d'un programme, ce qui peut se réaliser de diverses manières :
   - par transformation des scalaires en objets plus complexes. Par exemple définir des méthodes pour les calculs sur des éléments d'objets plutôt que des calculs directs sur des scalaires. On peut aussi en fonction du langage surcharger les opérateurs pour obtenir le même effet.
   - par conversion de données statiques en procédure, par exemple la valeur -1 peut être obscurcie suivant la formule $\frac{b+1-a}{cos(a+\pi-b)}$ pour $a = b$ en supposant que la précision décimale le permette ;
   - par changement du codage des valeurs. Par exemple permuter les valeurs des variables `TRUE` et `FALSE` ;
   - par modification de la durée de vie d'une variable, par exemple le passage du local au global ou alors du local à celui d'un élément d'un objet ;
2. l'**agrégation (de données)**, qui modifie la manière dont sont regroupées les données dans le but de compliquer la restauration des structures de données initiales du programme. Par exemple, ces techniques peuvent dé-

couper ou bien fusionner des tableau pour en compliquer l'accès, en transformant des tableaux à 2 dimensions en tableaux à une seule dimension et réciproquement. La puissance de ces transformations est élevée puisque, soit de nouvelles données sont rajoutées, soit des anciennes structures de données sont supprimées.

3. le **ré-agencement**, qui consiste à réordonner la structure interne des objets. Par exemple, ces transformations peuvent ré-agencer des tableaux en utilisant une fonction $f(i)$ pour déterminer la position du i^{e} élément du tableau alors que cet élément est généralement stocké en i^{e} position dans le tableau initial. La puissance de ces transformations est faible mais leur résilience peut être élevée [192].

1.2.4.4 L'obscurcissement du flot de contrôle

Le flot de contrôle désigne la séquence d'instructions exécutée par un programme. Obscurcir le flot de contrôle consiste alors à compliquer la détermination de l'enchaînement logique des instructions. Trois classes de transformations existent :

1. l'**insertion de « code mort »** . Cette technique consiste à introduire du code sémantiquement inutile ou alors qui ne sera jamais exécuté, par exemple, après une instruction de branchement conditionnelle (ou équivalente). Dans ce dernier cas, il s'agit en fait d'une instruction de branchement inconditionnelle. L'objectif consiste à induire en erreur un analyste ;
2. l'**obscurcissement calculatoire**. Des calculs supplémentaires sont rajoutés afin de compliquer le flot de contrôle initial du programme. Cette technique peut se décomposer en :
   - conversion de graphes réductibles en graphes irréductibles. Par exemple, supposons qu'un langage de haut niveau ne possède pas l'instruction `goto` alors que le « *bytecode* » généré utilise cette instruction. Dans ce cas le graphe de flot de contrôle représenté par source est toujours réductible [2] alors que le processus d'obscurcissement de code peut produire des graphes de flot de contrôle non-réductible.
   - extension des conditions de bouclage. Ici, l'utilisation de prédicats opaques permet de compliquer un critère d'arrêt d'une boucle.
   - emploi d'une table d'interprétation. Cette technique est une des plus efficace mais son coût est élevé. L'idée consiste à convertir une portion de code dans un « *bytecode* » de machine virtuelle. Ce code est alors interprété au moment de l'exécution par une machine virtuelle comprise dans le programme obscurci.
3. l'**agrégation**, qui consiste à découper et à fusionner des portions de code. Par exemple il est possible de substituer à l'appel d'une procédure le code de cette même procédure (cette transformation correspond au mot clé `inline` du langage C++). De manière similaire, une autre transformation utilisée consiste, par exemple, à regrouper au sein d'une même procédure des initialisations de variables.

4. le **ré-agencement du code**. Le but visé ici est de rendre aléatoire le placement du code tout en assurant la même fonctionnalité. Le code peut ainsi être simplement permuté lorsque les instructions sont indépendantes. Cette technique de ré-agencement est particulièrement efficace lorsqu'elle utilise des prédicats opaques conditionnant des branchements. Dans ce cas, le flot de contrôle est conditionné par ces branchements et donc la détermination de ces prédicats.

1.2.4.5 L'obscurcissement préventif

Ce type d'obscurcissement de code consiste à protéger un programme contre des outils dédiés à sa rétro-conception, comme les débogueurs, les désassembleurs, les décompilateurs, ou encore les environnements d'analyse. Par exemple, l'approche d'obscurcissement de code de Linn et Debray, que nous présentons en section 1.2.5.4 a pour objectif de perturber le désassemblage d'un binaire.

1.2.5 Principales Approches d'obscurcissement de code par combinaisons des techniques de base

Les différentes techniques présentées précédemment ont été employées et combinées dans les principales approches que nous résumons ici.

1.2.5.1 Approche de Collberg *et al.*

De manière simplifiée, l'approche d'obscurcissement de code présentée dans les travaux de Collberg *et al.* [44, 46] reprend l'ensemble des transformations décrites. Les algorithmes proposés peuvent se résumer en un algorithme générique comportant trois étapes. Dans un premier temps, une portion de code du programme à obscurcir est sélectionnée. Par portion de code les auteurs désignent aussi bien une classe, qu'une méthode, que des bibliothèques standards, qu'une portion de code composée exclusivement d'instructions séquentielles, etc. Dans un deuxième temps, une transformation d'obscurcissement de code est sélectionnée en adéquation avec la portion de code choisie. Enfin, cette transformation est appliquée à la portion de code sélectionnée. Le résultat ainsi obtenu est substitué à la portion de code d'origine. Plusieurs raffinements de cette approche sont proposés, notamment en ce qui concerne les fonctions de sélection. Dans tous les cas, l'approche globale proposée par Collberg *et al* souffre d'un manque de justifications théoriques concernant l'efficacité des transformations appliquées.

1.2.5.2 Approche de Wang *et al.* complétée par Ogiso *et al.*

Contrairement à Collber *et al.*, les travaux de Wang *et al.* [175, 176] apportent une preuve de résilience de leurs approches qui reposent sur des problèmes difficiles à résoudre en analyse statique. Le constat de départ est le suivant : les attaques ciblant des codes obscurcis s'appuient sur des informations sémantiques du programme généralement obtenues par analyse statique.

L'utilisation de transformations s'appuyant sur la difficulté de la détermination exacte des alias impacte alors directement la précision de l'analyse statique. Le principe proposé consiste à rendre l'analyse du flot de contrôle d'un programme dépendante de celle du flot de données. En particulier, les auteurs proposent de découper un programme en blocs d'instructions séquentielles afin que chaque bloc de code puisse potentiellement correspondre au successeur ou au prédécesseur de tout autre bloc de code. Le flot de contrôle est alors assuré par une fonction de distribution dynamique dénommée dispatcheur (« *dispatcher* »). Chaque bloc de code exécuté se termine par des manipulations complexes de pointeurs sur une variable utilisée par le dispatcheur afin d'assurer le bon séquencement des blocs à exécuter.

L'approche de Wang *et al.* se limite à de l'obscurcissement de code au sein d'une même procédure (l'approche est dite intra-procédurale). Une extension logique portant sur le même type de transformation mais appliquée, cette fois-ci à plusieurs procédures est proposée par Ogiso *et al.* [135]. La preuve de cette approche intra-procédurale repose sur celle de Wang *et al.*.

Ces deux approches partagent toutefois une même lacune : toute la sécurité repose sur le dispatcheur qui représente une procédure facilement identifiable. Les preuves de résilience sont fournies dans un cadre d'analyse statique. Dans le cadre d'une analyse dynamique, le dispatcheur représente un point de convergence rapidement identifiable.

1.2.5.3 Approche de Chow *et al.*

Un autre résultat prometteur est celui exposé par Chow *et al.* [32]. Il consiste en l'utilisation de problèmes combinatoires complexes dont les solutions sont connues du programme d'obscurcissement de code. Ces problèmes combinatoires sont insérés dans un programme au moyen de transformations visant à conserver la sémantique initiale du programme. L'objectif est de protéger une propriété P d'un programme de sorte que déterminer P revient à trouver une solution du problème combinatoire utilisé. Les auteurs montrent alors que la transformation inverse de celle appliquée est un problème PSPACE-complet.

1.2.5.4 Approche de Linn et Debray

Linn et Debray [115] proposent une approche visant à obscurcir des programmes à partir de leur forme binaire. L'idée présentée repose sur la difficulté du désassemblage de binaires, c'est-à-dire la traduction du code machine dans un langage assembleur, compréhensible par un être humain. Une hypothèse couramment faite lors du désassemblage est que tout appel de fonction (instruction `call`) est supposé retourner à l'instruction suivante après exécution de la fonction appelée[8]. Les auteurs proposent de remplacer les branchements inconditionnels par un appel à une fonction de distribution. Cette fonction de distribution f ne se comporte pas comme une fonction classique car une fois terminée, l'exécution ne se poursuit pas après l'appel à f mais au niveau du

8. Cette approche de désassemblage, dite récursive, est présentée en section 1.3.2.1.a

branchement original. Les données se trouvant après cet appel seront alors interprétées comme du code machine. Ensuite, un octet de « code mort » est inséré juste après l'instruction `call` afin d'optimiser la désynchronisation du désassembleur.

1.2.5.5 Approche de Wroblewski

Dans [185, 186] Wroblewski propose un modèle ainsi qu'une approche d'obscurcissement de binaire. Le modèle proposé lui permet de prouver l'existence d'un algorithme purement séquentiel d'obscurcissement de code. Son approche quant à elle repose sur deux techniques précédemment décrites : le ré-ordonnancement et l'insertion de blocs de codes. Une comparaison empirique est aussi exposée par rapport aux approches de Collberg *et al.* [44, 46] et celle de Wang *et al.* [175, 176] pour montrer la flexibilité et la portabilité de l'approche.

1.2.6 Techniques d'obscurcissement de code employées par les codes malveillants

La plupart des codes malveillants collectés à l'heure actuelle se présentent sous la forme de binaires comme en témoignent les souches téléchargeables sur le site Offensive Computing [9]. De même, le site VX Heavens [171] offre une collection de quelques 271 092 programmes à caractère malveillant dont 95% sont des binaires et seulement 5% des codes en langages interprétés.

Nous présentons ici les techniques employées par les codes malveillants binaires afin d'obscurcir leur propre code. Il s'agit là d'une présentation non exhaustive visant à illustrer la différence entre les techniques d'obscurcissement de code utilisées par les codes malveillants et celles de protection logicielle présentées précédemment en sections 1.2.4 et 1.2.5. Pour plus de détails, un descriptif complet est présenté dans [158, chapitre 7] dont nous exposons ici les grandes lignes.

1.2.6.1 L'insertion de code mort

Cette technique consiste à rajouter du code inutile par rapport à la fonctionnalité initiale du programme. Les processeurs architecture à jeu d'instructions complexe (« *Complexe Instruction Set Computer* » ou CISC) offrent plusieurs combinaisons d'instructions pour aboutir au même résultat. Des exemples d'instructions assembleur inutiles sont présentées en figure 1.2. Le premier consiste à rajouter 0 à un registre. Le second affecte la valeur contenue dans un registre à ce même registre. Le troisième affecte à un registre le résultat d'un `OU` logique de sa valeur avec la valeur 0. Le dernier affecte à un registre le résultat d'un `ET` logique de sa valeur avec un masque dont tous les bits sont à 1.

9. Ce site est accessible à l'URL suivante : `http://www.offensivecomputing.net/` (dernier accès en décembre 2010).

Exemple	Signification
`add Reg,0`	`Reg ← Reg + 0`
`mov Reg,Reg`	`Reg ← Reg`
`or Reg,0`	`Reg ← Reg | 0`
`and Reg,-1`	`Reg ← Reg & -1`

TABLE 1.2 – Exemples de « code mort » en pseudo-assembleur x86. À gauche le code assembleur. À droite la signification correspondante.

1.2.6.2 La substitution de variables

Au niveau assembleur cette approche consiste à permuter les registres employés. Un exemple est exposé en figure 1.3 dans lequel deux programmes apparaissent comme identiques à une permutation des registres près. En effet, les registres `eax`, `ebx`, `edx` et `edi` du deuxième programme se substituent respectivement aux registres `edx`, `edi`, `esi` et `eax` du premier programme.

Programme 1	Programme 2
`pop` **`edx`**	`pop` **`eax`**
`mov` **`edi`**`,04h`	`mov` **`ebx`**`,04h`
`mov` **`esi`**`,ebp`	`mov` **`edx`**`,ebp`
`mov` **`eax`**`,0Ch`	`mov` **`edi`**`,0Ch`
`add` **`edx`**`,088h`	`add` **`eax`**`,088h`

TABLE 1.3 – Exemple d'échange de registres pour deux instances du virus W95.REGSWAP.

1.2.6.3 La permutation des instructions

Cette technique consiste à modifier l'ordre des instructions indépendantes. Le tableau 1.4 illustre cette technique à travers la dernière instruction copiant `ecx` octets contenus à l'adresse pointée par le registre `esi` vers l'adresse pointée par `edi`. Les trois affectations de ces registres sont interchangeables. Les deux programmes présentés sont donc équivalents.

Programme 1	Programme 2
`mov ecx,104h`	`mov edi,dword ptr [ebp+08h]`
`mov esi,dword ptr [ebp+0Ch]`	`mov ecx,104h`
`mov edi,dword ptr [ebp+08h]`	`mov esi,dword ptr [ebp+0Ch]`
`repnz movsb`	`repnz movsb`

TABLE 1.4 – Exemple de permutations d'instructions dans deux programmes équivalents.

1.2.6.4 La substitution d'instructions

Cela consiste à utiliser des règles de réécriture de code permettant de conserver la même sémantique. Le tableau 1.5 illustre ce type de transformations à

travers trois exemples. Le premier est une équivalence directe entre deux instructions, à savoir le OU exclusif (XOR) entre deux registres et l'affectation de la valeur 0 à ce registre. Le deuxième exemple montre que l'affectation d'une valeur immédiate à un registre peut aussi se décliner en utilisant la pile comme espace de stockage intermédiaire : la valeur immédiate est empilée avant d'être dépilée dans le registre considéré. Le dernier exemple retranscrit une opération logique (*OP*) entre deux registres via l'utilisation d'une adresse mémoire temporaire mem.

Instruction simple	Instructions multiples
`xor Reg,Reg`	`mov Reg,0`
`mov Reg,Imm`	`push Imm` `pop Reg`

OP `Reg,Reg2`	`mov Mem,Reg`
	OP `Mem,Reg2`
	`mov Reg,Mem`

TABLE 1.5 – Exemples de substitutions d'instructions employées dans le virus WIN32.METAPHOR [54]. À gauche, une instruction assembleur. À droite une séquence d'instructions équivalentes.

1.2.6.5 L'insertion de branchements inconditionnels ou conditionnels

Cela consiste à permuter le code de manière aléatoire tout en assurant la même exécution au moyen d'instructions de transfert. Ces instructions peuvent être inconditionnelles (instruction `jmp`), conditionnelles après une instruction de comparaison ou encore pseudo-conditionnelles. Le tableau 1.6 présente justement un code illustrant l'utilisation de branchements pseudo-conditionnels. En effet, quel que soit le chemin choisi dans le programme 1, le code exécuté sera toujours le même, c'est-à-dire celui donné dans la colonne de droite (programme 2).

Programme 1	Programme 2

`je label1`
`mov eax, 435098`
`sub eax, 340934`
`jmp label2`
`label1:`
`mov eax, 435098`
`sub eax, 340934`
`label2:`
`...`
	`mov eax, 435098`
	`sub eax, 340934`
	`...`

TABLE 1.6 – Exemple de branchements pseudo-conditionnels.

L'ensemble des transformations précédentes est récapitulé sur le tableau 1.7 qui présente des virus métamorphes utilisant ces transformations.

	Evol	Zmist	Zperm	Regswap	Gastropod	MetaPHOR
substitution d'instructions	○	○	○	○	○	●
permutation d'instructions	●	●	○	○	○	●
substitution de variables	●	●	○	●	●	●
insertion de « code mort »	●	●	○	○	○	●
insertion de branchements	○	●	●	○	○	●

Table 1.7 – Techniques d'obscurcissement de code employées dans des codes malveillants métamorphes connus extraits de [158, chapitre 7] (● représente la présence d'une transformation et ○ son absence).

1.2.7 Fonctionnement et illustration du processus d'auto-reproduction des codes métamorphes connus

Jusqu'à présent nous avons étudié l'étape de mutation de code intervenant dans le processus d'auto-reproduction du métamorphisme. Or, Lakhotia *et al.* [106] ont remarqué, à juste titre, qu'un code malveillant métamorphe doit pouvoir inverser ses propres transformations afin de se répliquer. Dans le cas contraire, l'emploi systématique de techniques d'obscurcissement de code, complexifiant sans cesse la structure du programme, conduirait à une augmentation progressive et incontrôlée de la taille du code. De fait, le cycle de reproduction métamorphe procède au minimum en deux temps, comme illustré sur la figure 1.6 :

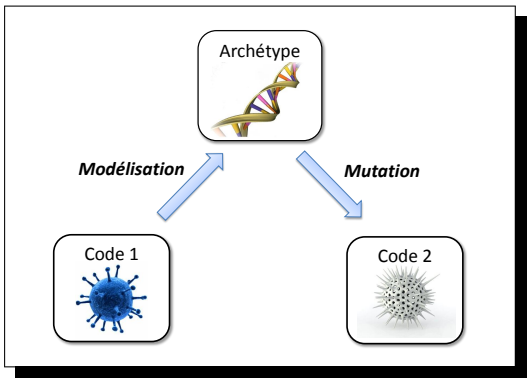


Figure 1.6 – Schéma simplifié du processus d'auto-reproduction d'un code métamorphe.

1. dans un premier temps, le programme se « modélise » afin d'obtenir une représentation abstraite de son propre code ainsi que de sa structure. Par la suite, nous utiliserons le terme *archétype* pour désigner cette représen-

tation bien que d'autres appellations telles que *forme normale* ou encore *forme canonique* sont parfois employées. Cette forme abstraite permet au code métamorphe une manipulation plus facile des éléments qui le composent (instructions et structures de données) ;

2. dans un deuxième temps, les techniques de mutations présentées jusque là s'appliquent afin de produire une nouvelle mutation de son propre code.

Dans la section 1.2.7.1 nous détaillons le fonctionnement d'un code métamorphe tel que présenté d'un point de vue général en figure 1.6. Puis, en section 1.2.7.2 nous illustrons le processus de réplication d'un virus métamorphe représentatif.

1.2.7.1 Processus de réplication d'un code métamorphe

Les travaux de Walenstein *et al.* [173] décomposent le fonctionnement d'un virus métamorphe selon cinq étapes que nous présentons succinctement :

1. la **localisation de son propre code**. Un code métamorphe doit impérativement localiser son propre code lors de chaque mutation. Dans le cas d'un virus, il s'agit de pouvoir déterminer son propre code parmi celui du programme infecté. Cette problématique de localisation est beaucoup plus simple en cas de non infection tel que celui d'un vers ou encore d'un cheval de Troie. En effet, dans ce cas, le programme entier correspond au code à localiser ;
2. le **décodage**. Après la localisation, un code métamorphe doit pouvoir décoder son propre code, c'est-à-dire passer de sa forme exécutable à une représentation intermédiaire lui permettant de simplifier la manipulation de son code ainsi que de sa structure. Pour des programmes binaires, cette étape correspond à celle du désassemblage. Différentes options se présentent pour cette étape : soit le programme se décode (désassemble) facilement, dans ce cas l'obscurcissement de code employé vise à compliquer l'étape suivante qui est l'analyse du code obtenu ; soit le désassemblage est déjà la cible des transformations utilisées. Dans ce deuxième cas, le programme métamorphe doit embarquer des informations supplémentaires lui permettant de se décoder plus facilement. À titre d'exemple, le code malveillant Miss Lexotan [10] capable de se désassembler lui même, insert l'instruction `xor ebp,imm` qui ne présente pas d'utilité directe pour la fonctionnalité du code. Toutefois, cette méta-instruction permet de spécifier par le biais du second opérande quels sont les registres utilisés qui ne peuvent être modifiés. Les autres registres peuvent être utilisés dans l'élaboration du « code mort » ;
3. l'**analyse**. Une fois le décodage terminé, l'étape d'analyse du code permet d'en extraire l'*archétype*. Là encore, l'obscurcissement de code employé peut grandement compliquer le travail, aussi bien pour un outil externe que pour le programme métamorphe lui même. Différentes options se présentent pour la phase d'analyse en fonction du choix fait lors du décodage :

10. Les fichiers sources sont téléchargeables à l'URL suivante : `http://vx.netlux.org/dl/mag/29a-6.zip` (dernier accès en décembre 2010).

si les mutations de code visent l'analyse, le code métamorphe doit embarquer des informations supplémentaires lui permettant de réaliser sa propre analyse. Dans le cas où l'obscurcissement de code ne cible que le décodage, l'analyse ne requiert pas d'information supplémentaire ;

4. la **transformation**. À partir de son *archétype*, le programme métamorphe emploie les techniques d'obscurcissement de code présentées précédemment pour modifier son code ;
5. l'**attachement**. Une fois l'*archétype* transformé, le code est alors attaché à un programme hôte. L'attachement correspond à la phase d'infection dans le modèle viral d'Adleman. Bien entendu cette étape est facultative si le code malveillant n'est pas de type infectieux. Dans tous les cas, le code exécutable est produit à partir du modèle transformé.

La conception d'un code métamorphe représente un compromis entre l'efficacité de l'obscurcissement de code et la nécessité d'auto-analyse. Si les techniques employées dans le cadre de la mutation s'avèrent trop complexes, alors le programme métamorphe se verra dans l'incapacité de se modéliser et *a fortiori* de se répliquer. Dans le cas contraire, si les transformations de codes s'avèrent trop facilement inversables, alors l'analyse du code par un outil externe et donc sa détection statique n'en sera que plus simple. Nous illustrons ci-après le choix fait par les développeurs dans le cas du virus métamorphe MetaPHOR.

1.2.7.2 Étude de cas : le virus MetaPHOR

Le meilleur exemple pour illustrer cette anatomie du métamorphisme est sans conteste le virus Win32.MetaPHOR [11] qui constitue à ce jour le code métamorphe le plus abouti [65]. Une description complète est exposée dans [12] et [65] dont nous reprenons ici les principaux aspects.

Écrit en 2002 par *The Mental Driller*, ce programme comporte 14 000 lignes de code assembleur dont 70% correspond au moteur de métamorphisme. Ce programme viral se compose de deux parties : la première qui constitue l'amorce du virus, la seconde représente le corps même du virus. L'amorce joue le rôle de chargeur du corps de virus en allouant et déchiffrant le cas échéant la deuxième partie du code (qui est chiffrée 15 fois sur 16) [12].

Le moteur de métamorphisme repose sur un ensemble de règles de réécriture (appelées aussi règles de substitution). En interne, ces instructions sont représentées par du pseudo-code assembleur afin de faciliter la transformation du code. Trois catégories de substitution sont utilisées en fonction du nombre d'instructions produites en sortie pour une instruction équivalente en entrée.

Le processus de réplication de ce virus comporte les cinq étapes présentées précédemment, qui illustrent les choix de conception adoptés par le développeur.

11. Le source assembleur est téléchargeable à l'URL suivante : `http://vx.netlux.org/src_view.php?file=metaphor1d.zip` (dernier accès en décembre 2010).

12. Certains travaux [189] considèrent ce virus comme semi-métamorphe, voire polymorphe, à cause de la présence d'un corps chiffré. Nous considérons ici que ce virus est effectivement métamorphe (d'ordre 0) puisque le chiffrement s'applique sur le corps du programme qui est déjà obscurci comme l'est le chargeur.

La figure 1.7 présente la réplication du virus pour une portion de l'amorce virale.

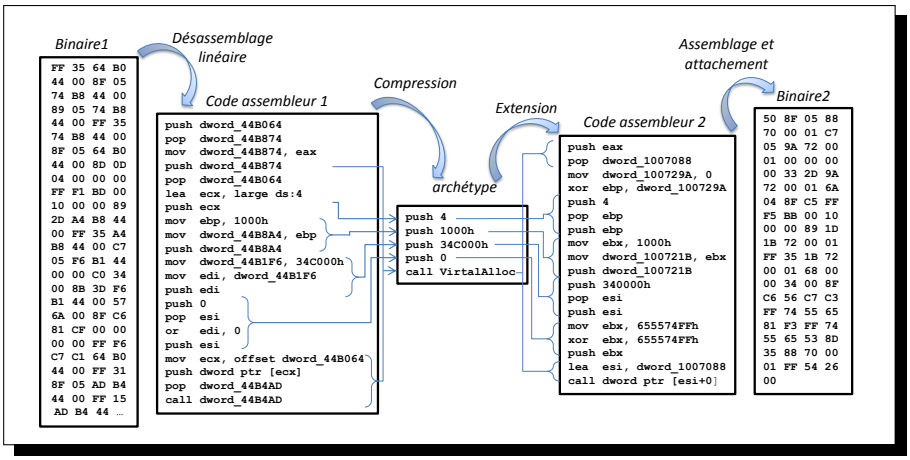


FIGURE 1.7 – Illustration du fonctionnement de l'amorce du virus METAPHOR.

1. le **désassemblage et dé-permutation**. Le désassemblage du virus consiste, à partir du point d'entrée du virus sous forme binaire, à retranscrire son propre code dans un pseudo-assembleur qui constitue une représentation abstraite plus facilement manipulable par la suite. Après cette étape de désassemblage, la dépermutation intervient dans le corps du virus pour reconstruire la version linéaire du code. Cette étape est illustrée à gauche de la figure 1.7 qui montre la représentation hexadécimale du premier binaire ainsi que sa retranscription en langage assembleur. Le virus désassemble son propre binaire pour en extraire le code assembleur correspondant ;
2. la **compression**. Partant du listing de son pseudo-code, le virus utilise ses règles de réécriture de manière itérative afin minimiser son code pour revenir à son *archétype*. Cette étape aboutit au code de la partie centrale de la figure 1.7. Ce code alloue dans l'espace d'adressage du processus un espace mémoire de 34C000h octets ($\simeq$ 3,3 Mo) accessible en lecture et en écriture ;
3. la **permutation**. Partant de sa forme normale, le virus permute les instructions constituant son corps. Cette étape n'est pas illustrée dans la figure 1.7 car l'amorce du virus n'est pas permutée ;
4. l'**extension**. Les règles de réécriture sont à utiliser dans le sens expansif et de manière aléatoire afin de produire un nouveau code obscurci équivalent. Cette étape est représentée par les accolades de droite sur la figure 1.7 ;
5. l'**assemblage**. Finalement, le pseudo-code est assemblé afin de produire le code binaire exécutable qui sera inséré à l'intérieur d'un programme hôte. La représentation hexadécimale correspondante est donnée en fin de figure 1.7.

Comme en témoigne la figure 1.7, les formes mutées de ce virus métamorphisme sont syntaxiquement différentes. Une approche de détection par découverte d'un motif binaire apparaît alors inefficace confrontée à de tels programmes.

1.2.8 Bilan des techniques de mutation par obscurcissement de code

Dans cette section, nous avons abordé les techniques de transformation de code utilisées par les programmes évolutifs sous l'angle de l'obscurcissement de code. Nous avons vu que l'obscurcissement de code au sens de Barak *et al.* est impossible, en cela qu'il existe des fonctions dont une description algorithmique peut entièrement être obtenue par la simple observation des couples enrée/sortie. Même dans un cadre moins restrictif que celui de la « boîte noire virtuelle », Godwasser *et al* montrent que ce résultat demeure inchangé. Si une protection absolue n'est pas atteignable dans tous les cas, Lynn *et al.* prouvent que les fonctions points, permettant notamment l'authentification par mot de passe, démontre l'existence de fonctions obscurcissables. De même, l'utilisation du τ-obscurcissement de code apporte aussi un résultat positif dans le domaine en garantissant la propriété de « boîte noire virtuelle » pendant une certaine durée.

Après ces résultats qui nous ont permis de faire le point sur les possibilités théoriques de l'obscurcissement de code, nous avons ensuite présenté les principaux critères d'évaluation des transformations utilisées : la puissance, la résilience, ainsi que le coût. Suite à ces mesures d'efficacité, les principales techniques et approches d'obscurcissement de code employées dans le cadre de la protection des logiciels ont été exposées. Le bilan de ces techniques est positif puisque certaines approches permettent de prouver la résilience des transformations employées dans un contexte d'analyse statique.

En ce qui concerne la création de codes malveillants métamorphes, les techniques de mutation de code utilisées actuellement sont plus simples. Cette simplicité a été justifiée dans plusieurs travaux [21, 106] par la nécessité, pour un code métamorphe, de pouvoir inverser ses propres transformations afin de se reproduire. Après avoir exposé ces transformations, nous avons présenté le cycle d'auto-reproduction des codes malveillants métamorphes connus. Ce processus a été illustré sur un cas d'étude réel utilisant des règles de réécriture simples.

La problématique identifiée dans cette section provient du décalage constaté entre d'une part, les techniques de protection logicielle, pour lesquelles des preuves de résilience peuvent être avancées et d'autre part, les transformations empiriques utilisées dans le cadre du métamorphisme. En résumé, est-il possible d'utiliser des techniques d'obscurcissement de code à forte résilience dans un programme métamorphe ? Cette question est à l'origine de la partie I de ce mémoire.

1.3 Techniques de détection adaptées aux codes métamorphes

Nous présentons dans cette section la détection des codes malveillants métamorphes suivant deux approches :

- d'une part, les approches de **détection statique** issues des techniques employées dans le cadre de la compilation de programmes. Ces techniques travaillent directement sur l'image d'un binaire pour en autoriser une détection avant exécution ;
- d'autre part, les approches de **détection dynamique** qui se focalisent sur les interactions observables entre un programme en cours d'exécution et son environnement, c'est-à-dire le système d'exploitation hôte. Ces techniques sont utilisées pour s'affranchir des transformations syntaxiques employées par les codes évolutifs.

Avant de décliner ces différentes approches, nous commençons par présenter le problème de la détection des codes malveillants. Cette présentation permet d'introduire des propriétés classiques utilisées par la suite pour caractériser un détecteur.

1.3.1 Définition et propriétés d'un détecteur de codes malveillants

Nous introduisons ici les concepts et notions de base concernant le problème de la détection de codes malveillants. Un *détecteur* est un modèle de classification, que l'on nomme *classifieur*, permettant de prédire l'appartenance des objets observés (ici des programmes) à l'une des deux classes suivantes : la classe des codes malveillants et la classe des programmes « légitimes ». Plus formellement, étant donné un ensemble de programmes $\mathcal{P}$ et un ensemble de codes malveillants $\mathcal{M} \subset \mathcal{P}$, un détecteur est alors représenté par une fonction booléenne de détection D définie sur $\mathcal{P}$ par :

$$\forall p \in \mathcal{P}, \begin{cases} D(p) = 1 \text{ si } p \in \mathcal{M} \\ D(p) = 0 \text{ sinon.} \end{cases}$$

Le résultat de détection du programme p par le détecteur D est dit positif si $D(p) = 1$. Dans le cas contraire, le résultat de détection est dit négatif.

Comme nous l'avons vu en section 1.1, la détection des codes malveillants est un problème difficile. En fonction du modèle considéré ce problème complexe peut même s'avérer impossible. De fait, les résultats d'un *détecteur* de codes malveillants sont nécessairement approximatifs et peuvent donc être erronés. Ces résultats se représentent généralement sous la forme d'une matrice de confusion [57], appelée aussi tableau de contingence, dont une illustration est donnée en figure 1.8.

Une matrice de confusion s'interprète de la façon suivante : si le détecteur considéré fournit un résultat positif, et que le résultat attendu est effectivement

positif, on parle alors de vrai positif ; dans le cas où le résultat attendu est négatif, on parle alors de faux positif. Si maintenant le détecteur considéré présente un résultat négatif alors que le résultat attendu est positif, on parle de faux négatif ; dans le cas contraire où le résultat attendu est aussi négatif, on parle alors de vrai négatif.

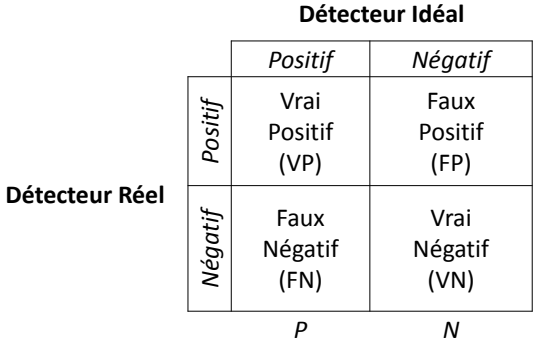


FIGURE 1.8 – Matrice de confusion illustrant les quatre types de résultats possibles en détection.

Afin de mesurer l'efficacité d'un détecteur, plusieurs caractéristiques ont été proposées dont nous exposons les principales [57].

Définition 20. *(Fiabilité, sensibilité ou complétude). La fiabilité (sensibilité ou encore complétude) d'un détecteur de codes malveillants désigne sa capacité à émettre une alerte en cas de présence de code malveillant. Elle est égale au taux de vrais positifs :*

$$\textit{fiabilité} = \frac{VP}{VP + FN} = \frac{VP}{P}.$$

Un *détecteur* est dit *fiable* (*sensible* ou encore *complet*) s'il émet peu de (idéalement aucun) faux négatifs.

Définition 21. *(Pertinence, spécificité ou correction). La pertinence (spécificité ou encore correction) d'un détecteur de codes malveillants désigne sa capacité à ne pas émettre d'alerte en cas de présence de code légitime, c'est-à-dire autre que malveillant. Elle est égale à :*

$$\textit{pertinence} = \frac{VN}{VN + FP} = \frac{VN}{N}.$$

Un *détecteur* est dit *pertinent* (*spécifique* ou encore *correct*) s'il émet peu de (idéalement aucun) faux positifs.

Plusieurs autres mesures sont aussi définies et utilisées à partir d'une matrice de confusion, par exemple la *précision*, qui est définie par $\frac{VP}{VP+FP}$, ainsi que l'*exactitude*, en anglais « *accuray* », qui est égale à $\frac{VP+VN}{VP+FP+FN+VN} = \frac{VP+VN}{P+N}$.

1.3.2 Détection statique

La détection statique de codes malveillants métamorphes s'appuie sur des techniques d'analyse statique de code développées à l'origine dans le cadre de la compilation de programmes. Le processus de compilation consiste à produire un code exécutable par une machine à partir des sources d'un programme dans un langage de plus haut niveau. Il peut aussi bien s'agir d'un code binaire nativement exécutable sur un ordinateur que d'un « *bytecode* » interprété par une machine virtuelle. Les différentes étapes du processus de compilation sont illustrées sur la gauche de la figure 1.9 :

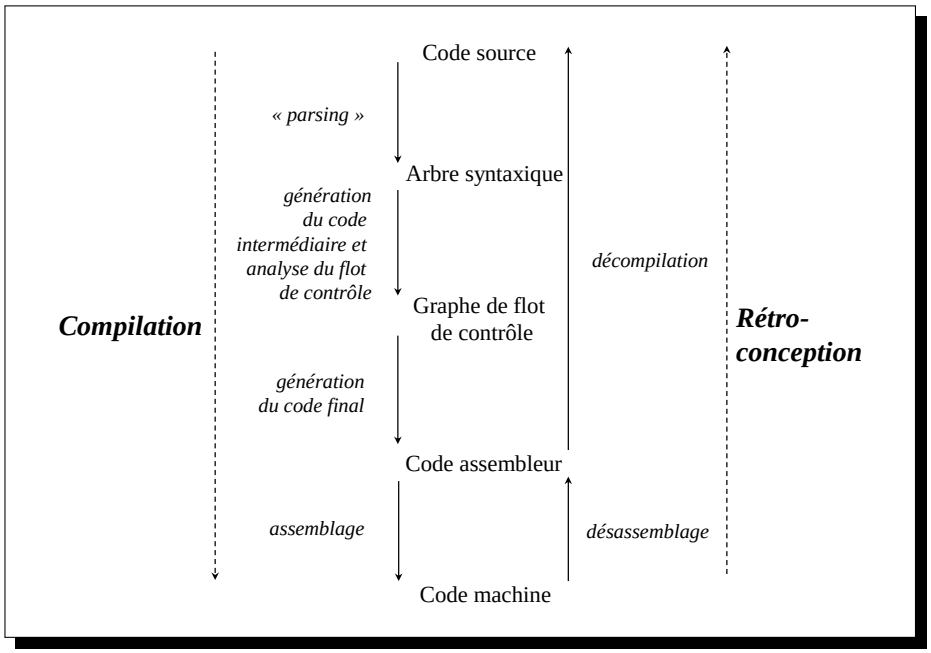


FIGURE 1.9 – Schéma du lien existant entre une chaîne de compilation et le processus de rétro-conception.

1. les sources du programme à compiler sont parsées afin de produire des arbres syntaxiques abstraits (« *Abstract Syntax Trees* » ou ASTs) [2] ;
2. le code intermédiaire est ensuite généré à partir des ASTs obtenus, pour ensuite être regroupé sous la forme d'un graphe de flot de contrôle (« *Control Flot Graph* » ou CFG) ;
3. le code assembleur est alors généré à partir du CFG contenant le code intermédiaire. Cette étape est facultative dans une chaîne de compilation classique mais est utile pour notre propos puisque toute détection statique de codes malveillants évolutifs s'appuie *a minima* sur les instructions assembleurs d'un programme ;

4. le code assembleur produit est finalement assemblé pour produire le code machine souhaité.

Le processus inverse de la compilation, appelé rétro-conception, est illustré à droite de la figure 1.9 et détaillé en section 1.3.2.1. Il comporte deux étapes qui reprennent celles de la compilation mais en sens inverse :

1. la première étape est le désassemblage présentée en section 1.3.2.1.a. Elle consiste à produire un source assembleur à partir d'un programme donné sous forme binaire ;
2. la deuxième étape, appelée décompilation [37, 38], est illustrée à droite de la figure 1.9. Cette étape consiste à remonter à un code source possible de ce programme dans un langage dit de haut niveau (C, C++, Pascal, etc.), à partir du source assembleur obtenu par désassemblage. La décompilation nécessite de construire un CFG du programme considéré, pour ensuite optimiser à la fois ce CFG et le flot de données.

L'obscurcissement de code peut s'interpréter comme une compilation volontairement non optimisée. Dans ce cas, détecter un code métamorphe nécessite alors d'inverser cette chaîne de compilation, c'est-à-dire de le rétro-concevoir [34].

1.3.2.1 Le processus de rétro-conception statique

Nous nous plaçons dans le cas général de la rétro-conception de programmes malveillants sous forme binaire. Pour les autres codes malveillants se présentant sous la forme de code interprété le processus est simplifié par la présence d'informations complémentaires telles que les noms des symboles, la séparation entre le code et la donnée, le typage des données, etc.

1.3.2.1.a Le désassemblage

À partir d'un programme métamorphe sous forme binaire, toutes les approches statiques de détection proposées jusqu'alors nécessitent au minimum de recourir aux instructions du programme. La traduction d'une suite binaire en une instruction assembleur, c'est-à-dire du langage machine en une instruction compréhensible par un être humain constitue le désassemblage élémentaire. Dans notre cas, l'objectif consiste à extraire d'un programme P, donné sous forme binaire, l'ensemble des instructions de P, c'est-à-dire produire le désassemblage complet de P instruction par instruction.

Par désassemblage élémentaire nous ne désignons pas la simple traduction d'une suite binaire en instructions assembleur, conformément à une spécification d'un CPU, mais le problème suivant : étant donné un programme P sous forme binaire et un offset x dans P, est-ce que la donnée située en position x correspond à une instruction dans le source du programme P ? Un désassembleur peut ainsi être envisagé en tant que détecteur de code. Le désassemblage complet d'un programme consiste alors à itérer le désassemblage élémentaire sur l'intégralité du binaire d'un programme. Conformément à la figure 1.8, un désassembleur

est alors *fiable* s'il est capable d'identifier tout le code initialement contenu dans le programme. De manière équivalente, il est dit *pertinent* si aucune donnée effective du programme initial n'est typée en tant que code. Malheureusement, le problème du désassemblage élémentaire est indécidable. En effet, le problème de l'arrêt [161] se réduit directement à celui du désassemblage [13].

D'un point de vue pratique, le désassemblage s'avère particulièrement difficile pour des architectures de type CISC où la densité d'instructions est telle que presque toute donnée peut s'interpréter comme une instruction assembleur. Différentes approches classiques existent toutefois pour approximer le désassemblage d'un programme :

- le **désassemblage linéaire**. Il s'agit de l'algorithme le plus simple. Partant du point d'entrée du programme, le désassemblage se fait de manière séquentielle, instruction par instruction, indépendamment du flot de contrôle. L'avantage de cette approche est d'être extrêmement simple. L'inconvénient réside dans la perte du flot de contrôle : le désassemblage se poursuit séquentiellement même après un saut, ce qui peut conduire à typer de la donnée comme du code. Cette approche n'est donc pas *pertinent*e.
- le **désassemblage récursif**. L'algorithme employé correspond à un désassemblage séquentiel qui cette fois-ci est rappelé de manière récursive en cas de branchement conditionnel et d'appel à une procédure. L'algorithme 1.3 représente un désassemblage récursif. Le principe consiste à désassembler le programme linéairement jusqu'à arriver à une instruction de transfert de contrôle. Dans ce cas, l'algorithme est appelé récursivement sur chacune des adresses de destination.

```
1  global startAddr, endAddr;
2  proc DisasmRec(addr)
3  begin
4    while (startAddr < addr < endAddr) do
5      if (addr has been visited already) return;
6      I := decode instruction at address addr;
7      mark addr as visited;
8      if (I is a branch or function call)
9        for each possible target t of I do
10         DisasmRec(t);
11       od
12       return;
13     else addr += length I ;
14   od
15 end
```

Listing 1.3 – Algorithme de désassemblage récursif d'un programme [115].

13. Il suffit pour cela de considérer un appel à un programme externe. Dans ce cas, la donnée binaire située après cet appel correspondra effectivement à du code utile si le programme externe se termine. Dans le cas contraire, il s'agit alors de données effectives puisque jamais exécutée.

Cette approche n'est ni *fiable* ni *pertinent*e bien que dans la pratique, les résultats obtenus soient meilleurs que ceux issus d'un désassemblage linéaire. L'approche n'est pas *fiable* car elle peut omettre certaines portions de codes par méconnaissance du contexte d'exécution au niveau d'une instruction. Par exemple, pour une instruction du type `jmp eax` la valeur du registre `eax` pour cette instruction est nécessaire afin de déterminer la destination du saut et ainsi de poursuive correctement le désassemblage. L'approche n'est pas *pertinent*e non plus. Par exemple, en cas d'appel à une procédure f via l'instruction `call`, l'adresse de retour contenue dans la pile peut être modifiée par la fonction f afin de ne pas exécuter le code situé juste après l'instruction `call`. Ce cas de figure n'est pas pris en compte par l'algorithme présenté qui suppose que l'adresse de retour l'instruction `call` demeure constante.

Des approches hybrides ont été envisagées pour le désassemblage [151] afin de combiner les avantages de ces deux approches. Cependant, le désassemblage de binaire de type x86, architecture qui représente la cible privilégiée des codes malveillants évolutifs, demeure difficile et apparaît encore comme une limitation reconnue de la plupart des approches de détection statique présentées par la suite.

1.3.2.1.b Construction du graphe de flot de contrôle (CFG)

Une fois la liste des instructions assembleur obtenues, l'étape suivante consiste à en extraire les structures de contrôle du programme initial, à savoir les boucles, les fonctions et autres branchements. À ce titre, les instructions d'un programme sont représentées sous forme d'un graphe appelé graphe de flot de contrôle (« *Control Flot Graph* » ou CFG).

Définition 22. *(Graphe de flot de contrôle [2]). Un graphe de flot de contrôle (*« Control Flot Graph *» ou CFG) est un graphe orienté* (N, E) *où* N *désigne l'ensemble des sommets du graphe et* E *l'ensemble des arêtes. Chaque sommet représente un bloc de base, c'est-à-dire un ensemble ordonné d'instructions séquentielles se terminant :*

- *soit par une instruction de transfert ;*
- *soit par une instruction séquentielle immédiatement suivie d'une instruction séquentielle appartenant à un autre bloc de base.*

Chaque arrête e issue d'un bloc de base correspond à une sortie conditionnelle ou inconditionnelle de ce bloc.

Le CFG représente tous les chemins qui peuvent être empruntés au cours des exécutions d'un programme. Il sert ensuite de base pour l'analyse du flot de données. La figure 1.10 montre une implémentation itérative correcte, en langage C, de la fonction factorielle pour une architecture 32 bits, ainsi que son CFG associé.

La construction d'un CFG à partir d'un source assembleur ne présente pas de difficulté particulière. Cependant, l'exactitude d'un tel graphe est conditionnée par la *fiabilité* et la *pertinence* de l'étape de désassemblage. En effet, si le

```
1  #define MAXINT 2147483647
2
3  int fact (int n) {
4      int r,i;
5      if ((n<1)||(n==MAXINT)){
6          r=0;
7      }else{
8          r=1;
9          for(i=2;i<=n;i++){
10             if (r<=MAXINT/i){
11                 r=r*i;
12             }else{
13                 r=0;
14             }
15         }
16     }
17     return r;
18 }
```

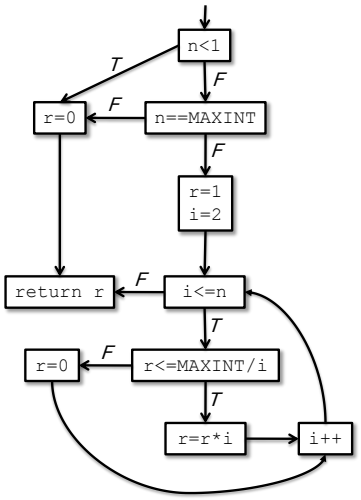


FIGURE 1.10 – Exemple de programme avec son CFG associé.

désassembleur n'est pas *fiable*, le CFG est alors incomplet puisqu'une partie du code n'a pas été correctement identifiée. Si le désassembleur n'est pas *pertinent*, le CFG contient des blocs de base invalides puisque des données ont été interprétées comme étant du code.

1.3.2.1.c Optimisation du flot de données et du CFG

L'analyse du flot de données est une discipline bien documentée dans le cadre de la compilation de programmes [2, 83]. Une présentation complète et formelle de l'analyse du flot de données dans le cadre de la rétro-conception de programme est exposée dans les travaux de Cifuentes *et al.* [37, 38]. Nous présentons ici l'analyse et l'optimisation du flot de données d'un point de vue pratique telles qu'elles sont appliquées dans les approches de détection des codes malveillants métamorphes. De manière informelle, l'optimisation du flot de données consiste à propager les valeurs des données afin de simplifier le code contenu dans les blocs de base pour, au final, optimiser le CFG d'un programme. Les différentes étapes permettant d'aboutir à ce résultat sont :

- la **propagation des données**, qui consiste à propager la valeur d'une donnée initialisée. Au sein d'un même bloc de base, lorsqu'une instruction définie une variable locale et que cette variable est ensuite utilisée par d'autres instructions qui ne la redéfinissent pas, toutes les occurrences de cette même variable peuvent alors être remplacées par sa valeur. De manière similaire la propagation des données se fait aussi entre les blocs de base ;
- la **suppression du « code mort »** , dont l'objectif est de supprimer les instructions dont le résultat n'est jamais utilisé. Par exemple, une variable est inutile si elle est définie deux fois dans un même bloc de base sans ja-

mais être utilisée entre ces deux définitions. De telles instructions peuvent être supprimées sans modifier le résultat du programme ;
- les **simplifications algébriques**. La plupart des instructions au niveau du langage machine se ramène à des calculs algébriques sur des variables afin de construire les paramètres nécessaires à des appels à une API. Les calculs algébriques classiques permettent d'en simplifier les expressions ;
- les **optimisations (compressions) du CFG**. L'ensemble des étapes d'analyse de flot de données précédentes permet dans certain cas de déterminer les destinations de blocs de base qui demeuraient jusque là inconnues, par exemple en cas d'indirections. Ces résultats permettent alors de compléter le CFG. De plus, certaines tautologies peuvent être découvertes et donc des blocs de base peuvent alors être fusionnés. Le but de cette dernière étape est de faire apparaître les structures de contrôle de haut niveau telles qu'elles apparaissaient dans le programme d'origine avant la production du binaire (boucle `for`, `while`, `switch`, etc.)

Comme nous venons de le voir, le processus de rétro-conception statique comprend au minimum trois étapes. En fonction du niveau d'abstraction requis par une approche de détection, certaines étapes ne sont alors pas nécessaires comme l'analyse du flot de donnée, ou encore la construction du CFG. Dans tous les cas, une approche statique nécessite au minimum les instructions assembleurs d'un programme binaire. Or, cette première étape de désassemblage est déjà impossible dans le cas général. Même son approximation pour des processeurs de type CISC demeure difficile et plus particulièrement dans le cas de l'analyse de codes malveillants qui comprend des programmes obscurcis et auto-modifiants. De plus, nous avons vu en section 1.2.5 que l'inter-dépendance entre le flot de contrôle et le flot de données peuvent grandement complexifier ce processus de rétro-conception pour les étapes suivant le désassemblage. Nous présentons maintenant, par ordre de généricité croissante, les différentes approches de détection statique qui s'inscrivent dans le processus de rétro-conception.

1.3.2.2 Approches par modèles de Markov cachés

Les modèles de Markov cachés (« *Hidden Markov Model* » ou HMMs) [142] font partie de la famille des modèles d'apprentissage, au même titre que les réseaux de neurones ou encore diverses méthodes dites de « *data mining* ». Ils constituent un modèle particulièrement adapté à l'analyse de motifs statistiques. De manière simplifiée, un HMM est un automate à états dont les transitions entre les états présentent une probabilité fixe. À chaque état de l'automate est associé une distribution de probabilité correspondant à un ensemble de symboles observés. Il est possible d'« entraîner » un HMM à représenter un ensemble de données. Ces données correspondent à une séquence d'observations. Les états de l'automate représentent les caractéristiques des données d'entrée. Les transitions ainsi que les probabilités d'observation correspondent aux propriétés statistiques de ces caractéristiques. Après apprentissage, il est possible d'utiliser un HMM

afin d'évaluer la similarité entre la séquence apprise et celle fournie en entrée, à partir d'une séquence d'observations

La détection statique par HMMs de codes métamorphes trouve ses origines dans les propriétés statistiques de tels binaires par rapport aux autres. En effet, les binaires obtenus par compilations classiques présentent des propriétés statistiques remarquables en ce qui concerne l'enchaînement des instructions qui les composent. Il en est de même pour les séquences d'instructions utilisées dans le cadre du métamorphisme qui ne se retrouvent pas dans des programmes compilés de manière « classique ». Ainsi, les travaux de Wong *et al.* [184] proposent d'identifier des familles de codes malveillants métamorphes aux moyens de HMMs. Le principe repose sur des similarités statistiques sur l'enchainement des instructions entre divers variantes d'une même famille. En termes de modélisation de virus, les états du HMM correspondent aux caractéristiques du code du virus alors que les observations représentent les instructions. Leurs résultats montrent que différents kits de constructions viraux sont discernables par leur approche.

Cette approche présente plusieurs limitations. Tout d'abord, elle n'est pas *fiable*, car il est possible de simuler une séquence de code légitime pour ne plus être discernable [114]. Ensuite, une transformation d'obscurcissement de code utilisée à la fois par un code malveillant et un programme légitime peut introduire des séquences de codes caractéristiques, c'est-à-dire « apprise » par le HMM. Un tel enchaînement peut alors provoquer de nombreux faux positifs sur des programmes légitimes. Cette approche n'est donc pas non plus *pertinent*e.

1.3.2.3 Approches par *model-checking*

Le « *model-checking* » désigne un famille de méthodes de vérification d'un modèle par rapport à une spécification donnée [42]. Le principe de la détection par « *model checking* » repose sur la spécification d'un comportement malveillant au moyen d'une formule de logique temporelle puis sa vérification sur un programme soumis à analyse.

En 2003, Singh *et al.* [154] décrivent un système de détection vérifiant des propriétés du CFG d'un programme suspect par rapport à une formule de logique temporelle linaire (« *Linear Tomporal Logic* » ou LTL) décrivant le comportement malveillant d'un ver. Toutefois, la spécification comportementale en LTL proposée ne permet pas de prendre en compte la mutation de code.

Une amélioration est proposée par Kinder *et al.* [94] en introduisant une nouvelle logique temporelle dénommée « *Computation Tree Predicate Logic* » (CTPL). Cette logique est aussi expressive que « *Computation Tree Logic* » (CTL) mais permettant de prendre en compte le cas du renommage des registres. Un algorithme de « *model checking* » pour cette logique est présenté afin de vérifier la présence d'un motif malveillant. Plus précisément, si le CFG d'un programme est un modèle pour la formule de spécification d'un comportement malveillant, alors le programme contient du code à caractère malveillant. Plusieurs variantes des vers NetSky, MyDoom et Klez ont pu être détectées au moyen d'une seule formule CTPL.

Un exemple de réplication, extrait de [94], est donné en figure 1.11 avec sa formule CTPL correspondante. Brièvement, cette formule CTPL décrit le com-

```
                                 ∃L_m ∃L_c ∃v_File(
                                  ∃r_0 ∃r_1 ∃L_0 ∃L_1 ∃c_0
mov  edi,[ebp+arg0]                 EF(lea(r_0, v_File)∧EX E(¬∃t(mov(r_0, t)∨lea(r_0, t)) U#loc(
xor  ebx,ebx                              L_0)) ∧
push edi                            EF(mov(r_1, 0)∧EX E(¬∃t(mov(r_1, t)∨lea(r_1, t)))U#loc(L_1))
...                                       ∧
lea  eax,[ebp+ExFileName]           EF(push(c_0)∧EX E(¬∃t(push(t)∨pop(t)))
mov  [esp+65Ch+var65C],104           U(push(r_0)∧#loc(L_0)∧EX E(¬∃t(push(t)∨pop(t)))
push eax                               U(push(r_1)∧#loc(L_1)∧EX E(¬∃t(push(t)∨pop(t)))
push ebx                                 U(call(GetModuleFileNameA)∧#loc(L_m))))
call ds:GetModuleFileNameA               )
lea  eax,[ebp+NewFileName]        ∧(∃r_0 ∃L_0(
push ebx                           EF(lea(r_0, v_File)∧EX E(¬∃t(mov(r_0, t)∨lea(r_0, t)) U#loc(L_0
push eax                                ∧
lea  eax,[ebp+ExFileName]          EF(push(r_0)∧#loc(L_0)∧EX E(¬∃t(push(t)∨pop(t)))
push eax                            U(call(CopyFileA)∧#loc(L_c)))
call ds:CopyFileA                  ))
                                   ∧EF(#loc(L_m)∧EF#loc(L_c))
                                 )
```

FIGURE 1.11 – Exemple de code auto-reproducteur avec sa formule CTLP.

portement du ver, c'est-à-dire sa réplication. Cet exemple correspond à l'appel de la fonction `GetModuleFileNameA` pour récupérer son nom de fichier, puis utilise ce nom pour l'appel à `CopyFileA`. C'est cette dernière fonction qui duplique alors le binaire. Chacun de ces appels est décliné en sous-formules décrivant la mise en place des arguments pour les appels de fonctions.

Les descriptions comportementales en CTPL sont produites manuellement pour détecter une portion du code malveillant. De plus, cette approche souffre d'un manque de généricité puisqu'elle ne traite que le cas du renommage des variables (ici les registres utilisés).

1.3.2.4 Approches par normalisation de code

Normaliser un programme consiste à en simplifier le code afin d'en obtenir une représentation la plus simple possible. Idéalement, cette étape a pour but l'obtention de l'*archétype* du programme. Le principe de la normalisation repose sur des techniques d'optimisation de code visant à en réduire la taille. Un exemple de réduction est illustré sur la figure 1.12 représentant, à gauche, le code d'origine d'un virus et à droite, le même code une fois réduit. Seules les instructions grisées de la partie droite constituent l'*archétype* du programme. Les autres instructions affectent des variables globales temporaires qui peuvent être potentiellement utilisées par la suite, ce qui empêche leur simplification dans ce contexte. De manière simplifiée, ce code permet de récupérer l'adresse virtuelle à laquelle la bibliothèque dynamique `kernel32` a été chargée dans l'espace d'adressage du processus. Cette portion de code comporte essentiellement des calculs intermédiaires visant à reconstruire la chaîne de caractère constituée des trois variables globales consécutives `dword_1`, `dword_2` et `dword_3`.

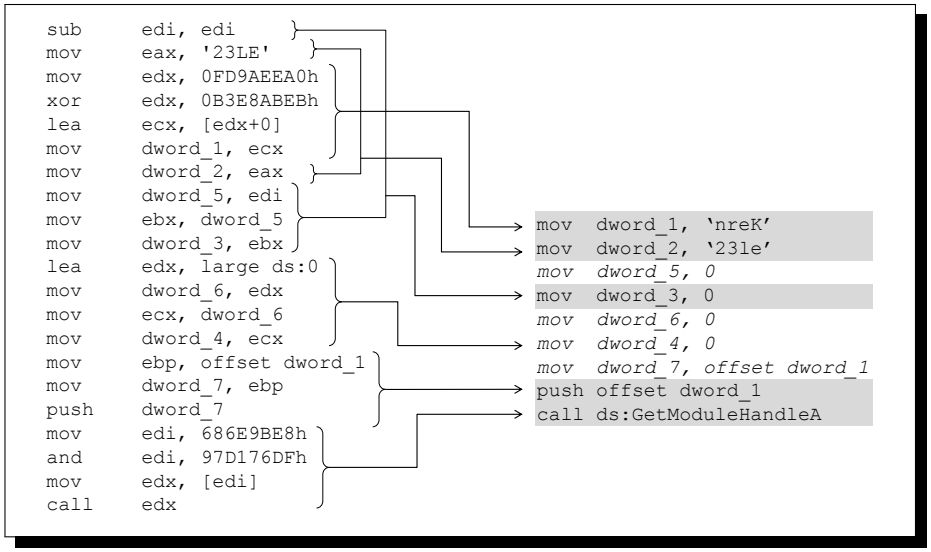


FIGURE 1.12 – Illustration d'une réduction de code sur le virus METAPHOR.

Plusieurs approches de détection de codes malveillants métamorphes à base de normalisation de code ont été proposées :

1.3.2.4.a Approche de Christorodescu *et al.*

Christodorescu *et al.* [36] proposent trois algorithmes spécifiques de normalisation visant à optimiser du code ayant subit les transformation suivantes : la permutation du code, l'insertion de « code mort » et le « *packing* » sous certaines hypothèses (code non auto-modifiant et exécution indépendante des entrées). Pour valider leur approche, les auteurs soumettent le ver BEAGLE.Y à plusieurs anti-virus, avant et après normalisation. Leurs résultats montrent que leur normalisation permet d'accroître les taux de détection des anti-virus testés pour toutes les transformations prises en compte.

1.3.2.4.b Approche de Walenstein *et al.*

Walenstein *et al.* [174] proposent une approche plus générique qui reprend le formalisme des grammaires formelles afin de modéliser les mutations de code employées. Ils utilisent pour cela des règles de réécriture dans le but de simplifier un code métamorphe. Les auteurs spécifient alors manuellement un ensemble de règles de réécriture utilisé par le virus W32.EVOL à partir de la liste des instructions qui le composent. Ces règles sont ensuite appliquées de manière itérative pour simplifier ce code viral. Les résultats obtenus montrent que les différentes variantes normalisées sont similaires à 98%. La principale lacune de cette approche réside dans la spécification manuelle des règles de réécriture employées propres à chaque code malveillant.

1.3.2.4.c Approche de Webster *et al.*

Les travaux de Webster *et al.* [179, 180] vont plus loin en proposant d'utiliser une spécification algébrique d'un langage assembleur afin de prouver l'équivalence ou la semi-équivalence de portions de code via l'utilisation d'outils d'assistance de preuve (« *theorem prover* »). Plus précisément, la syntaxe ainsi que la sémantique d'un sous-ensemble des langages assembleur IA-32 et Intel64 sont spécifiées au moyen d'un outil dédié OBJ [73]. Cet outil permet aussi d'interpréter la forme algébrique simplifiée afin de vérifier l'équivalence ou la semi-équivalence de deux portions de code. Des expériences sont menées sur des fragments de code issus de deux virus métamorphes Win95/Bistro et Win9x.Zmorph.A pour valider l'efficacité de la démarche. Cette approche est générique puisqu'elle s'attache à la sémantique du code indépendamment du type de transformation employé. Toutefois la *fiabilité* n'est pas prouvée, ni même illustrée sur des portions significatives d'un code malveillant, notamment en cas d'obscurcissement du flot de contrôle.

1.3.2.5 Approches par comparaison de graphes

Les approches par comparaisons de CFG partent de l'hypothèse que deux binaires provenant d'un même code métamorphe présentent des similarités dans le CFG de leurs *archétypes* respectifs. L'objectif consiste alors à extraire le CFG de l'*archetype* d'un code malveillant afin d'identifier ce code. D'un point de vue théorique, ce problème correspond à celui de l'isomorphisme de sous-graphe, problème qui est prouvé NP-complet [92]. Les approches suivantes tentent donc de trouver une solution approximative à ce problème pour la détection de codes malveillants.

1.3.2.5.a Approches de Christorodescu *et al.*

Christorodescu *et al.* [33] proposent une première architecture de détection composée de deux éléments. Le premier élément est un programme d'annotation qui, à partir du CFG d'un programme, produit un CFG dont les instructions sont annotées suivant différents types : instruction ou saut illégitime, appel indirect, assignation, boucle, etc. Le second élément est le détecteur, implémenté sous la forme d'un automate fini déterministe qui correspond à la description manuelle d'un code malveillant. Si le langage de l'automate présente une intersection non vide avec le langage correspondant à l'automate du programme analysé alors le patron malveillant est bien présent dans le code.

Dans [35], Christorodescu *et al.* proposent une description formelle de la sémantique d'un programme. Chaque comportement malveillant est décrit sous la forme d'un patron (« *template* ») qui représente une spécification abstraite des actions menées (affectation, opération algébrique, test, etc). L'algorithme de détection présenté fonctionne en déterminant pour chaque nœud d'un patron, un nœud correspondant dans le programme analysé. Leur approche formelle permet de montrer la *pertinence* de l'algorithme de détection. Les transformations qui sont prises en compte par cet algorithme sont : le ré-ordonnancement des

instructions, le renommage de registres ainsi que l'insertion de « code mort » . Pour ce qui est des substitutions d'instructions, leur prise en compte est limitée. Ces travaux ont été complétés par ceux de M.D. Preda *et al.* [140] qui fournissent une sémantique de traces afin de caractériser le comportement du programme en utilisant l'interprétation abstraite pour « masquer » les aspects inutiles de ce comportement. Dans ce cadre formel de l'interprétation abstraite, ils proposent aussi une définition des notions de *complétude* et de *correction* par rapport à la détection d'une classe d'obscurcissement de code. Ils montrent ensuite que le détecteur proposé par Christodorescu *et al.* [35] est aussi *complet* par rapport à certaines techniques d'obscurcissement de code communément employées par les codes malveillants (la permutation d'instructions, le renommage de registres, l'insertion de « code mort ») mais pas en ce qui concerne la substitution d'instructions.

1.3.2.5.b Approche de Bruschi *et al.*

Bruschi *et al.* [20, 21] proposent deux architectures de détection par comparaison de graphes composées à chaque fois de deux éléments. Le premier élément est une composante de normalisation de graphes qui reprend l'essentiel des étapes présentées en section 1.3.2.1. Le second élément, le comparateur de graphes, a pour finalité de mesurer la similarité entre le graphe du code sous analyse et le graphe de référence qui constitue la « signature » d'un code malveillant.

Dans [20] inspiré des travaux de Kruegel *et al.* [104], chaque nœud du CFG se voit attribuer une étiquette représentant le type de nœud : arithmétique entière ou réelle, opérations logiques, comparaison, appel de fonction, appel indirect de fonction, branchement, saut, saut indirect et retour de fonction. La comparaison entre graphes ainsi labellisés est réalisée au moyen de l'algorithme VF2 de la bibliothèque VFLib [67].

Dans [21], la similarité entre les graphes est inspirée d'autres travaux [101] dans lesquels une portion de code est caractérisée par un vecteur de mesures logicielles. Plus précisément, elle est égale la distance euclidienne entre sept mesures : le nombre de nœuds et d'arêtes dans le CFG, le nombre d'appels directs et indirects, le nombre de sauts directs et indirects ainsi que le nombre de branchements conditionnels.

1.3.2.5.c Approche de Zhang *et al.*

Zhang *et al.* [190] proposent une approche sensiblement équivalente à celle de Bruschi *et al.* Après l'étape de normalisation, chaque bloc de base du programme traité constitue un élément de patron de détection. Deux patrons sont alors comparés, un correspondant à un code malveillant, l'autre étant le programme soumis à détection. Une matrice de similarité est alors calculée, chaque élément mesurant la similarité entre les blocs de base des deux patrons. Après affectation entre blocs, réalisée au moyen de l'algorithme hongrois [105], la similarité finale entre patrons est calculée en tant que somme pondérée des valeurs d'affectation.

1.3.2.5.d Approche de Bonfante *et al.*

Bonfante *et al.* [17] partent du constat que pour un CFG, le nombre de successeurs pour un bloc de base donné est limité, ce qui a pour conséquence de diminuer la connexité du graphe à un ou deux successeurs excepté pour les indirections et les retours de fonctions. Ils ramenènent alors le problème d'isomorphisme de sous-graphe initial à un problème d'automates d'arbres. Pour chaque codes malveillants, le CFG est extrait, optimisé et réduit pour être ramené à une structure d'arbre. La base de données virales est alors constituée d'un ensemble fini d'arbres. Chaque candidat pour la détection, représenté lui aussi sous forme d'un arbre t, est alors soumis à un automate d'arbres ascendant [48] qui vérifie en temps linéaire par rapport à la taille de t si ce dernier (t) correspond à l'un de ceux contenus dans la base.

1.3.3 Détection dynamique

La détection dynamique constitue la seconde grande famille de techniques de détection de codes malveillants métamorphes. Elle consiste à exécuter un programme avec des entrées particulières afin de récupérer des informations permettant d'identifier un code malveillant à travers ses interactions avec son environnement. La principale motivation de l'analyse dynamique est de s'affranchir de toutes les difficultés inhérentes aux techniques d'analyse statique, à commencer par le désassemblage. Bien qu'elle soit aussi utile en détection statique, la notion de *comportement* constitue la notion incontournable des approches de détection dynamique. Nous en adoptons ici la définition de Jacob *et al.* [87] : « *le comportement d'un programme se traduit par ses interactions (automatiques ou conditionnées) avec les ressources matérielles, logicielles et humaines de son environnement d'exécution. Ces interactions doivent être observables depuis le référentiel choisi* ». Le processus de détection dynamique nécessite deux composants :

1. un **outil d'observation** en charge de collecter les informations décrivant le comportement du programme en cours d'exécution. Cet outil peut soit retransmettre directement ces informations au détecteur, soit les retranscrire sous forme de rapport une fois l'exécution terminée ;

2. un **algorithme de détection**, qui à partir des informations collectées par l'outil d'observation et un modèle de code malveillant fournit un résultat de détection.

La problématique de la détection dynamique est de définir un ensemble de *comportements* qui autorise à la fois une détection *fiable* et *pertinente*. La section 1.3.3.1 présente les différents approches utilisées pour l'observation d'un programme. Les sections 1.3.3.2 et 1.3.3.3 exposent les principales approches de détection dynamique comportementale.

1.3.3.1 Outils d'observation dynamique (*Monitoring*)

Une des principales caractéristiques de ces outils est leur niveau de transparence vis-à-vis du programme analysé. Un code malveillant analysé peut en effet tenter de détecter son environnement d'exécution et le cas échéant modifier son comportement afin de contourner sa détection. Le processus de collecte d'information se doit aussi d'être à la fois le plus précis et le plus complet possible puisque les résultats de détection en dépendent directement. Les différentes techniques de collecte employées actuellement sont les suivantes :

1.3.3.1.a L'instrumentation dynamique de binaires

Le fonctionnement de ces programmes consiste à modifier dynamiquement le binaire en cours d'exécution afin d'en prendre le contrôle. L'une des façons les plus simples de procéder consiste, par exemple, à détourner les API systèmes afin de collecter les interactions du programme avec son environnement d'exécution. Le principe généralement appliqué est celui de la *translation de binaire* [160] qui consiste à exécuter nativement un bloc de base et d'en récupérer le contrôle à la fin. De nombreux outils d'instrumentation de binaires existent parmi lesquels : Pin [118], Cobra [166], DynamoRIO [69], Valgrind [132], Diota [120], etc. Cette approche a le mérite d'être la plus simple en termes de déploiement par contre elle présente plusieurs inconvénients. Tout d'abord, le niveau d'intrusion dans le binaire est élevé, ce qui peut être gênant pour des codes vérifiant leur intégrité. De plus ces outils ne fonctionnent pas sur des modules noyaux et surtout gèrent difficilement les codes auto-modifiants. Par ailleurs, le problème de la compromission de l'environnement n'est pas pris en compte par ce genre d'outils. Il est alors nécessaire d'y adjoindre un dispositif de restauration du système pour retrouver une configuration intègre de l'environnement d'exécution.

1.3.3.1.b Les environnements « bacs à sable »

L'exécution est confinée dans un espace hermétique. Le processus lancé tourne alors avec des privilèges restreints et se voit offert un accès limité aux services du système d'exploitation. L'avantage de cette technique est d'autoriser un contrôle précis, éventuellement instruction par instruction du code analysé. Cependant, la restriction des services offerts rend ces environnement facilement détectables.

Le logiciel Norman Sandbox [134] est un exemple d'environnement « bac à sable » (« *sandbox* ») qui exécute un programme, soumis à analyse, dans un environnement contrôlé. Le principe repose sur l'interception des appels aux APIs pour en simuler les actions, sans recourir à leur exécution. Ainsi, aucune action malveillante n'est menée sur le système hôte, qu'il n'est donc pas nécessaire de restaurer. Toutefois, cette approche doit garantir la cohérence entre plusieurs appels aux APIs du système pour ne pas révéler sa présence [14].

14. Il suffit par exemple de considérer un programme qui écrit dans un fichier pour ensuite vérifier plus tard la présence de la donnée écrite. Si les simulations de l'écriture et de la lecture ne sont pas correctes alors un biais pas rapport au cas nominal d'exécution peut être relevé.

1.3.3.1.c Les machines virtuelles

D'après Goldberg [74], une machine virtuelle est « *un double efficace et isolé d'une machine réel* ». L'avantage de ces environnements virtuels est de pouvoir restaurer entièrement le système dont l'intégrité a été compromise. L'environnement peut être soit émulé, soit virtualisé :

- l'émulation consiste ici à imiter le comportement d'une machine physique au moyen de différents logiciels (CPU, mémoires, carte-mère, etc). Un exemple d'émulateur libre est représenté par le projet Bochs [108] qui permet d'émuler une architecture IA-32 ;
- la virtualisation consiste à exécuter nativement des instructions par le CPU d'une machine physique. Toutefois, certaines instructions « privilégiées » doivent être interceptées afin de garantir le cloisonnement entre la machine hôte et la machine invité (virtualisée). De nombreux outils de virtualisation sont aujourd'hui accessibles dont les principaux sont VMware [15], VirtualBox [178], VirtualPC [16] et QEMU [14]. Parmi les outils d'analyse s'appuyant sur des machines virtuelles, les plus utilisés sont sans doute Anubis [11] s'appuyant sur QEMU, et CWSandbox [183] généralement utilisé dans une machine virtuelle.

L'inconvénient de ces environnements virtuels réside principalement dans la possibilité de leur détection. En effet, les travaux de Ferrie [59] proposent un état de l'art des techniques de détection pour la plupart des machines virtuelles actuelles (Bochs, QEMU, VirtualBox, VirtualPC, VMWare et CWSandbox). Par exemple, le programme assembleur 1.4 décrit comment détecter VMWare en quelques lignes. Ce programme utilise l'instruction `in` d'accès en lecture sur un port d'entrée/sortie qui est normalement uniquement accessible en mode noyau, excepté pour VMWare qui l'intercepte et l'interprète comme une instruction virtuelle pour laquelle la valeur `'VMXh'` est retournée dans le registre `ebx`.

```
1 mov eax, 'VMXh'
2 mov ecx, 0ah        ; get VMware version
3 mov dx, 'VX'
4 in  eax, dx
5 cmp ebx, 'VMXh'
6 je  VMWareDetected
```

Listing 1.4 – Exemple de code assembleur permettant la détection de VMWare.

1.3.3.1.d La virtualisation matérielle

Les processeurs Intel et AMD actuels comprennent nativement des instructions dites de virtualisation. Ces technologies de virtualisation (Intel-VT et AMD-V) permet de filtrer au niveau du matériel les instructions privilégiées,

15. Disponible à l'URL : `http://www.vmware.com/` (dernier accès en décembre 2010)
16. Disponible à l'URL : `http://www.microsoft.com/windows/virtual-pc/` (dernier accès en décembre 2010).

les entrées/sorties ainsi que certains accès à la mémoire. Des environnements de virtualisation utilisant ces capacités matérielles ont alors vu le jour. Par exemple, l'outil Ether [53] s'appuyant sur l'hyperviseur Xen [9], propose un environnement d'analyse de code malveillant à base de virtualisation matérielle permettant de tracer un programme instruction par instruction ou bien au niveau de ses appels systèmes. Toutefois ces environnements restent détectables comme en témoigne les travaux de Desnos *et al.* [51].

Pour un code malveillant, déterminer la présence d'un environnement d'exécution émulé ou virtualisé peut lui permettre d'adapter son comportement afin de biaiser l'analyse. En cas d'absence d'un tel environnement, le comportement nominal malveillant peut être adopté. Dans le cas contraire, un autre comportement peut autoriser la non détection du programme observé. C'est pour cette raison que certains environnement d'analyse, comme joebox [22], s'appuient directement sur des machines physiques. L'inconvénient réside alors dans le temps nécessaire à la restauration de la machine physique contrairement aux machines virtuelles.

1.3.3.2 Approches par grammaires formelles

Jacob *et al.* [88] proposent un langage Turing-complet de spécification de comportements au moyen de grammaires formelles attribuées. Ces grammaires permettent, en plus des règles de productions des grammaires formelles, d'exprimer des règles sémantiques afin d'identifier et de typer des objets. Plusieurs comportements malveillants ont été spécifiés en tant que grammaires attribuées comme la réplication, la propagation, la résidence (capacité à se maintenir actif au sein d'un système après un redémarrage) et enfin le test de sur-infection. La figure 1.13 présente une grammaire attribuée décrivant le comportement de duplication. En résumé, cette grammaire exprime les différentes façons, pour un code, de se dupliquer sachant que les actions nécessaires sont de lire son image pour ensuite l'écrire dans un fichier préalablement créé.

```
    <Duplicate>            : :=   <Create><Open><Read><Write>
                            |     <Open><Create><Read><Write>
                            |     <Open><Read><Create><Write>
{   <Duplicate>.srcId      =      <Open>.objId
    <Duplicate>.srcTp      =      this
    <Duplicate>.targId     =      <Create>.objId
    <Duplicate>.targTp     =      obj_perm
    <Open>.objTp           =      <Duplicate>.srcTp
    <Create>.objTp         =      <Duplicate>.targTp
    <Read>.objId           =      <Duplicate>.srcId
    <Read>.objTp           =      <Duplicate>.srcTp
    <Read>.objId           =      <Duplicate>.targId
    <Read>.objTp           =      <Duplicate>.targTp
    <Write>.varId          =      <Read>.varId }
```

FIGURE 1.13 – Exemple de grammaire attribuée décrivant le comportement de duplication tiré de [88].

À partir d'une trace d'exécution correspondant à une suite d'appels aux

APIs systèmes et des paramètres associés, les auteurs utilisent des automates déterministes à pile avec évaluation des attributs sémantiques afin de détecter l'un des comportements malveillants spécifiés. Ces automates permettent de vérifier si la trace d'exécution est bien un mot du langage décrit par une grammaire attribuée. Les résultats obtenus pour 200 codes malveillants et 50 logiciels bénins révèlent un taux de détection de 51%, sans faux positif pour les signatures décrites. Les principaux problèmes identifiés concernent le comportement de propagation pour lequel la configuration réseau n'est pas forcément adaptée, ainsi que la rupture du flot de données à cause du manque de précision de l'outil de collecte.

1.3.3.3 Approches par comparaison dynamique de graphes

Les approches par comparaison de graphes visent à construire une représentation des comportements malveillants sous la forme de graphes représentant les dépendances entre les appels systèmes collectés. La figure 1.14 illustre une partie du comportement du vers NETSKY décrit sous forme d'un graphe représentant les appels systèmes observés ainsi que leurs dépendances.

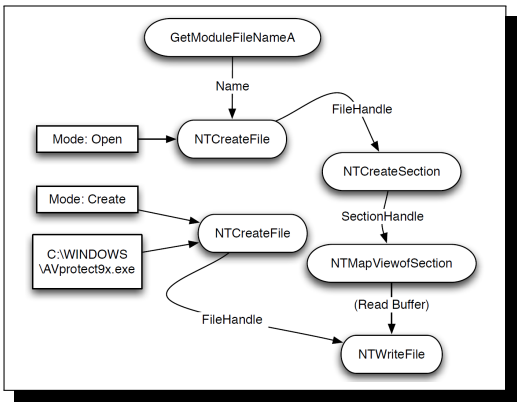


FIGURE 1.14 – Illustration de d'un graphe de dépendances entre les appels systèmes tiré de [99] représentant le vers NETSKY.

La difficulté dans la construction d'un graphe comportemental est de pouvoir précisément observer les dépendances entres les appels systèmes comme souligné dans l'approche de Jacob *et al.*. Autrement dit, le problème à résoudre est de savoir si un paramètre d'un appel système provient effectivement, après calculs intermédiaires, du résultat d'un appel précédent. Les différentes approches présentées tentent d'apporter une solution à la construction des graphes de dépendances des données.

1.3.3.3.a Approche de Yin *et al.*

Yin *et al* [188] proposent une architecture de détection dynamique par émulation de code reposant sur QEMU [14]. Leur solution s'appuie sur la technique dite de « *data tainting* » [133], qui consiste à marquer et à suivre la propagation de données sensibles au cours de l'exécution d'un programme. Ils peuvent ainsi construire un graphe de propagation de ces données à travers les processus, modules et autres ressources du système d'exploitation mis en jeu. Les auteurs identifient trois comportements jugés anormaux qu'ils cherchent à identifier : l'accès anormal à certaines informations, la fuite anormale d'informations et l'accès excessif à certaines informations. Les résultats obtenus montrent que 42 codes malveillants sont effectivement détectés (100% de vrais positifs) et que parmi les 56 programmes bénins, 3 sont détectés comme étant malveillants.

1.3.3.3.b Approche de Kolbisch *et al.*

Kolbisch *et al.* [99] présentent une architecture permettant un suivi précis du flot de données du programme analysé sans recourir au « *data tainting* ». Comme dans les travaux de Yin *et al*, un code malveillant est d'abord analysé dans un environnement contrôlé afin de construire un modèle de son comportement. Les modèles ainsi obtenus retranscrivent le flot d'informations entre les appels systèmes. Les auteurs utilisent des techniques dites de « *slicing* » [182] pour déterminer les instructions intervenant dans la manipulation des données entres deux appels systèmes, c'est-à-dire de quelle manière les entrées du deuxième appel sont générées à partir des sorties du premier. Leur approche originale consiste à récupérer les paramètres d'entrée ainsi que la valeur de retour au moment où le programme fait appel à une API. Cette valeur est ensuite donnée en entrée au « *slice* » précédemment défini pour en prédire la sortie. Plus tard, si la valeur du paramètre d'entrée de l'appel système suivant correspond effectivement à la valeur ainsi calculée alors le flot de données correspond bien au modèle. Cette approche leur permet de déployer la détection directement sur un poste utilisateur. Leur évaluation a portée sur 264 codes malveillants ainsi que 5 applications bénignes. Les résultats obtenus montrent un taux de détection de 64% sans faux positif.

1.3.3.3.c Approche de Frederikson *et al.*

Fredrikson *et al.* [68] poursuivent ces travaux de détection dynamique mais en adoptant une démarche différente. Les autres approches cherchent en effet à obtenir des graphes de dépendances les plus précis possibles afin de décrire et de comparer les comportements de différents programmes (malveillants et bénins). Le prix à payer est généralement conséquent pour les solutions proposées aussi bien en termes de charge processeur que de consommation en mémoire. Plutôt que de se focaliser sur un graphe de dépendance le plus précis décrivant des comportements malveillants à détecter, Fredrikson *et al.* cherchent à générer des spécifications comportementales discriminantes. Ces spécifications décrivent les propriétés propres à un ensemble de programmes en contraste avec un autre

ensemble de programmes. À partir d'une famille de codes malveillants identifiée par des anti-virus, ils génèrent un modèle comportemental le plus discriminant possible par rapport à un ensemble de programmes bénins. Cette approche leur permet de maximiser les résultats de détection qui atteignent 86% sur des codes malveillants inconnus, tout en minimisant les faux positifs (0%). Ces résultats sont obtenus sur 912 codes malveillants et 49 application bénignes.

1.3.4 Bilan de la détection

Nous avons présenté dans cette section les deux approches utilisées dans le cadre de la détection de codes malveillants métamorphes : la détection statique ainsi que celle dynamique.

La détection statique analyse directement l'image exécutable d'un programme pour permettre une détection un code malveillant avant exécution. Cependant, l'analyse statique d'un binaire est reconnue comme difficile. Des techniques de protection, telles que la compression de programmes, le chiffrement ou encore l'obscurcissement de code, sont couramment utilisées par les codes malveillants afin de limiter leur détection. Dans le cadre du métamorphisme, les travaux de détection statique essaient de s'affranchir des transformations d'obscurcissement de code utilisées par ces programmes métamorphes. Toutefois, ces approches sont adaptées aux cas simples d'obscurcissement de code tels que ceux rencontrés actuellement. Aucune des approches présentées dans cette section ne tient compte des techniques de protection logicielles exposées en section 1.2.5.

La détection dynamique consiste à faire abstraction des techniques de protection de code en exécutant directement un programme dans un environnement adapté pour l'observation. L'analyse porte alors sur les interactions du code en cours d'exécution avec le système d'exploitation, c'est-à-dire son comportement. Ainsi, la détection dynamique est utilisée pour palier aux difficultés de l'analyse statique. En contrepartie, cette technique présente plusieurs inconvénients :

- le premier problème rencontré réside dans la durée d'exécution d'un programme. En effet, une exécution interrompue prématurément peut engendrer des faux négatifs. Il suffit pour cela de considérer une bombe logique qui diffère sa charge au delà du temps d'exécution alloué pour la détection. De fait, cette durée impacte directement la *fiabilité* de la détection dynamique. À notre connaissance toutes les approches de détection dynamique proposées prédéfinissent une durée d'exécution arbitraire.
- le second problème réside dans l'environnement d'observation utilisé qui conditionne l'exécution et donc le comportement du code malveillant observé. En effet, un tel code peut comporter plusieurs actions malveillantes en fonction de : du système d'exploitation sur lequel il s'exécute, ses droits et privilèges, ainsi que les applicatifs présents, etc. Ce problème est abordé par Moser *et al.* [129]) qui proposent une architecture d'analyse dynamique dédiée permettant d'explorer plusieurs chemins d'exécution. Dans ce cas, une architecture complexe doit être mise en œuvre pour l'analyse.
- le dernier problème identifié est la compromission de l'environnement d'observation dans lequel un code malveillant a été exécuté. En effet, il apparaît

alors impératif de restaurer l'environnement hôte pour revenir à un état stable antérieur à l'exécution. Dans le cas d'un système dédié, qui joue un rôle de sas de décontamination, il suffit d'annuler les actions menées par le programme analysé. Ce cas, permettant de l'analyse multi-chemins et des techniques de « *data tainting* », nécessite de mettre en place un système de détection complexe du point de vue de l'utilisateur final. Par contre, dans le cas d'une détection sur un poste utilisateur, se pose alors le problème des actions à supprimer. En effet, toutes les actions menées par un programme infecté ne sont pas nécessairement malveillantes. Il suffit de considérer un document rédigé au moyen d'un traitement de texte infecté par un virus. Ce document, bien que non malveillant en soit risque d'être supprimé en tant qu'action menée par le virus détecté.

1.4 Bilan de l'état de l'art et problématique de la thèse

Dans ce chapitre nous avons présenté un état de l'art sur la notion de métamorphisme. Partant d'une définition informelle issue de l'étymologie de ce terme, notre présentation s'est orientée suivant trois axes : les fondements théoriques permettant de définir le métamorphisme, les mutations de code d'un point de vue implémentation et enfin la détection. La figure 1.15 présente un schéma récapitulatif de cet état de l'art dont nous résumons maintenant les grande lignes.

Dans la section 1.1, nous avons proposé une nouvelle définition du métamorphisme qui présente à nos yeux l'avantage d'en unifier les précédentes définitions au moyen d'une hiérarchisation du métamorphisme. Ainsi, les mutations de code intervenant dans le cadre du métamorphisme peuvent se formaliser au moyen d'une grammaire formelle G. Le premier niveau, le métamorphisme d'ordre 0, consiste à considérer chaque forme mutée v comme un mot de $L(G)$. Il s'agit de codes malveillants métamorphes tels que ceux identifiés actuellement. Pour le niveau suivant, c'est-à-dire le métamorphisme d'ordre 1, chaque variante v est un mot de $L(L(G))$. Dans ce cas, ce sont les règles de mutations elles-mêmes qui évoluent lors de chaque réplication. Cette définition permet alors d'envisager l'étude du métamorphisme pour des ordres supérieurs.

Dans la section 1.2, nous avons étudié les mutations intervenant dans le cadre du métamorphisme en tant que techniques d'obscurcissement de code. Nous avons alors présenté l'écart entre les techniques utilisées pour la protection logicielle, et celles employées dans le cadre du métamorphisme. Cet écart a été justifié par le fonctionnement d'un code métamorphe qui doit pouvoir extraire de son code obscurci son *archétype*. Ce constat est à l'origine de notre première problématique : un programme métamorphe peut-il utiliser des techniques d'obscurcissement de code avancées lui permettant de remonter à son *archétype* sans pour autant faciliter sa détection ?

Dans le section 1.3, nous avons exploré les différentes techniques de détection

adaptées au cas des codes métamorphes. Deux types de détection ont été présentés, les techniques de détection statique ainsi que celles de détection dynamique. Bien que la détection statique offre la sécurité d'une analyse avant exécution, elle est soumise aux difficultés inhérentes à l'analyse statique, à commencer par le désassemblage. La détection dynamique permet quant à elle de s'affranchir de ces limitations au prix d'inconvénients en termes de sécurité, de contrainte temporelle, et d'environnement d'exécution.

Dans ce mémoire, nous présentons un moteur générique de métamorphisme d'ordre 0 ainsi qu'une approche de détection dynamique fondée sur la similarité comportementale. Le moteur que nous proposons utilise un technique d'obscurcissement de code dont la résilience est prouvée dans le cadre de l'analyse statique. Nous appliquons ensuite ce moteur aux sources du ver MyDoom, que nous utilisons par la suite pour la classification « boîte noire » des techniques de détection utilisées par des anti-virus actuels représentatifs. Après cela, nous proposons une nouvelle mesure de similarité issue de la complexité de Kolmogorov pour évaluer le degré d'inclusion d'un objet dans un autre au moyen d'un algorithme de compression sans perte. Cette mesure ainsi qu'une distance de compression sont finalement évaluées dans le cadre de la détection des codes malveillants, au moyen d'une architecture de détection dynamique, conçue pour être directement déployable sur un poste utilisateur.

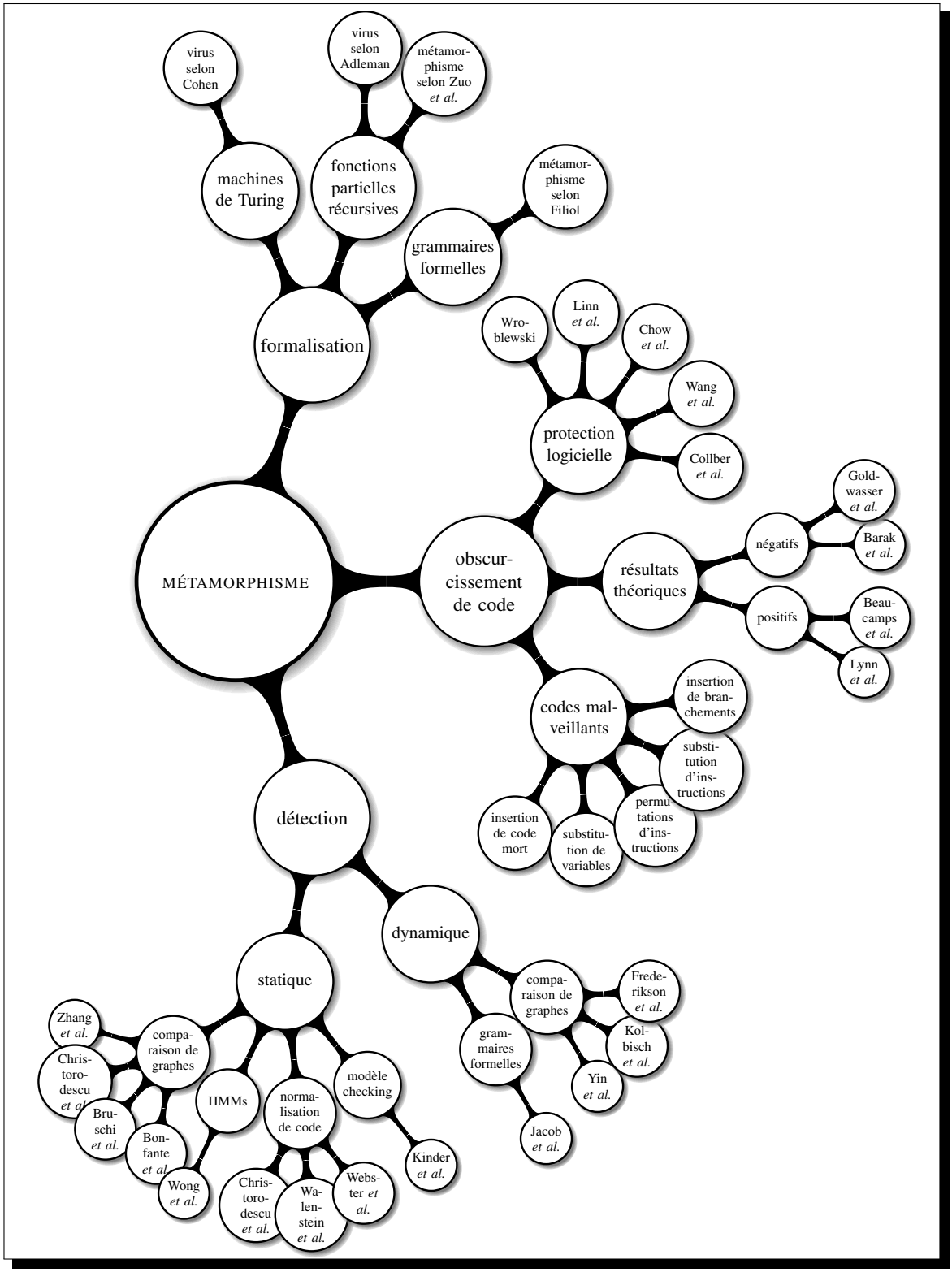


FIGURE 1.15 – Schéma récapitulatif de l'état de l'art sur le métamorphisme

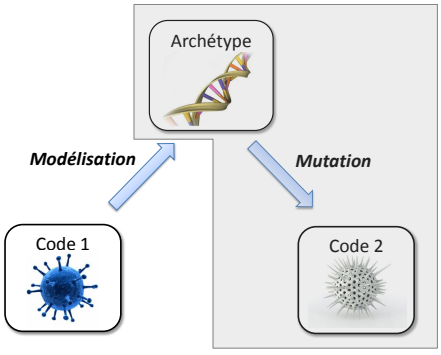
Première partie

Conception d'un moteur générique de métamorphisme

Chapitre 2

Approche d'obscurcissement de code adaptée aux programmes métamorphes

Ce chapitre présente notre étude concernant un premier aspect du processus d'auto-reproduction d'un programme métamorphe : l'obscurcissement de code. Nous reprenons pour cela le schéma simplifié de réplication d'un code métamorphe de la figure 1.6 dont la partie étudiée ici, à savoir la mutation de l'*archétype*, apparait grisée. Bien que chronologiquement la mutation de code n'intervienne qu'après la modélisation, c'est cette dernière partie que nous avons choisie de présenter en premier. Il apparait en effet plus simple d'exposer de quelle façon un nouveau code est construit à partir de l'*archétype* que son contraire, à cause du lien étroit existant entre la modélisation et les techniques d'obscurcissement de code utilisées.



Notre objectif est, dans un premier temps, de proposer un modèle théorique d'obscurcissement de code permettant d'en prouver la résilience pour, dans un

deuxième temps, déduire une approche d'obscurcissement de code pratique et efficace. Nous rappelons que la résilience se définit (voir définition 17 page 21) comme la difficulté, pour un outil dédié, à inverser l'obscurcissement de code employé. Ce modèle et l'approche qui en découle doivent toutefois satisfaire une double contrainte :

1. la première contrainte concerne les hypothèses sur lesquelles reposent la preuve de résilience. Ces hypothèses doivent en effet correspondre à des conditions réelles de détection des codes malveillants. En d'autres termes, nous devons vérifier que notre modèle est en accord avec les techniques employées par les outils de détection accessibles aujourd'hui au « grand public » ;
2. la seconde contrainte porte sur la spécificité des codes métamorphes, à savoir, leur capacité à pouvoir extraire leur propre *archétype*. Ainsi, notre modèle d'obscurcissement de code doit permettre au programme en cours d'exécution d'obtenir son propre *archétype* sans pour autant faciliter sa détection vis a vis de tout autre programme.

Afin de répondre à cet objectif, ce chapitre s'organise de la façon suivante : la section 2.1 présente notre modèle théorique permettant de prouver l'efficacité des transformations appliquées. Puis, la section 2.2 illustre le fonctionnement de cette approche d'un point de vue pratique. Finalement, la section 2.3 présente une série d'expériences dans le but de valider les hypothèses sur lesquelles repose la résilience de l'obscurcissement de code.

2.1 Difficulté de détection d'une catégorie particulière de codes métamorphes

Par nature, deux instances quelconques d'un code métamorphe présentent deux formes structurelles distinctes. Toutefois, ces deux implémentations partagent la même fonctionnalité. Partant de ce constat, notre étude s'appuie naturellement sur la notion d'équivalence fonctionnelle entre programmes.

2.1.1 Équivalence fonctionnelle

Définition 23. *(Équivalence fonctionnelle). Soient deux programmes (algorithmes) A et B dont les ensembles d'entrées possibles respectifs sont $\mathcal{D}_A$ et $\mathcal{D}_B$. A et B sont dit fonctionnellement équivalents, ce que nous notons $A \equiv B$, si et seulement si, ils produisent les mêmes résultats pour les mêmes entrées, soit :*

$$\begin{cases} \mathcal{D}_A = \mathcal{D}_B, \\ \forall x \in \mathcal{D}_A,\ A(x) = B(x). \end{cases}$$

Le fait que le calcul de A pour l'entrée x ne se termine jamais est représenté par $A(x) = \perp$.

Cette définition correspond bien entendu à une relation d'équivalence entre programmes au sens mathématique du terme (la relation étant réflexive, symétrique et transitive). Nous définissons alors la détection d'un code métamorphe V comme suit :

Définition 24. *(Détection d'un code métamorphe [18]). Le programme D_V détecte le programme métamorphe V (avec V tel que $\forall x, V(x) \neq \perp$) si, pour tout programme P,*

$$\begin{cases} D_V(P) = 1 \text{ si } P \equiv V, \\ D_V(P) = 0 \text{ sinon .} \end{cases}$$

En d'autres termes, tout programme P est détecté comme une instance du code métamorphe V si P est fonctionnellement équivalent à V. Notons maintenant $\overline{V}$ la classe d'équivalence de V, c'est-à-dire l'ensemble de programmes qui sont fonctionnellement équivalents à V et $\chi_{\overline{V}}$ la fonction caractéristique de $\overline{V}$. On remarquera alors que le détecteur D_V du programme métamorphe V n'est autre qu'une implémentation de la fonction caractéristique de la classe d'équivalence de ce programme V, soit : $D_V = \chi_{\overline{V}}$.

Théorème 11. *(Indécidabilité de l'équivalence fonctionnelle). Il n'existe pas d'algorithme capable de déterminer si, pour deux programmes quelconques P et P' non nuls, $P' \equiv P$.*

Ce résultat est un cas particulier du théorème de Rice [144] qui stipule que toute propriété non-triviale, c'est-à-dire non toujours vrai ou toujours fausse, concernant un langage Turing-complet est indécidable.

2.1.2 Obscurcissement du flot de contrôle

À partir de l'indécidabilité du problème de l'équivalence fonctionnelle, nous introduisons maintenant notre modèle d'obscurcissement du flot de contrôle.

Choix 1.

Notre modèle d'obscurcissement de code s'appuie sur la difficulté de la détermination précise des alias dans le cadre de l'analyse statique de programmes.

Définition 25. *(k-obscurcisseur de code [18]). Soit P un programme constitué de n instructions consécutives $I_1 I_2 \dots I_{n-1} I_n$. Nous définissons une transformation $\mathcal{T}$ de ce programme P comme suit :*

- *P est découpé en k blocs notés $P_1, P_2, \dots, P_k$. Chaque bloc contient un nombre aléatoire non nul d'instructions consécutives et se termine par une instruction séquentielle ;*
- *nous considérons une permutation de l'ensemble $[\![1, k]\!]$ dans lui-même, notée σ. Pour chaque bloc P_i, un nouveau bloc $P'_{\sigma(i)}$ est défini comme suit : chaque bloc $P'_{\sigma(i)}$ contient exactement les mêmes instructions que le bloc $P_{(i)}$ correspondant et se termine par 2 branchements conditionnels vers 2*

autres blocs P'_j et P'_l. Cette transformation définit alors la construction d'un programme P' à partir d'un programme original P, soit $P' = \mathcal{T}(P)$. Nous définissons alors un ensemble d'obscurcisseurs (k-obscurcisseurs), noté $\mathcal{O}_k$ comme suit : une telle transformation $\mathcal{T}$ est un k-obscurcisseur de code, ce que l'on notera $\mathcal{T} \in \mathcal{O}_k$, si et seulement si $\forall i \in [\![1, k]\!]$, lors de son exécution, le bloc $P'_{\sigma(i)}$ a pour sortie le bloc $P'_{\sigma(i+1)}$.

On remarquera que si $\mathcal{T} \in \mathcal{O}_k$ alors $\mathcal{T}(P) \equiv P$ et donc que $\mathcal{T}$ est bien un obscurcisseur de code puisqu'il respecte la propriété d'équivalence fonctionnelle.

Nous introduisons alors une catégorie particulière de codes métamorphes, utilisant le k-obscurcissement de code et dont le problème de détection se définit par :

Définition 26. *(Détection d'un programme métamorphe k-obscurci [18]). On considère un code métamorphe V dont une nouvelle instance V' est définie par $V' = \mathcal{T}(V)$ où $\mathcal{T} \in \mathcal{O}_k$. On dit alors qu'un programme D_V détecte le code métamorphe V si et seulement si pour tout programme P,*

$$\begin{cases} D_V(P) = 1 \text{ s'il existe } \mathcal{T} \in \mathcal{O}_k \text{ tel que } P = \mathcal{T}(V) \\ D_V(P) = 0 \text{ sinon.} \end{cases}$$

Proposition 1. *(Difficulté de la transformation inverse du k-obscurcissement de code [18]). Dans l'hypothèse où tous les chemins d'un programme sont exécutables, pour tous programmes P et P', savoir s'il existe $\mathcal{T} \in \mathcal{O}_k$ tel que $P' = \mathcal{T}(P)$ est un problème NP-complet.*

Avant d'apporter la preuve de cette proposition 1, revenons un instant sur l'hypothèse selon laquelle tous les chemins d'un programme sont exécutables. De manière générale, il existe plusieurs chemins dans un programme. Toutefois, tous ces chemins ne correspondent pas à une exécution. Par exemple, considérons les deux lignes de code suivantes :

```
if (a > 5) b = 1;
if (a < 6) b = 0;
```

Quelle que soit la valeur du paramètre d'entrée `a`, l'exécution de ce code conduit toujours à une unique affectation de `b`. En effet, il est impossible de trouver une valeur d'entrée `a` pour laquelle la variable `b` n'est jamais affectée, ou est affectée deux fois. Il s'agit là d'un cas simple pour lequel il est facile de déterminer quel chemin est effectivement emprunté. Malheureusement, dans le cas général, déterminer si un chemin est exécutable est indécidable par réduction au problème de l'arrêt [1]. Par conséquent, l'analyse statique est indécidable [107]. Pour surmonter ce problème, l'hypothèse couramment faite en analyse statique suppose que tous les chemins d'un programme sont supposés exécutables [2].

1. La preuve repose sur l'appel à un programme externe A. Dans ce cas, savoir si le chemin qui suit cet appel sera effectivement exécuté revient à savoir si A se termine.

D'un point de vue pratique, comme nous l'avons vue en section 1.2.4.1, l'utilisation de prédicats opaques peut considérablement compliquer l'évaluation du chemin effectivement emprunté lors de l'exécution d'un programme. C'est dans ces conditions que nous nous plaçons pour apporter une preuve de la résilience de notre approche.

Preuve. *Le problème est dans NP. En effet, pour tous programmes P et P' donnés, un algorithme non déterministe peut vérifier en temps polynomial s'il existe une transformation $\mathcal{T}$ conforme à la définition 25 telle que $P' = \mathcal{T}(P)$. De plus, cet algorithme peut aussi vérifier en temps polynomial, s'il existe une sortie de chaque bloc $P'(i)$ telle que que $\mathcal{T} \in \mathcal{O}_k$.*

Soit S une instance du problème NP-complet 3-SAT [92] et soit P un programme, nous démontrons qu'il est possible de construire en temps polynomial par rapport à la taille de S un programme P' ($P' = \mathcal{T}(P)$) tel que, S est satisfiable si et seulement si $\mathcal{T} \in \mathcal{O}_k$. La construction du programme P' à partir de P et de S constitue la réduction du problème 3-SAT au problème de l'existence d'une transformation $\mathcal{T} \in \mathcal{O}_k$ telle que $P' = \mathcal{T}(P)$.

Une instance S du problème 3-SAT est donnée sous la forme :

$$S = \bigwedge_{i=1}^{n} (l_{i,1} \vee l_{i,2} \vee l_{i,3})$$

$$\text{avec } \begin{cases} \{v_1, v_2, ..., v_m\} \text{ un ensemble de variables booléennes;} \\ \forall (i,j) \in [\![1,n]\!] \times [\![1,3]\!], \exists k \in [\![1,m]\!],\ l_{i,j} = v_k \text{ ou } l_{i,j} = \overline{v_k}. \end{cases}$$

Nous construisons une partition constituée d'éléments consécutifs de l'ensemble $[\![1,n]\!]$, notée $(u_i)_{i \in [\![1,k]\!]}$. La partition (u_i) peut se représenter sous la forme : $\underbrace{1,2,3}_{u_1}, \underbrace{4,5,\ldots,8}_{u_2}, \ldots, \underbrace{n-1,n}_{u_k}$. Notre instance S de problème 3-SAT s'écrit alors :

$$S = \bigwedge_{i=1}^{k} S_i \text{ avec } S_i = \bigwedge_{j=min(u_i)}^{max(u_i)} (l_{j,1} \vee l_{j,2} \vee l_{j,3})$$

Nous utilisons une transformation $\mathcal{T}$ de la définition 25 pour construire en temps polynomial le programme P' constitué de la famille des $(P'_i)_{i \in [\![0,k]\!]}$ à partir du programme P. Le schéma de la figure 2.1 illustre une telle réduction.

Comme tous les chemins sont supposés exécutables, nous ne tenons pas compte des conditions au niveau des branchements, ce que nous représentons sous la forme :

```
if (-) {code1} else {code2}
```

La figure 2.1 présente le pseudo-code contenu à l'intérieur du bloc P'_0. Ce code sert à l'initialisation du programme P'. Chaque variable V_i et sa négation $\overline{V_i}$ sont déclarées en première ligne comme des pointeurs de pointeurs de fonctions.

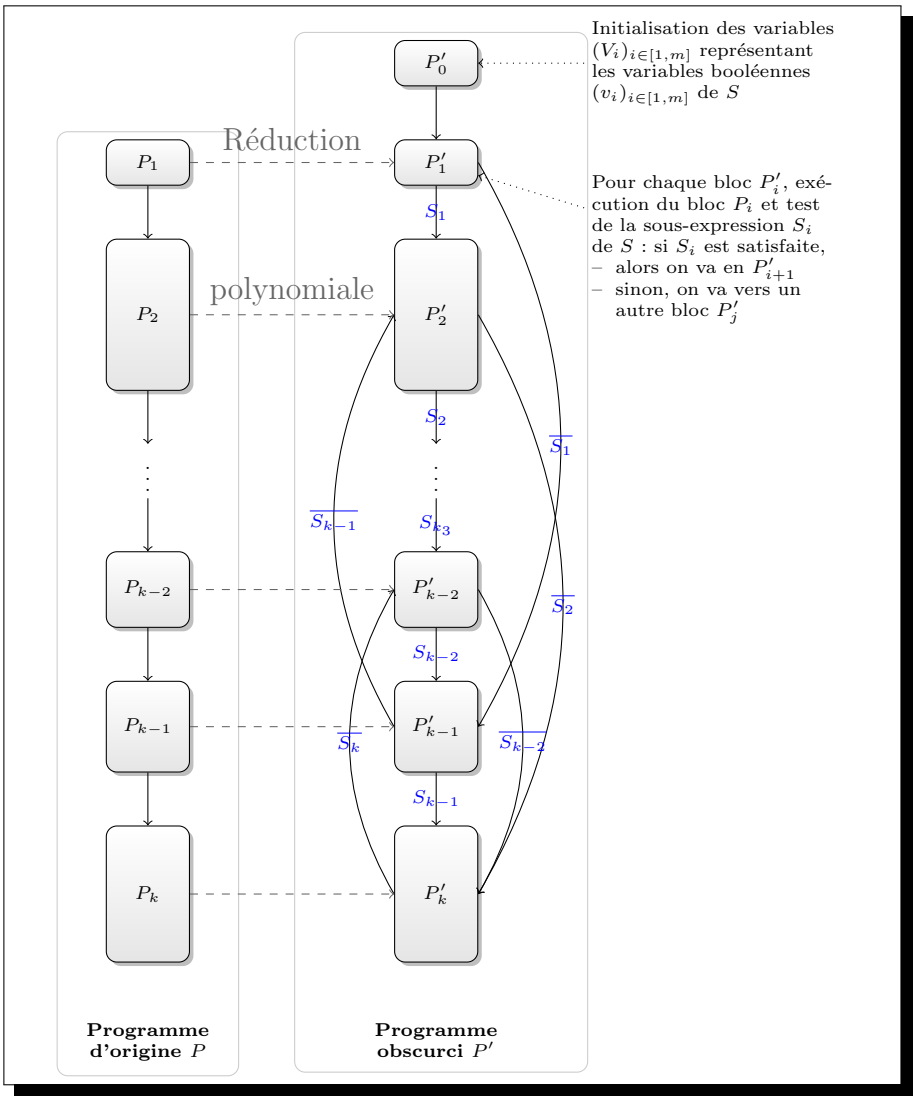


FIGURE 2.1 – Réduction polynomiale : construction du programme obscurci P' à partir de P et d'une instance S du problème 3-SAT.

Chacune de ces variables peut pointer sur `True` *ou* `False` *déclarées en ligne 2 comme des pointeurs de fonctions. Chaque* V_i *et* $\overline{V_i}$ *représentent respectivement une variable booléenne de* S *et sa négation. Ainsi, une affectation de la variable booléenne* v_i *à « vrai » pour* S *correspond à l'affectation* $*V_i$=`True` *pour* P'*. Un chemin d'exécution de* P'_0 *(lignes 3 à 6) initialise donc toutes les variables* V_i *et* $\overline{V_i}$*, ce qui correspond à fixer les valeurs des variables booléennes* v_i *pour notre équation* S*. Deux cas vont alors influencer le comportement du reste du*

programme P' : soit l'affectation des v_i satisfait S, soit elle ne satisfait pas S.

```
1 void (**V1)(),(**¬V1)(),...,(**Vm)(),(**¬Vm)();
2 void (*True)(),(*False)();
3 if (-) {V1=&True;¬V1=&False;} else {V1=&False;¬V1=&True;}
4 if (-) {V2=&True;¬V2=&False;} else {V2=&False;¬V2=&True;}
5 if (-) {Vm=&True;¬Vm=&False;} else {Vm=&False;¬Vm=&True;}
6 goto    P'1;
```

Listing 2.1 – Pseudo-code d'initialisation du bloc P'_0.

La figure 2.2 définit, pour tout i variant de 1 à $k-1$, le bloc P'_i à partir du bloc P_i. Le pointeur `False` *est initialisé avec l'adresse du bloc P'_{i+1}. La deuxième ligne représente l'insertion du code contenu à l'intérieur du bloc P_i de sorte qu'exécuter le bloc P'_i conduit à exécuter P_i. La notation $l_{i,j}$ dans les expressions de la forme $*l_{i,j}$=&P'_g (lignes 3 à 5) est une notation abstraite qui représente une variable V_k ou sa négation $\overline{V_k}$ (comme dans la formulation de S). Les lignes numérotées de 3 à 5 permettent de modifier la destination du pointeur* `False` *en fonction des valeurs des V_i et $\overline{V_i}$ initialisées en P'_0 comme nous allons le voir un peu plus loin. Finalement, la fonction* `False` *est appelée en ligne 6.*

```
1 False=&P_{i+1};
2 .../* insertion du code de P_i */
3 if (-) {*l_{min(u_i),1}=&P'_g;} else if (-) {*l_{min(u_i),2}=&P'_g;} else {*l_{min(u_i),3}=&P'_g;}
4    ...
5 if (-) {*l_{max(u_i),1}=&P'_g;} else if (-) {*l_{max(u_i),2}=&P'_g;} else {*l_{max(u_i),3}=&P'_g;}
6 False();
7 return;
```

Listing 2.2 – Pseudo-code des blocs du programme P'.

Le bloc P'_k diffère légèrement des autres blocs P'_i, $i \in [\![1, k-1]\!]$), pour terminer le programme.

Pour la suite, sauf précision, nous nous réfèrerons toujours à la figure 2.2. De plus, nous désignons par sortie *du bloc P'_i une des deux destinations possibles : P'_{i+1} ou P'_g. Nous montrons maintenant l'équivalence entre notre problème exprimé dans la proposition 1 et le problème 3-SAT.*

1. *Supposons que S soit satisfiable. Dans ce cas chaque sous-expression S_i ($i \in [\![1, k]\!]$) est satisfiable. Or, S_i satisfiable correspond au niveau du pseudo-code à $\forall j \in [\![\min(u_i), \max(u_i)]\!]$, $l_{j,1}$ ou $l_{j,2}$ ou $l_{j,3}$ pointe sur la variable* `True`*. Comme au moins un littéral de la forme $l_{j,t}$, $t \in [\![1, 3]\!]$ pointe sur la variable* `True` *alors il existe au moins un chemin dans ce graphe pour lequel la destination du pointeur* `True` *est affectée à l'adresse P'_g. Cette propriété étant vraie pour toutes les lignes de 3 à 5, il existe donc un chemin de P'_i pour lequel seule la variable* `True` *à été réaffectée à l'adresse du bloc P'_g à la ligne 6 et donc pour lequel la variable* `False` *n'a*

pas été redéfinie. Dans ce cas, la sortie *du bloc P'_i est le bloc P'_{i+1}. Cette propriété étant vérifiée pour tous les P'_i, il en résulte que $\mathcal{T} \in \mathcal{O}_k$. Nous obtenons donc que, si S est satisfiable alors $\mathcal{T} \in \mathcal{O}_k$.*

2. *Réciproquement, supposons maintenant que $\mathcal{T} \in \mathcal{O}_k$. Dans ce cas, pour tout i, la* sortie *du bloc P'_i est le bloc P'_{i+1}. Autrement dit pour chaque P'_i, `False` pointe sur le bloc P'_{i+1} en ligne 5. Or `False` pointe sur le bloc P'_{i+1} uniquement si, pour le chemin de P'_i considéré, chaque littéral de la forme $l_{j,t}$ pointe sur la variable `True`. En effet, si pour ce chemin, un seul littéral $l_{j,t}$ pointait sur `False`, alors `False` se verrait affecté l'adresse du bloc P'_g dans une des lignes de 3 à 5 et pointerait donc sur un autre bloc que P'_{i+1} en ligne 6. Ce résultat pour le bloc P'_i implique que S_i est satisfiable. Comme cette relation doit être vérifiée pour tout i de 1 à k, S est satisfiable, d'où le résultat attendu : si $\mathcal{T} \in \mathcal{O}_k$ alors S est satisfiable.*

Nous obtenons donc l'équivalence entre notre problème et le problème 3-SAT. □

Corollaire 1. *(Difficulté de détection des virus métamorphes [18]). La détection d'un code métamorphe telle que définie en définition 26 est un problème NP-complet.*

À partir de ce résultat exprimant la difficulté de détection d'une catégorie de codes métamorphes d'un point de vue de l'analyse statique, nous proposons une approche d'obscurcissement de code qui pourrait être utilisée par des codes métamorphes plus complexes que ceux connus actuellement.

2.2 Approche pratique de *k*-obscurcissement de code pour des programmes métamorphes

Notre approche d'obscurcissement de code concerne à la fois le flot de contrôle et celui de données. L'obscurcissement du flot de contrôle est traité en section 2.2.1 ; celui du flot de données est présenté en section 2.2.2.

2.2.1 Obscurcissement du flot de contrôle

En ce qui concerne le flot de contrôle, notre approche s'appuie sur la démonstration de la proposition 2.1.2. Notre algorithme est inspiré de celui présenté dans les travaux de Chow *et al.* [32], s'appuyant sur un automate à états finis, et de celui de Ogiso *et al.* [135], utilisant des pointeurs de fonctions. Toutefois, notre approche d'obscurcissement de code présente deux aspects spécifiques :

1. d'une part, elle s'applique directement sur un source assembleur. Ainsi, les techniques d'obscurcissement de code présentées précédemment restent compatibles avec notre approche. En effet, les techniques portant sur un langage de plus haut niveau (voir section 1.2.5 page 28) s'appliquent avant la chaîne de compilation qui produit les sources assembleur sur lesquelles porte notre obscurcissement de code. Celles présentées en sections 1.2.6

peuvent s'appliquer sur les sources assembleur puisque notre approche ne fait pas d'hypothèse particulière sur les sources d'entrée.

L'avantage de travailler sur un source assembleur est de pouvoir disposer de toutes les informations du source de haut niveau (comme par exemple les langages C et C++) tout en étant sûr de conserver les transformations appliquées. Cela n'est pas forcément le cas dans une chaîne de compilation classique où les optimisations successives peuvent simplifier une partie de l'obscurcissement de code appliqué ;

2. d'autre part, le processus d'obscurcissement de code proposé est conçu pour prendre en compte la spécificité des codes métamorphes, à savoir la nécessité de parvenir à se modéliser tout en maintenant la complexité de l'analyse statique. Ce point particulier n'est toutefois pas abordé ici mais sera détaillé dans le chapitre suivant.

L'approche globale de construction d'un programme obscurci P' à partir d'un programme original P peut se résumer comme suit :

1. comme dans la démonstration de la proposition 1, nous découpons le programme P, constitué d'une suite de n instructions $(I_j)_{j\in[\![1,n]\!]}$, en k ensembles de taille aléatoire non nulle d'instructions consécutives pour former les blocs $(P_i)_{i\in[\![1,k]\!]}$. Ce découpage est fait de telle sorte que chaque bloc P_i ne se termine ni par un saut inconditionnel (`goto`), ni par un retour de procédure (`ret`). Sans cette restriction, la condition de passage de P_i à P_{i+1} serait inutile.

P_1	`mov ebp, 7ABBEDE5h`
	`mov dword_40C89E, 0B7777E5Fh`
	`and ebp, dword_40C89E`

P_2	`mov dword_40C2F4, 'NrEK'`
	`mov eax, dword_40C2F4`
	`lea ebx, ds:'llD.'`
	`lea edx, [ebx]`
	`lea edi, [edx+0]`

P_3	`mov dword_40C0B4, edi`
	`mov dword_40C0B0, ebp`
	`mov dword_40C0AC, eax`
	`lea edi, large ds:0`
	`mov dword_40C0B8, edi`
	`push offset dword_40C0AC`
	`pop dword_40C6C4`
	`mov ecx, dword_40C6C4`

P_4	`mov dword_40C1F4, ecx`
	`push dword_40C1F4`
	`pop dword_40C9BF`
	`mov eax, dword_40C9BF`

P_5	`push eax`
	`lea edi, GetModuleHandleA`
	`call dword ptr [edi]`

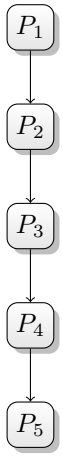


FIGURE 2.2 – Exemple de découpage en blocs (P_i) pour une amorce possible du virus WIN32.METAPHOR.

La figure 2.2, présente un exemple de découpage du code assembleur correspondant au premier bloc de base du CFG du virus METAPHOR. L'explication détaillée du code n'est pas nécessaire pour la compréhension de la suite. L'enchaînement des blocs est ici linéaire soit P_1, puis P_2, et ainsi de suite jusqu'à P_5 ;

2. nous construisons $(G_i)_{i\in[\![1,p]\!]}$ une famille de séquences d'instructions qui représente un bloc de « code mort » . Chacun de ces blocs comporte un nombre aléatoire non nul d'instructions cohérentes. Par cohérente, nous désignons une suite d'instructions provenant d'un programme réel et non pas une suite d'instructions générée de manière aléatoire. Le but de ces blocs est d'augmenter la complexité de l'analyse statique du programme P'. En effet, le choix d'instructions issues de programmes réels augmente la similarité syntaxique entre le code obscurci et un programme quelconque. Toutefois, ces blocs ne sont jamais exécutés et constituent de fait du « code mort » . La figure 2.3 présente des exemples simples de « codes morts » dont la compréhension exacte n'est pas nécessaire pour la suite des explications.

G_1	`xor ebx, ebx`
	`mov ebp, 24h`
	`mov eax, 26h`

G_2	`mov ebp, esp`
	`xor edi, edi`
	`mov eax, 1F03Fh`

FIGURE 2.3 – Exemples de blocs de code morts (G_i).

3. Nous construisons maintenant le programme obscurci P' constitué des blocs $(P'_i)_{i\in[\![0,k+p]\!]}$ de la façon suivante :
   (a) $P'(0)$ constitue le bloc d'initialisation du programme obscurci. Il initialise une donnée K conditionnant l'unique aiguillage en sortie de chaque bloc P'_i de sorte que $P' \equiv P$. Cette information K constitue donc l'élément essentiel de l'obscurcisseur de code, qui sera désignée par la suite sous le terme de *paramètre d'obscurcissement*.
   (b) $\forall i \in [\![1, k+p]\!]$, deux choix se présentent pour la construction du bloc P'_i :
   – soit P'_i est un bloc légitime, c'est-à-dire qu'il existe un unique entier s compris entre 1 et k tel que P'_i contienne le code de P_s. Dans ce cas, nous définissons une sortie légitime vers le bloc P'_{i+1} et une autre sortie choisie aléatoirement parmi les autres blocs.
   – soit P'_i est un bloc illégitime (code mort), c'est-à-dire qu'il existe un unique entier t compris entre 1 et p tel que P'_i contienne le code de G_t. Dans ce cas, nous définissons deux sorties aléatoires.

K est l'élément essentiel à une reconstruction statique du programme P à partir de P'. Ce paramètre d'obscurcissement peut être initialisé au moyen de prédicats opaques. Nous avons vu en section 1.2.4.1 plusieurs possibilités afin de concevoir des prédicats opaques comme l'utilisation de calculs arithmétiques ou encore l'emploi de conjectures mathématiques. L'objectif ici consiste à illustrer

le fonctionnement de notre approche d'obscurcissement de code et non pas à décrire précisément les mécanismes mis en jeu. Les détails d'implémentation sont donnés au chapitre suivant.

2.2.2 Protection des données

Une première approche afin protéger la confidentialité des données d'un programme consiste à les chiffrer comme dans le cas des codes malveillants chiffrés et ceux polymorphes. La différence consiste à déchiffrer, puis à rechiffrer « à la volée » les données du programme de sorte qu'elles apparaissent en clair dans le mémoire uniquement lors de leur utilisation.

2.2.2.1 Chiffrement à la volée des données

Afin de protéger la confidentialité des données qui pourraient représenter un motif potentiel de détection statique, une façon classique de procéder de consiste à déchiffrer ces données juste avant leur accès, puis, à les rechiffrer juste après utilisation. Par données, nous désignons tout bloc de données élémentaires, c'est-à-dire qu'un tel bloc ne peut être divisé sans perte sémantique (par exemple, une chaîne de caractère, une table de `switch`, une structure, etc.). Cette technique de chiffrement/déchiffrement à la volée est déjà employée par certains codes malveillants (parmi lesquels DarkParanoid [93], W32/Elkern [58] et Whale [63]).

Soit f une fonction prenant en entrée un bloc de données noté D. Notre programme original P calcule, à un moment de son exécution, cette fonction f sur la donnée D. Soit E un algorithme de chiffrement symétrique. Nous modifions le programme P afin d'obtenir le programme P' de sorte que P' contienne (dans son code binaire) une clé de chiffrement K et une donnée chiffrée $C = E_K(D)$. Durant son exécution, P' commence par déchiffrer la donnée chiffrée C. Après quoi, P' calcule la fonction f avec pour entrée la donnée précédemment déchiffrée D. Finalement, P' rechiffre cette même donnée D avec la même clé K. Aussi, sans la connaissance de la clé K', la confidentialité de la donnée chiffrée D est garantie par la robustesse de l'algorithme de chiffrement.

Le processus d'obscurcissement des données consiste alors à rendre aléatoire la position ainsi que la valeur de la clé K de façon à ce que seule la portion de code ayant préalablement accès à cette clé, ait toujours accès à la nouvelle clé générée. Le nouveau programme contient alors la nouvelle clé de chiffrement K' associée à la nouvelle donnée chiffrée $C' = E_{K'}(D)$. Dans ce cas, nous supposons que le seul moyen d'obtenir la clé de déchiffrement, autre que par force brute, consiste à désassembler le code. Ainsi, la protection des données repose directement sur l'efficacité (résilience) de l'obscurcissement de code employé.

2.2.2.2 Protection par génération de clés environnementales

Pour la protection des données, d'autres approches plus avancées sont possibles, comme la génération de clés environnementales proposée initialement

par Riordan et Schneier [146] et reprise par Filiol pour la conception du virus Bradley [62].

Le principe proposé par Riordan et Schneier repose sur la génération de clés cryptographiques propres à un environnement d'exécution. Des données environnementales sont dérivées au moyen d'une fonction à sens unique, dite fonction de hachage, pour obtenir une clé de chiffrement symétrique. Plusieurs constructions sont proposées dans [146]. Pour cela, considérons N un entier correspondant à une donnée environnementale observée, H une fonction à sens-unique, M la valeur du haché de l'observation N souhaitée et K une clé. La valeur M est contenue dans le code obscurci. La génération de la clé K peut être obtenue comme suit :

- si $H(N) = M$ alors $K = N$,
- si $H(H(N)) = M$ alors $K = H(N)$,
- si $H(N_i) = M_i$ alors $K = H(N_1|\dots|N_i)$ avec $\forall i \in \mathbb{N}, H(N_i) = M_i$ et chaque N_i correspond à une donnée environnementale observée [2].

Dans [62], l'utilisation de données environnementales propres à une machine précise permet de concevoir un code malveillant ciblé pour lequel l'analyse n'est possible que si la donnée utilisée est connue. En effet, le code viral, s'exécutant sur une cible donnée, déchiffre son corps au moyen de la clé environnementale propre à cette cible. Si cette clé n'est pas correcte, c'est-à-dire si le virus ne s'exécute pas sur la cible pour laquelle il a été conçu, alors le déchiffrement ne produira pas le code escompté, ce qui provoquera un arrêt du programme. Il en est bien sûr de même pour l'analyste, qui ne dispose pas de la bonne donnée environnementale. L'avantage de cette approche est d'interdire l'analyse du code sans la connaissance des données environnementales utilisées sur la cible visée. L'inconvénient réside dans l'impossibilité d'exécuter le code sur une autre machine ne disposant pas de la donnée ciblée.

Choix 2 (Protection des données).

Afin de produire un obscurcisseur de code le plus générique possible, nous optons pour la première technique proposée : le déchiffrement et le rechiffrement à la volée des données dont la sécurité repose sur celle de l'obscurcissement de code utilisé.

2.2.3 Illustration de la difficulté d'analyse statique d'un programme obscurci

À titre d'exemple, la figure 2.4 reprend l'amorce du virus MetaPHOR de la figure 2.2 en illustrant les transitions inter-blocs du programme obscurci P' obtenu à partir d'un programme P.

Le bloc P'_0 initialise les variables booléennes $(x_i)_{i\in[\![1,6]\!]}$ conditionnant les sorties de chaque bloc. Par exemple, pour le bloc P_1, si x_1 vaut 1, alors le

2. le symbole | désigne la concaténation.

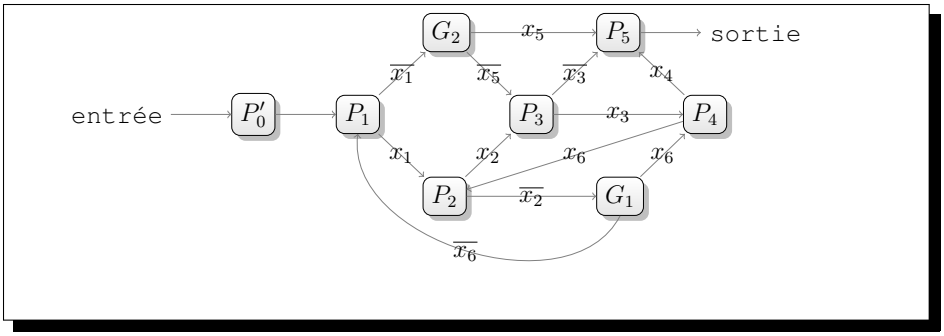


FIGURE 2.4 – Exemple d'obscurcissement possible du programme P.

prochain bloc exécuté est P_2, sinon le bloc suivant est G_2. Pour que l'exécution du programme P' corresponde effectivement au programme P, il faut conserver l'enchaînement des blocs comme dans la figure 2.2, ce qui correspond dans ce cas, à $\forall i \in [\![1,4]\!]$, $x_i = 1$. **Sans la connaissance du contexte K, le nombre de chemins d'exécutions possibles est de 2^{k+p-1}.**

Les variables x_5 et x_6 conditionnent uniquement les sorties des blocs de code mort G_1 et G_2, blocs qui ne sont jamais exécutés dans le cas où $P' \equiv P$. Le tableau suivant ne présente donc que les 16 premières valeurs de K possibles en supposant que x_5 et x_6 sont nuls. Nous considérons maintenant les exécutions du programme obscurci V' pour les 16 cas possibles d'affectations des $(x_i)_{i\in[\![1,4]\!]}$ dont les résultats sont résumés dans le tableau 2.1.

K	**Affectations des** (x_i)	**Chemins d'exécution**	**Résultats du programme** P'
0	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, \overline{x_2}, \overline{x_1}$	P_1, G_2, P_3, P_5	erreur fatale
1	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, \overline{x_2}, x_1$	$(P_1, P_2, G_1)^*$	boucle sans fin
2	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, x_2, \overline{x_1}$	P_1, G_2, P_3, P_5	erreur fatale
3	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, x_2, x_1$	P_1, P_2, P_3, P_5	erreur fatale
4	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, \overline{x_2}, \overline{x_1}$	$(P_1, G_2, P_3, P_4, P_2, G_1)^*$	boucle sans fin
5	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, \overline{x_2}, x_1$	$(P_1, P_2, G_1)^*$	boucle sans fin
6	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, x_2, \overline{x_1}$	$P_1, G_2, (P_3, P_4, P_2)^*$	boucle sans fin
7	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, x_2, x_1$	$P_1, (P_2, P_3, P_4)^*$	boucle sans fin
8	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, \overline{x_2}, \overline{x_1}$	P_1, G_2, P_3, P_5	erreur fatale
9	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, \overline{x_2}, x_1$	$(P_1, P_2, G_1)^*$	boucle sans fin
10	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, x_2, \overline{x_1}$	P_1, G_2, P_3, P_5	erreur fatale
11	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, x_2, x_1$	P_1, P_2, P_3, P_5	erreur fatale
12	$\overline{x_6}, \overline{x_5}, x_4, x_3, \overline{x_2}, \overline{x_1}$	P_1, G_2, P_3, P_4, P_5	incorrect
13	$\overline{x_6}, \overline{x_5}, x_4, x_3, \overline{x_2}, x_1$	$(P_1, P_2, G_1)^*$	boucle sans fin
14	$\overline{x_6}, \overline{x_5}, x_4, x_3, x_2, \overline{x_1}$	P_1, G_2, P_3, P_4, P_5	incorrect
15	$\overline{x_6}, \overline{x_5}, x_4, x_3, x_2, x_1$	P_1, P_2, P_3, P_4, P_5	**correct**

TABLE 2.1 – Résultats de l'exécution du programme P' pour différentes valeurs de K.

Le tableau 2.1 expose quatre types de résultats regroupés comme suit :
- 7 cas ($K = 1, 4, 5, 6, 7, 9, 13$) correspondent à une boucle sans fin pour laquelle l'instruction `call` n'est jamais atteinte. Ce qui implique une non détection du bloc comme appartenant au virus ;
- 6 cas ($K = 0, 2, 3, 8, 10, 11$) entraînent une erreur de l'application lors de l'appel à la fonction `GetModuleHandle`, erreur due à un paramètre incorrect qui ne constitue pas un pointeur valide ;
- 2 cas ($K = 12$ et 14) renvoient 0 après l'appel ; Ici, le pointeur passé en paramètre est bien valide mais ne pointe pas sur la chaîne recherchée `"Kernel32.dll"` ;
- un seul cas ($K = 15$) correspond à l'appel recherché conformément au choix de la valeur de K ayant servi à la construction de P'.

Cet exemple montre l'impact concret du résultat théorique sur une détection statique dans le cas où le paramètre d'obscurcissement K n'est pas connu.

2.3 Validation de l'hypothèse d'analyse statique pour la détection des codes métamorphes par des anti-virus actuels représentatifs

Comme nous l'avons vu en section 1.3.2, il existe de nombreuses approches de détection de codes malveillants métamorphes s'appuyant sur des techniques issues de l'analyse statique de binaire. La question que nous abordons ici est de savoir si les logiciels anti-virus « grand public » en font de même pour des menaces métamorphes.

Afin de répondre à cette question, cette section présente les résultats de détection obtenus sur des virus métamorphes que nous obscurcissons dans le but de déterminer si les produits testés reposent sur des techniques d'analyse statique ou non. Pour cela, nous employons la démarche suivante comportant quatre expériences :

1. dans une première expérience, qui peut être vue comme un test de calibrage, nous comparons les capacités de détection d'un panel d'anti-virus confrontés à un virus métamorphe donné. L'objectif est de sélectionner uniquement les outils capables de détecter le virus métamorphe initial ;
2. dans une seconde expérience, nous localisons quelle partie du virus (qui en contient deux distinctes) est détectée afin de simplifier l'étape suivante d'obscurcissement de code ;
3. dans une troisième expérience, nous soumettons des versions obscurcies du virus métamorphe [3] afin d'exclure les outils employant uniquement des techniques d'analyse statique. L'hypothèse sur laquelle est fondée cette expérience est que l'emploi de techniques d'évaluation dynamique de code

3. Il convient de remarquer que le virus métamorphe déjà obscurci par nature l'est une seconde fois au moyen de techniques que nous détaillons au moment de l'expérience.

(comme l'émulation) permet de détecter un virus obscurci. Par contraposée, nous estimons donc que si un virus obscurci n'est pas détecté, alors la détection ne repose pas sur une technique d'évaluation dynamique du code ;

4. dans une quatrième et dernière expérience, nous nous focalisons sur les anti-virus restants pour identifier si le code viral est réellement émulé afin d'être détecté. En effet, l'expérience précédente nous montre que si le virus n'est pas détecté, alors la détection s'appuie sur une analyse statique du binaire. Toutefois, en cas de détection, rien ne nous garantit que la détection se fait par émulation du code. Nous utilisons alors des versions inertes de virus métamorphes pour identifier des faux positifs qui révèlent la non-émulation du code analysé et donc une détection par analyse statique.

2.3.1 Description du contexte expérimental

Avant de détailler chaque expérience, nous présentons notre contexte expérimental composé de trois éléments :

1. une souche virale métamorphe de référence, qui sera utilisée pour toutes les expériences présentées ;
2. les programmes cibles qui vont servir d'hôtes pour le virus choisi ;
3. une collection d'anti-virus, représentatifs de ceux utilisés au moment des expériences, en charge de la détection des programmes infectés.

2.3.1.1 La souche virale métamorphe

Nous rappelons que cette section 2.3 a pour objectif de valider l'hypothèse d'analyse statique pour la détection de codes malveillants métamorphes. Nous utilisons donc une souche virale métamorphe, à savoir le virus MetaPHOR, détaillé en section 1.2.7.2. Pour la suite, il est nécessaire de revenir sur la structure de ce virus métamorphe, composé de deux parties :

1. une première partie, qui peut être vue comme le « chargeur », est responsable du déchiffrement en mémoire du reste du virus avant de l'exécuter ;
2. une seconde partie, qui correspond au corps chiffré du virus.

Toutes les expériences qui suivent, utilisent exclusivement ce virus.

2.3.1.2 Les programmes hôtes

Comme programmes à infecter, nous considérons la suite logicielle Sysinternals [150] composée de 70 outils communément utilisés sur les systèmes d'exploitation Windows. Parmi ces programmes, seuls 57 se sont avérés infectables par le virus considéré. Les 13 restants ne peuvent être infectés à cause de leur structure. En effet, le virus cible uniquement des fichiers exécutables dont la dernière section est accessible en écriture.

Notre objectif consiste maintenant à obtenir un nombre suffisant de souches virales syntaxiquement différentes. Comme le virus considéré est métamorphe, deux binaires identiques contiendront deux souches virales de formes différentes après infection. Nous pouvons donc dupliquer les 57 programmes hôtes initiaux tout en étant sûr d'obtenir, au final, des programmes infectés de formes différentes. Ainsi, nous dupliquons 18 fois chaque outil infectable pour aboutir à un panel de 1026 programmes cibles. Après infection par notre virus, ces programmes vont être utilisés au cours des expériences suivantes en tant que programmes infectés représentatifs. Nous rappelons que la souche initiale étant métamorphe, les 1026 programmes infectés sont alors syntaxiquement différents et nous disposons donc bien 1026 souches virales de formes différentes. En fait, dupliquer les 57 programmes initiaux nous permet de simplifier la localisation du « chargeur ». En effet, nous observons que la première partie du virus, bien que syntaxiquement différente, est toujours localisée à la même position pour un programme hôte donné, du moment que la section à infecter est suffisamment large pour contenir la première partie du virus. Dans le cas contraire, le « chargeur » est écrit dans la dernière section du programme. Connaissant la position de la première partie du virus dans un programme donné, il est alors facile d'obscurcir les 17 autres.

2.3.1.3 Les anti-virus utilisés

Afin d'être le plus reproductible possible dans nos expérimentations, nous utilisons des services anti-viraux accessibles sur Internet. Les 32 anti-virus à jour, hébergés par le site *Virus Total* [4], correspondent aux principaux outils de détection utilisés au moment de nos expériences, conduites en mai 2008 [5]. Ces logiciels constituent notre référentiel de détection. Les résultats de détection obtenus sont présentés de manière anonyme dans le but de valider notre hypothèse d'analyse statique et non pas d'établir un comparatif entre les anti-virus testés.

2.3.2 Première expérience : test d'un panel d'anti-virus

Cette première expérience consiste à tester le taux de détection des anti-virus disponibles. Les 1026 programmes infectés par le virus Win32.MetaPHOR sont soumis à notre panel d'outils de détection. En tant que virus métamorphes, ces programmes constituent des formes infectées du même virus. Les résultats obtenus sont représentés en figure 2.5 qui donne les taux de détection des 32 anti-virus considérés, par ordre croissant, pour 1026 fichiers infectés.

Cette première expérience montre que tous les logiciels testés ne sont pas capables de détecter l'intégralité des virus métamorphes soumis. Nous distinguons

4. Ce site est consultable à l'URL suivante : `http://www.virustotal.com` (dernier accès en décembre 2010).

5. Les 32 anti-virus disponibles au moment des expériences sont (par ordre alphabétique) : AhnLab-V3, AntiVir, Authentium, Avast, AVG, BitDefender, CAT-QuickHeal, ClamAV, Dr-Web, eSafe, eTrust-Vet, Ewido, Fortinet, F-Prot, F-Secure, Gdata, Ikarus, Kaspersky, McAfee, Microsoft, NOD32v2, Norman, Panda, Prevxl, Rising, Sophos, Sunbelt, Symantec, TheHacker, VBA32, VirusBuster, Webwasher-Gateway.

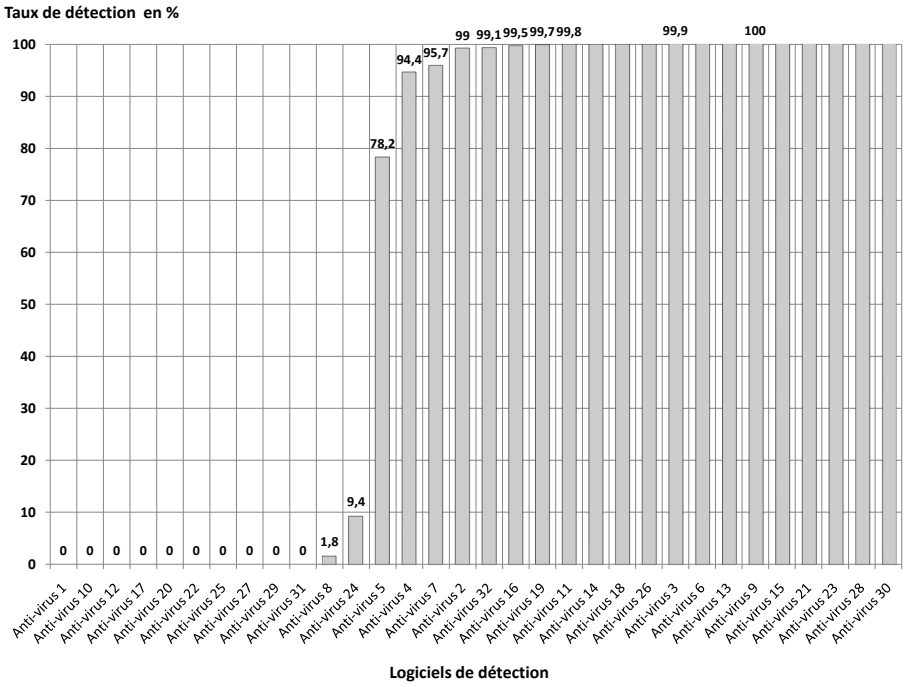


FIGURE 2.5 – Taux de détection du virus WIN32.METAPHOR soumis à 32 logiciels anti-virus.

alors trois catégories de résultats :

- une première catégorie comprend 6 programmes capables de détecter toutes les instances virales soumises ;
- une seconde catégorie est composée de 16 programmes capables d'identifier certains fichiers infectés avec des taux de détection variant de 1,8% à 99,9% ;
- une dernière catégorie comprend 10 programmes incapables de détecter le moindre fichier infecté.

2.3.3 Deuxième expérience : localisation de la partie détectée du virus

Cette seconde expérience emploie les logiciels de détection dont le taux de vrais positifs dépasse les 90%. Nous avons choisi ce seuil de 90% comme limite de *fiabilité* acceptable pour un utilisateur. On remarquera que le choix d'un seuil plus élevé conduirait aux mêmes résultats.

Comme le corps du virus WIN32.METAPHOR est chiffré, nous supposons que cette partie ne peut être détectée avant l'exécution (ou l'émulation) du « chargeur ». Le but de cette expérience est de vérifier si cette partie chiffrée est

utilisée par les anti-virus sélectionnés ou bien si la détection ne porte que sur la première partie du virus. Afin de répondre à cette question, nous supprimons la partie chiffrée du virus sur les 1026 fichiers infectés pour les soumettre ensuite à la détection. Nous désignons les fichiers ainsi obtenus comme étant « inertes » puisqu'ils ne possèdent pas de charge utile et ne se répliquent pas [6]. Les résultats sont illustrés sur la figure 2.6 qui présente l'évolution des taux de détection entre les fichiers originaux et ceux inertes.

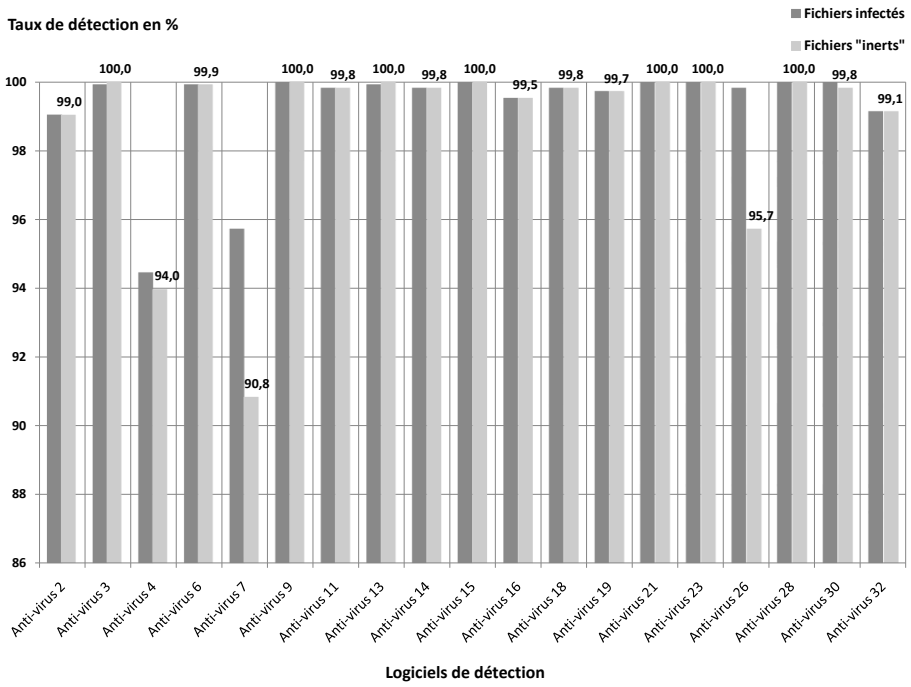


FIGURE 2.6 – Évolution des taux de détection pour les 19 anti-virus restant confrontés aux fichiers infectés originaux (en clair) et inertes (en foncé).

Tous ces anti-virus conservent un taux de détection supérieur à 90%. Les résultats obtenus pour les fichiers originaux et inertes sont sensiblement équivalents (moins de 0,5% d'écart) sauf pour les anti-virus 7 et 26. En effet, ces deux anti-virus perdent environ 5% de taux de détection avec la suppression du corps viral. Les raisons précises de cette diminution du taux de détection n'ont pas été étudiées. Nous concluons simplement cette expérience en remarquant que la détection porte sur la première partie du virus (« le chargeur »).

6. Il s'agit là de virus bénins au sens d'Adleman (voir la définition 4 de la section 1.1.2.1).

2.3.4 Troisième expérience : obscurcissement de code

Dans cette section 2.3.4, nous supposons que les techniques d'émulation de code permettent de faire abstraction de celles d'obscurcissement de code. En d'autres termes, nous supposons que si un outil de détection est capable d'émuler un code viral, alors ce même outil devrait aussi pouvoir détecter ce virus même en cas d'obscurcissement de code. Notre processus expérimental est représenté en figure 2.7 :

1. pour chaque fichier infecté, le « chargeur » est extrait ;
2. le binaire extrait est ensuite désassemblé afin d'obtenir le source assembleur correspondant ;
3. le source est obscurci en fonction de l'expérience conduite ;
4. le source obscurci est assemblé pour obtenir un nouveau binaire ;
5. le nouveau binaire est finalement injecté dans une copie du fichier infecté original.

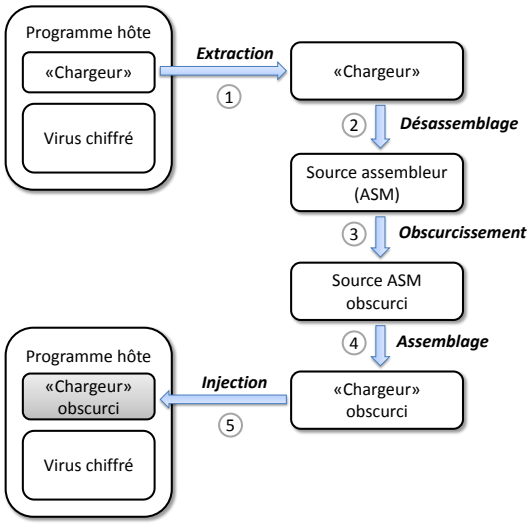


FIGURE 2.7 – Processus d'obscurcissement de code.

Ce processus a été appliqué à un échantillon de 107 fichiers infectés pour lesquels le taux de détection initial était de 100% avec les anti-virus sélectionnés. Nous nous focalisons maintenant sur l'étape 3 qui consiste à obscurcir le code :

1. le source assembleur du virus est décomposé en k blocs contenant chacun un nombre aléatoire d'instructions (ici de 2 à 6 instructions).
2. ces blocs sont permutés aléatoirement ;
3. à la fin de chaque bloc, une indirection vers le bloc suivant est dynamiquement calculée afin de conserver la même exécution.

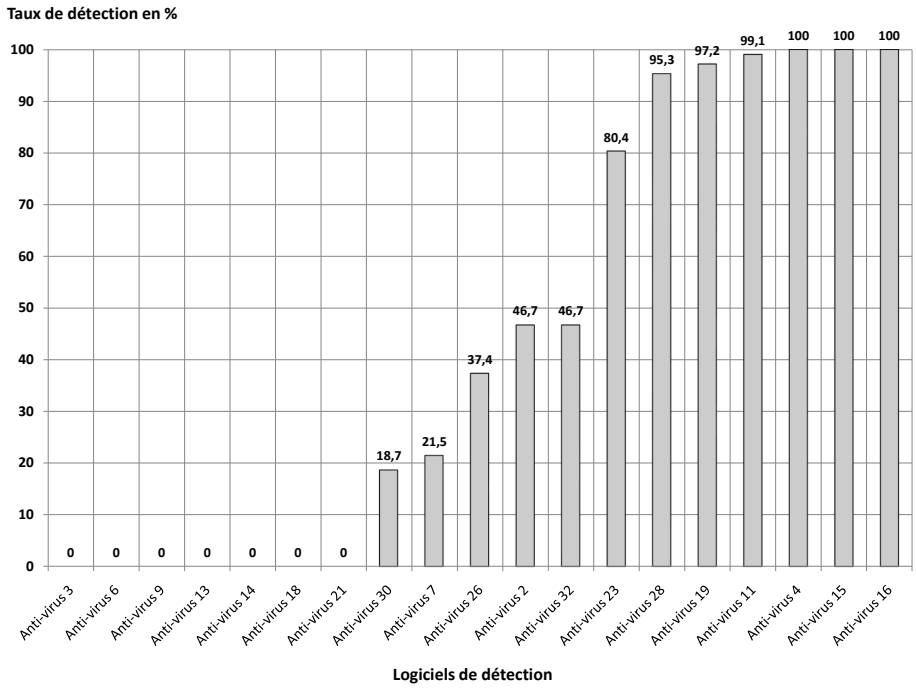


FIGURE 2.8 – Taux de détection des fichiers infectés après l'étape d'obscurcissement de code pour les 19 anti-virus restant.

Les 107 fichiers ainsi obscurcis sont soumis à détection. Les taux de détection sont donnés par ordre croissant sur la figure 2.8 :

- 7 programmes apparaissent incapables de détecter le moindre fichier infecté ;
- 5 programmes peuvent détecter des fichiers infectés avec des taux de détection inférieurs à 50%. Les taux de détection correspondants sont alors inférieurs à ceux d'un détecteur aléatoire pour ce type de virus ;
- 1 programme (Anti-virus 23) capable de détecter 80,4% des fichiers infectés. Ce taux de détection, inférieur à notre seuil de 90%, n'a pas fait l'objet d'une étude plus approfondie. Nous pouvons toutefois supposer que la détection s'appuie sur l'utilisation d'heuristiques particulières telles que celles présentées dans [158, chapitre 11]. Des expériences plus poussées seraient nécessaires afin de déterminer précisément comment se fait la détection de ces codes métamorphes ;
- 6 programmes détectent plus de 95% des fichiers infectés.

À partir de cette expérience, nous pouvons en conclure que 13 programmes n'utilisent pas d'émulation afin de détecter ce type de virus. Malheureusement, nous ne pouvons pas tirer directement de conclusion pour les 6 programmes restant qui présentent un taux de détection supérieur à 90%. En effet, une détection

fiable d'un tel code métamorphe n'implique pas nécessairement de l'émulation de code. Des heuristiques spécifiques à ce virus pourraient être employées afin de le détecter. C'est précisément la vérification du type d'analyse effectuée que nous étudions dans l'expérience suivante.

2.3.5 Quatrième expérience : analyse statique ?

Afin de déterminer si les 6 anti-virus restants émulent le code à détecter, nous modifions l'étape d'obscurcissement de code (3) de la figure 2.7. Le principe utilisé consiste à rendre les fichiers inertes, de sorte qu'une émulation de leur code ne devrait plus les détecter comme malveillants. Plus précisément, nous introduisons deux sous-expériences :

1. dans une première sous-expérience, nous modifions l'étape d'obscurcissement de code afin d'introduire une indirection invalide à la fin de chaque bloc du chargeur. Cette indirection provoque une exception qui stoppe l'exécution du programme avant toute action malveillante. En effet, dans notre processus d'obscurcissement de code, seules 2 à 6 instructions d'origine ont pu s'exécuter correctement. Aussi, en cas d'émulation du code, un tel fichier ne devrait donc pas être détecté ;
2. dans une seconde sous-expérience, nous remplaçons l'obscurcissement de code par du τ-obscurcissement de code [13]. Cette étape nous garantit que la première partie du virus ne peut être analysée (et même exécutée) avant une constante de temps τ. Cette constante τ a été fixée ici à 10 minutes. Dans ce cas, toute détection positive avant cette constante de temps implique que la détection ne s'appuie pas sur l'émulation du code.

Tous les résultats concernant l'obscurcissement de code sont donnés en figure 2.9 qui présente les taux de détection des outils restants pour l'insertion d'indirections, de « faux sauts » et du τ-obscurcissement de code. Avant d'interpréter cette figure, nous notons que **tous les anti-virus répondent en moins d'une minute**, et ce, quel que soit le fichier fourni dans ces expériences. En d'autres termes, aucun outil n'émule l'intégralité du « chargeur », tâche qui nécessiterait 10 minutes.

Cette dernière série d'expériences nous permet de conclure sur la validité de notre hypothèse d'analyse statique en mettant en avant deux types de résultats :

- 5 programmes sur les 6 testés détectent toutes les versions des fichiers infectés, quelle que soit la transformation utilisée, même en cas d'insertion de sauts arbitraires qui provoquent pourtant une erreur d'exécution fatale pour le programme. Les résultats positifs de détection obtenus par ces cinq anti-virus dans le cas d'insertion de sauts arbitraires s'interprètent alors comme des faux positifs. Ces détecteurs ne sont donc pas *pertinents* ;
- 1 programme (anti-virus 28) présente un résultat original qui laisse à penser qu'il est capable d'émuler les codes viraux à fin de détection. En effet, un taux de détection nul concernant l'insertion de « faux sauts » semble montrer la réelle émulation des codes soumis à détection. De manière similaire, les résultats négatifs de détection concernant le τ-obscurcissement

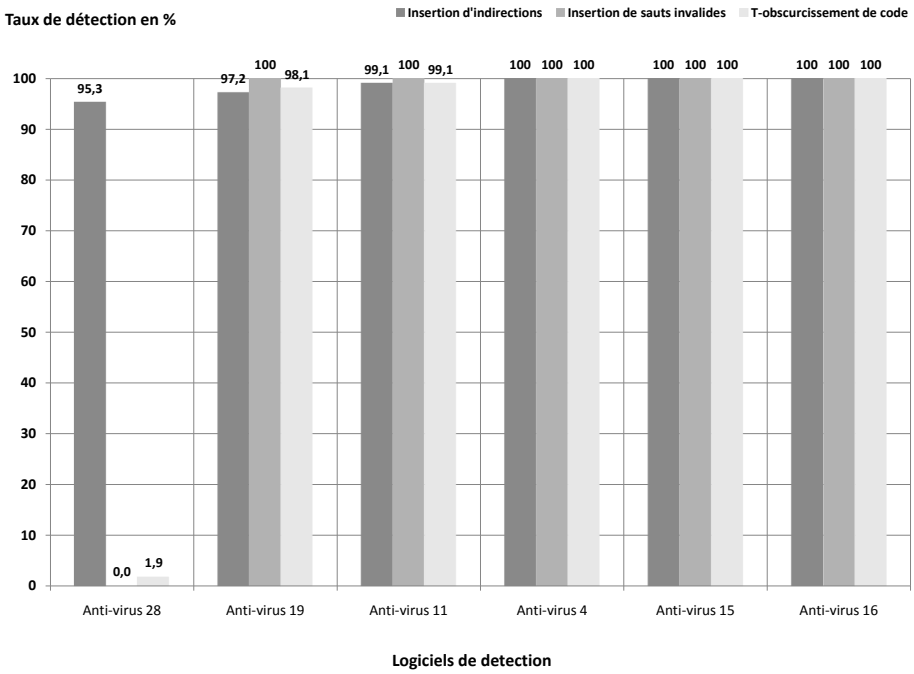


FIGURE 2.9 – Taux de détection des fichiers obscurcis pour les 6 anti-virus restants (indirection en gris clair, « faux sauts » au milieu et τ-obscurcissement de code en gris foncé.

de code pourraient s'expliquer par la nécessité d'un temps trop long (10 minutes) pour une réponse. Ceci explique les faux négatifs observés pour cet anti-virus qui n'est donc pas *fiable*.

Nous concluons donc ces tests menés sur 32 anti-virus avec pour résultat qu'un seul d'entre eux analyse réellement le virus métamorphe testé de manière dynamique afin de le détecter.

2.4 Bilan de notre approche d'obscurcissement de code

Dans ce chapitre, nous nous sommes intéressés à l'étape de mutation de code intervenant dans le processus d'auto-reproduction d'un programme métamorphe. À ce titre, nous avons proposé une approche d'obscurcissement de code s'appuyant sur un modèle théorique permettant d'en prouver la résilience dans le cadre de l'analyse statique. Ainsi, nous avons démontré l'existence de codes métamorphes pour lesquels la détection est prouvée NP-complète. Ce résultat théorique est obtenu dans le cadre d'une détection à la fois *fiable* et *per-*

tinente. En d'autres termes, nous nous plaçons dans le cadre où tous les codes métamorphes présentés, et uniquement ces codes-ci, doivent effectivement être détectés.

Notre résultat de NP-complétude est toutefois obtenu sous une hypothèse forte de l'analyse statique selon laquelle tous les chemins d'un programme sont supposés être potentiellement exécutables. Cette hypothèse traduit la difficulté de détermination des branchements conditionnels et notamment en présence de prédicats opaques. Dans ce cas, les deux chemins possibles du branchement sont supposés exécutables. Afin de valider cette hypothèse d'analyse statique, nous avons menés plusieurs expériences portant sur la détection d'un virus métamorphe. Ce virus a été transformé avant d'être soumis à 32 outils de détection « grand public » utilisés actuellement. Il en résulte que l'hypothèse d'analyse statique pour le virus considéré est validée pour 31 des 32 anti-virus testés.

Ces expériences constituent une première validation qu'il serait intéressant de compléter au moyen d'autres codes malveillants métamorphes afin de confirmer les résultats obtenus concernant l'hypothèse d'analyse statique dans le cadre de la détection des menaces métamorphes.

Suite aux expériences menées, nous avons mis en évidence l'impact du facteur temporel de l'obscurcissement de code sur les résultats de détection fourni. Ainsi, en sélectionnant convenablement la constante de τ-obscurcissement de code, il apparait possible de contourner non seulement la plupart outils d'analyse statique (27/32) mais aussi ceux de détection dynamique (anti-virus 28). Ce constat sera exploité dans le prochain chapitre qui aborde le deuxième point essentiel du cycle de reproduction des codes métamorphes : la manière dont un programme employant notre approche d'obscurcissement de code peut extraire son propre *archétype* sans pour autant faciliter sa détection par un outil spécifique.

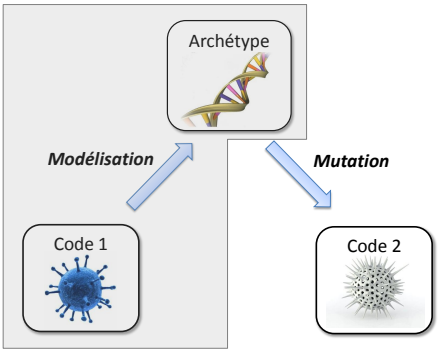
Les travaux présentés dans ce chapitre ont fait l'objet des publications suivantes :

- Jean-Marie Borello and Ludovic Mé. Techniques d'obscurcissement de code pour virus métamorphes. *2nd International Workshop on the Theory of Computer Viruses (TCV)*. May 2007.
- Jean-Marie Borello and Ludovic Mé. Code Obfuscation Techniques for Metamorphic Viruses. *Journal of Computer Virology*,4(3) :211-220. Springer. 2008.

Chapitre 3

Implémentation et évaluation d'un moteur de métamorphisme : application à la classification des outils de détection

Ce chapitre présente le fonctionnement de notre moteur de métamorphisme ainsi que son évaluation. Nous reprenons pour cela le schéma simplifié de réplication d'un code métamorphe de la figure 1.6 dont la partie étudiée ici, à savoir modélisation du code, apparait grisée.



L'objectif consiste, dans un premier temps, à présenter l'implémentation du processus d'auto-réplication de notre moteur de métamorphisme et principalement comment se fait l'extraction de son propre *archétype*, puis, dans un deuxième temps, à valider l'efficacité de ce moteur sur une souche malveillante connue. Cette évaluation, effectuée au moyen d'un ensemble de logiciels anti-

viraux représentatifs, permet non seulement de montrer l'efficacité de notre moteur, mais aussi de classer les différentes techniques de détection employées par ces logiciels anti-viraux. À ce titre, ce chapitre présente une contribution en termes de méthodologie pour l'évaluation « boîte noire » des outils de détection de codes malveillants.

L'organisation de ce chapitre est la suivante : la section 3.1 décrit le fonctionnement d'un moteur générique de métamorphisme que nous avons développé. Ensuite, la section 3.2 propose une application directe de ce moteur au ver MyDoom.A pour observer les techniques de détection employées par un panel représentatif des anti-virus actuels.

3.1 Implémentation d'un moteur de métamorphisme

Cette section décrit les choix d'implémentation ainsi que le fonctionnement global de notre moteur de métamorphisme. L'objectif de ce moteur est de fournir une API offrant à un programme quelconque, écrit en langage C, une fonctionnalité de réplication métamorphe. Il est important de remarquer qu'il s'agit précisément d'un moteur de métamorphisme et non pas d'un moteur métamorphe : c'est la totalité du programme dans lequel est intégré ce moteur qui mute lors de chaque réplication. D'un point de vue implémentation, ce moteur exporte trois fonctions : une fonction de réplication et deux fonctions permettant le chiffrement et le déchiffrement des données.

La présentation de notre moteur de métamorphisme se décompose en 4 étapes. La section 3.1.1 expose les choix d'implémentation du modèle d'obscurcissement de code présenté au chapitre précédent. La section 3.1.2 décrit de quelle manière un tel code métamorphe peut remonter à son *archétype*. La section 3.1.3 résume le processus complet de réplication d'un code métamorphe en faisant logiquement le lien entre les deux sections précédentes. La section 3.1.4 détaille l'obtention de la première souche métamorphe à partir du moteur et des sources d'un programme C.

3.1.1 Étape d'obscurcissement de code

Nous avons montré, au chapitre précédent, une preuve de résilience de notre approche d'obscurcissement de code. Cette preuve repose sur la difficulté de la détermination précise des alias en présence d'indirections dans le cadre de l'analyse statique et sous l'hypothèse multi-chemins. Ces alias assurent les transitions entre les différents blocs de code afin de garantir la même exécution que le programme d'origine. L'hypothèse multi-chemins traduit la difficulté de la détermination des conditions de branchements dans un programme et notamment en cas d'utilisation de prédicats opaques. Comme souligné en section 1.2.4.1, la génération de prédicats opaques efficaces représente un enjeu majeur en obscurcissement de code. Dans notre cas précis, nous optons pour une autre approche qui consiste à tirer avantage de la différence de contraintes temporelles

entre l'exécution d'un code malveillant et celle d'un outil de détection. En effet, comme souligné dans [64], même si ces deux catégories de programmes sont soumis aux mêmes limitations d'un point de vue théorique, le constat est tout autre en ce qui concerne leurs contraintes en termes d'exécution : la détection d'un code malveillant doit se faire le plus rapidement possible pour être acceptable du point de vue de l'utilisateur final, alors que ce même code peut considérablement allonger son temps d'exécution afin de mener son action malveillante. Tirant profit de ce constat, nous utilisons le τ-obscurcissement de code pour assurer les transitions inter-blocs.

Choix 3 (τ-obscurcissement).

Tirant profit de la dissymétrie entre le temps d'exécution possible pour un code malveillant et celui contraint pour sa détection, nous employons du τ-obscurcissement de code pour assurer les transitions inter-blocs du k-obscurcissement de code.

Plusieurs approches de τ-obscurcissement de code ont été présentées dans [13] dans le cadre de la protection logicielle. De manière simplifiée, l'approche utilisée ici consiste à calculer dynamiquement l'adresse du prochain bloc de code à exécuter au moyen d'une boucle obscurcie. Ce choix nous permet d'évaluer le temps d'obscurcissement de code en fonction du nombre d'itérations nécessaires au calcul final. **Nous ne donnons pas la description complète et précise de notre implémentation** pour deux raisons :

1. d'une part, nous considérons que d'un point de vue éthique la description complète des algorithmes utilisés permettrait à un développeur mal intentionné de produire un moteur métamorphe directement opérationnel, chose que nous ne pouvons nous permettre ;
2. d'autre part, les résultats qui suivent montrent que le τ-obscurcissement de code n'a pas d'impact visible sur les résultats de détection comme nous le verrons plus loin.

Nous illustrons tout de même le fonctionnement simplifié du τ-obscurcissement de code employé, au moyen du prédicat opaque suivant :

$$\sum_{i=1, i\neq 0[2]}^{2n-1} i = n^2. \tag{3.1}$$

Ce prédicat simple permet de comprendre les méchanismes mis en jeu dans le calcul dynamique des transitions même si **l'implémentation réellement employée est plus complexe.**

Le prédicat 3.1 peut s'interpréter comme une formule « opaque » permettant de calculer la fonction $n \mapsto n^2$ sur $\mathbb{N}$. Nous utilisons alors cet formule pour obscurcir l'adresse de destination du prochain bloc à exécuter. Pour cela, soit T l'adresse de destination du bloc que l'on cherche à atteindre dynamiquement. On se donne alors N, le nombre d'itérations souhaité. En notant $D = T - N^2$,

on obtient :

$$T = D + \sum_{i=1, i \neq 0[2]}^{2N-1} i.$$

Une description algorithmique en pseudo langage C peut être :

```
int Sum = D;
int i = 1;
do{
 Sum = Sum + i;
 i = i + 2;
} while( i <= 2N-1 );
goto Sum;
```

Pour la suite, nous utiliseront l'implémentation en assembleur x86 de cette fonction donnée par le programme 3.1.

```
     mov  ecx,N
     mov  esi,D          // int Sum = D;
     xor  edx,edx
     inc  edx            // int i = 1;
DoLoop:                  // do{
     lea  esi,[esi+edx]  // Sum = Sum + i;
     lea  edx,[edx+2]    // i = i + 2;

     lea  ebx,[ecx*2-1]  //
     cmp  edx,ebx        //
     jbe  DoLoop         // } while (i <= 2N-1);

     jmp  esi            // goto Sum;
```

Listing 3.1 – Exemple de code assembleur x86 de τ-obscurcissement simple d'une adresse de destination.

Déterminer la destination du saut final peut se faire de deux façons :

- soit en émulant l'algorithme implémenté, ce qui nécessite alors N itérations ;
- soit en identifiant, à partir du code fourni, le prédicat opaque. Dans ce cas, il suffit de remplacer le calcul itératif de la somme par le calcul direct $T = D + N^2$ pour obtenir l'adresse finale. Toutefois, il faut être capable d'extraire la formule arithmétique implémentée dans le code fourni. Nous développons cette remarque dans la section suivante.

Cette approche comporte cependant une restriction puisque le contexte d'exécution doit être préservé. En effet, le calcul des transitions doit être transparent, en cela qu'il ne doit pas modifier l'exécution du reste du code. Le programme 3.1 doit donc utiliser des registres libres ou alors les sauvegarder pour pouvoir les restaurer après utilisation (les registres `ecx`, `esi`, `edx`, `ebx` ont été modifiés, ainsi que le registre `EFLAGS` via l'instruction `cmp edx,ebx`).

3.1.2 Étape de modélisation : retour à l'archétype

Cette section présente l'étape d'abstraction permettant à un code métamorphe d'obtenir son *archétype* sur lequel il pourra appliquer les transformations présentées. À partir de maintenant, nous considérons que le moteur de métamorphisme M est déjà obscurci. Ce moteur obscurci sera désigné par M' pour le reste de la section. À partir de son point d'entrée, M' doit être capable d'extraire sa structure en mémoire afin de pouvoir se ré-obscurcir. Sans informations spécifiques sur la manière dont il a été produit, M' devrait alors se désassembler, comme tout programme extérieur devrait le faire. Dans ce cas, M' serait donc soumis aux difficultés d'inversion de ses propres transformations. Aussi, afin de pouvoir facilement analyser son propre code, M' doit donc incorporer des informations supplémentaires lui permettant de s'affranchir des transformations d'obscurcissement de code appliquées pour ainsi remonter à son code d'origine.

Conformément aux transformations présentées, revenir à M signifie retrouver la séquence originale de blocs de code $(M_1, \ldots, M_n)$. Les informations supplémentaires nécessaires sont les suivantes :

1. la description de la séquence originale des blocs de codes $(M_1, \ldots, M_n)$ ainsi que leurs tailles respectives ;
2. la description des blocs de données avec leur clé de chiffrement associée ;
3. la description des adresses absolues.

Nous illustrons maintenant en quoi la donnée de ces trois éléments permet au moteur de métamorphisme M' d'obtenir son *archétype* M.

3.1.2.1 La descriptions des blocs initiaux

La première information nécessaire pour la reconstruction de l'*archétype* est la séquence des blocs de codes originaux $(M_1, \ldots, M_n)$. Afin de simplifier notre explication, nous utilisons le programme 3.2 en tant qu'illustration. Ce code représente un bloc obscurci M'_i reprenant le programme 3.1 avec en plus les instructions du bloc M_i représentées en gras.

Supposons que M' ne dispose pas de la séquence d'origine des blocs de code. Dans ce cas, M' peut désassembler le bloc M'_i jusqu'à l'instruction `jump esi` de la ligne 14 pour laquelle l'adresse de destination contenue dans le registre `esi` est inconnue [1]. Maintenant, si M' dispose de cette adresse de destination, il peut donc déterminer le bloc suivant (M'_{i+1}) et ainsi de suite jusqu'au désassemblage complet de sont propre code. Toutefois, un problème persiste puisque le code correspondant aux transitions, c'est-à-dire celui du programme 3.1 par exemple, reste inclus dans le code de M'. En effet, le code du bloc M_i en cours de désassemblage est représenté en gras dans le bloc M'_i du programme 3.2. Tout le code restant calcule l'adresse du prochain bloc M'_{i+1}. C'est pourquoi, chaque bloc M_i

1. On se place dans le cas où le désassembleur n'est pas capable d'identifier le prédicat opaque utilisé et ne procède pas aux N itérations du programme 3.2.

```
1       mov  ecx,N
2       mov  esi,D
3       xor  edx,edx    // start of block M_i
4       push edx
5       push 01F0FFFh
6       push [esp+08h]  // end of block M_i
7       inc  edx
8   DoLoop:
9       lea  esi,[esi+edx]
10      lea  edx,[edx+2]
11      lea  ebx,[ecx*2-1]
12      cmp  edx,ebx
13      jbe  DoLoop
14      jmp  esi
```

Listing 3.2 – Exemple de bloc obscurci M'_i une fois désassemblé

doit être entièrement défini pour M' par la donnée de sa première instruction ainsi que de sa taille.

Il est important de noter que la distinction entre le code d'origine et celui calculant l'adresse de destination est difficile. Ainsi, sur l'exemple donné, on remarquera l'instruction `xor edx,edx` est utilisée à la fois par M_i et pour le calcul de l'adresse de M_{i+1}.

3.1.2.2 La descriptions des blocs de données chiffrées

En plus de la fonctionnalité de réplication exportée par notre moteur de métamorphisme, deux autres fonctions sont aussi fournies : une routine de protection de bloc de données et une autre de déprotection. Ces deux fonctions sont `AcquireData`, pour déchiffrer les données avant leur accès, et `RelaseData` pour rechiffrer ces données après utilisation. La description des blocs de données chiffrées est nécessaire pour permettre au moteur de métamorphisme de modifier les clés de chiffrement employées, c'est-à-dire, déchiffrer le bloc initial avec l'ancienne clé, puis générer une nouvelle clé aléatoire, et enfin rechiffrer ce bloc avec la nouvelle clé. Nous avons implémenté un chiffrement par masque jetable (« *One-Time-Pad* » ou OTP) [168].

3.1.2.3 La descriptions des adresses absolues

Au niveau du binaire d'un programme, chaque élément logique que ce soit un bloc de donnée, une instruction, une entrée dans la table des fonctions importées (« *Import Address Table* » ou IAT) est représentée par son adresse. En d'autres termes, ce typage de base est perdu durant le processus de compilation ou d'assemblage puisque seules les adresses persisent. Comme ces adresses changent à chaque mutation, M' doit être capable de localiser et de mettre à jour ces références en fonction de leurs nouvelles positions. Malheureusement, la détermination de ces références dans un programme binaire est difficile.

Afin d'illustrer ce problème, considérons l'instruction assembleur suivante : `cmp eax, 402000h`. Cette instruction compare la valeur contenue dans le registre `eax` avec la valeur hexadécimal `402000`. Considérant le moteur de métamorphisme (ou n'importe quel désassembleur), le problème consiste à déterminer le type de cette valeur. En d'autres termes, s'agit-il ou non d'une adresse ?

Maintenant considérons les deux programmes suivants décrits en langage C en figure 3.1 : les deux déclarent une valeur constante `MY_FLAG` en première ligne et une chaine de caractères en variable globale nommée `Global1` en ligne 2. La fonction `main` déclare simplement une variable en ligne 6, dont la valeur est supposée être définie plus tard dans la fonction `main`. Le point fondamental est l'instruction `if` en ligne 8 qui compare `var1` avec `MY_FLAG` du premier source avec `Global1` du second source.

```
1 #define MY_FLAG 0x402000
2 char    Global1[]="string";
3
4 main()
5 {
6     int var1;
7     ...
8     if (var1==MY_FLAG) {...}
9 }
```

```
1 #define MY_FLAG 0x402000
2 char    Global1[]="string";
3
4 main()
5 {
6     char* var1;
7     ...
8     if (var1==Global1) {...}
9 }
```

FIGURE 3.1 – Exemple illustrant la difficulté de la détermination des références.

Considérons le cas particulier où le processus de compilation place la chaîne `Global1` à l'adresse `402000` dans les deux binaires produits, la ligne 8 correspond à l'instruction assembleur précédente. Il est utile de remarquer que ce cas précis, bien que peu probable, peut se produire sous d'autres formes et permet de clairement illustrer le problème de la détermination des références mémoire. Concernant notre approche, le code ainsi que la donnée sont dispersés de manière aléatoire dans le programme lors de chaque réplication. Aussi, en considérant l'exemple précédent, l'adresse de `Global1` sera différente après réplication. Ainsi, pour être correct, l'instruction `cmp eax, 402000h` devra être mise a jour en remplaçant la valeur hexadécimale par la nouvelle adresse de `Global1` seulement dans le binaire du second programme.

3.1.3 Réplication du moteur sans forme constante

À ce stade, nous avons illustré :

1. comment obscurcir un programme pour garantir qu'il ne peut être désassemblé en dessous d'un certain temps τ ;
2. quelles sont les informations requises afin de créer un programme capable d'inverser ses propres transformations d'obscurcissement de code.

Il nous reste alors à expliquer pourquoi l'ajout d'informations supplémentaires permettant à M' d'extraire son *archétype* ne facilite pas pour autant sa

détection. Pour cela, nous décrivons maintenant comment le moteur de métamorphisme peut lier ces deux étapes afin d'achever sa réplication. La figure 3.2 illustre ce processus d'auto-reproduction. Pour des raisons de simplicité, nous présentons uniquement de quelle manière la description de la séquence de blocs de codes originaux est utilisée. L'utilisation des autres descriptions suit le même procédé.

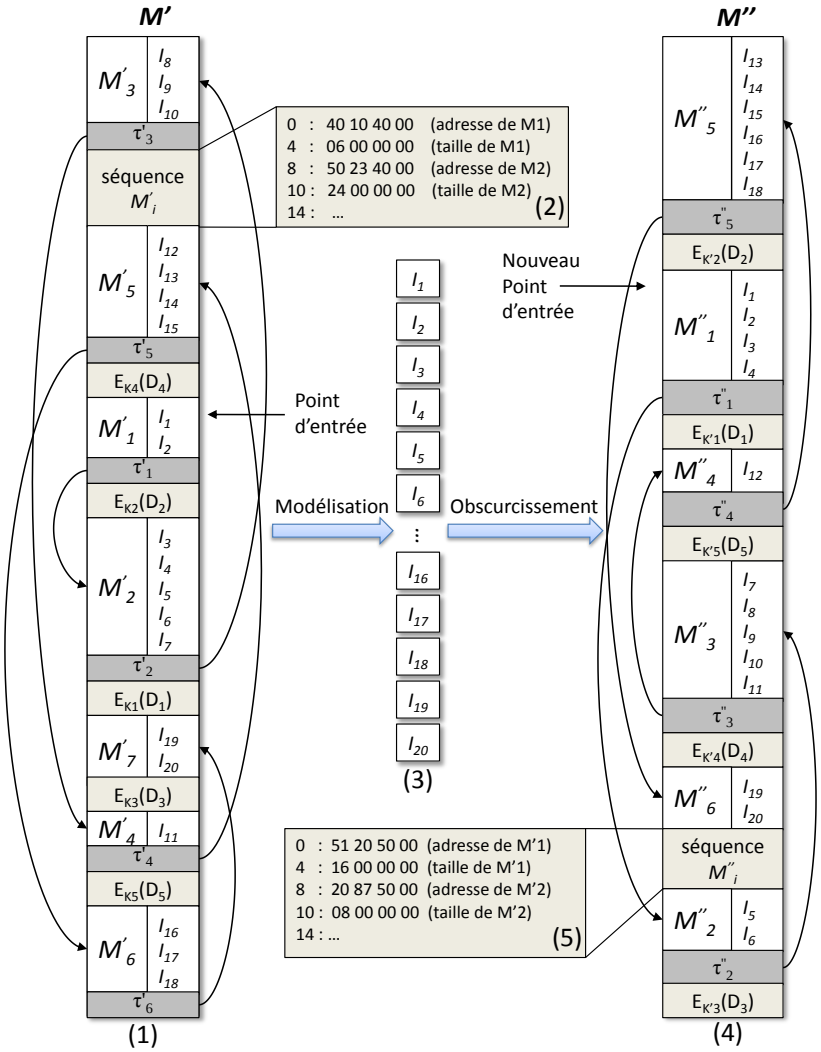


FIGURE 3.2 – Illustration du processus de réplication du moteur de métamorphisme.

Soit M' une version déjà obscurcie du moteur de métamorphisme M. M' comprend dans son binaire ses propres informations lui permettant de se recons-

truire. Plus précisément, M' est ici composé de 20 instructions notées $(I_1, \ldots, I_{20})$ distribuées en 7 blocs $(M'_1, \ldots, M'_7)$ comme représenté en (1). Chaque index d'instruction représente son ordre d'exécution. Chaque bloc M'_i contient un nombre d'instructions et une position aléatoire dans le programme M'. À la fin de chaque bloc, un autre noté τ'_i représente le branchement τ-obscurci dont la destination est fléchée. Comme nous l'avons préalablement indiqué, nous nous plaçons dans l'hypothèse où la destination de chacun de ces blocs ne peut être déterminée en dessous d'une constante de temps τ'_i.

Sans perte de généralité, supposons maintenant que les informations de reconstruction sont utilisées par l'instruction I_4 pour initier l'étape de modélisation. Cette instruction référence alors le premier bloc de description représenté en (2). Cette description donne la position ainsi que la taille de chaque bloc M'_i. Aussi, M' peut donc désassembler M'_1, puis M'_2, ainsi de suite jusqu'au dernier bloc M'_7. À partir de maintenant, M' possède une représentation abstraite de sa propre séquence d'instructions $(I_1, \ldots, I_{20})$ en mémoire comme illustré en (3). Il est important de souligner que cette représentation est plus complexe que la séquence initiale des instructions qui constituerait un motif de détection simple. Il s'agit d'une liste chaînée de structures décrivant chacune une instruction. L'étape de ré-obscurcissement de code, dont le résultat est illustré en (4), commence ici : les nouveaux blocs de code sont générés de manière aléatoire $(M''_1, \ldots, M'_6)$ avec leurs transition τ-obscurcies $(\tau''_1, \ldots, \tau''_6)$. La séquence originale des blocs de code $(M''_1, \ldots, M'_6)$ est insérée comme nouveau bloc de données représenté en (5). Le point clé consiste à mettre à jour la référence à cette information de reconstruction dans l'instruction I_4, pour être sûr que cette instruction utilisera la nouvelle description des blocs de code. Finalement, le point d'entrée de M'' est défini dans son en-tête afin de renseigner la position de la première instruction I_1 à exécuter dans M''.

Du point de vue de la détection, les informations de reconstruction ne présentent aucune donnée et position constante entre deux formes mutées. Ainsi, nous supposons que pour atteindre ces informations de reconstruction il faut être capable de désassembler un binaire obscurci jusqu'à identifier la portion de code utilisant ces informations. Dans ce cas, un quelconque désassembleur serait confronté à la robustesse des transformations d'obscurcissement de code appliquées. Aussi, la détection serait retardée de la durée τ.

3.1.4 Application du moteur de métamorphisme à un autre programme

Nous avons illustré jusqu'à présent comment le moteur de métamorphisme peut se reproduire grâce à ses informations de reconstruction. Cependant, une interrogation subsiste : comment ses informations ont elles été incorporées au premier binaire produit ? Pour apporter une réponse, nous revenons sur deux points importants. Tout d'abord, notre moteur de métamorphisme travaille au niveau du binaire afin de tirer avantage de la difficulté du désassemblage sur des architectures du type x86. Deuxièmement, l'objectif de ce moteur est d'être

générique pour pouvoir transformer un programme écrit en langage de haut niveau (autre qu'assembleur) pour rendre ce programme métamorphe. Nous prenons ici comme hypothèse, que le programme d'entrée est écrit en langage C. Ainsi, nous modifions le processus de compilation pour produire le premier binaire métamorphe comme s'il était issu d'une mutation. Ce résultat est atteint en insérant notre obscurcisseur de code dans la chaîne de compilation comme illustré en figure 3.3.

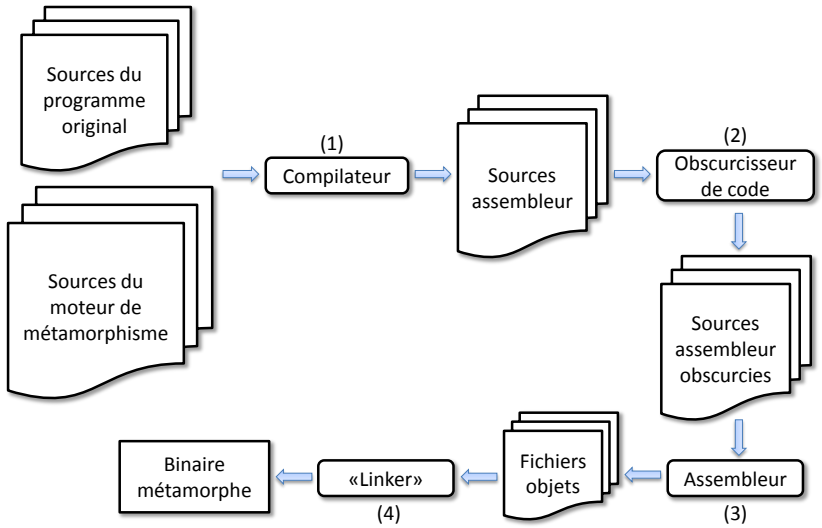


FIGURE 3.3 – Illustration de la production du premier binaire métamorphe à partir des sources du moteur de métamorphisme et de celles d'un programme donné.

La chaîne de compilation commence normalement en prenant en entrée deux programmes, le moteur de métamorphisme et le programme à obscurcir. Tout d'abord, le compilateur produit les sources assembleur correspondant. Ensuite, l'obscurcisseur de code transforme ces sources comme présenté en section 2.2. Les sources assembleur obscurcies contiennent maintenant toutes les informations de reconstruction pour le programme entier. Finalement, ces sources sont assemblées pour produire les fichiers objets correspondants qui seront liés à des bibliothèques additionnelles pour obtenir le premier binaire métamorphe. Cette étape finale est la même que celle d'un processus d'assemblage classique.

3.2 Résultats expérimentaux : classification des détecteurs de codes malveillants

Cette section 3.2 a pour but d'évaluer l'impact de notre moteur de métamorphisme sur les outils de détection actuels. L'objectif est double : d'une part

ces expériences nous permettent de valider l'efficacité et la généricité de notre moteur de métamorphisme. D'autre part, cela nous permet d'évaluer la *fiabilité* des outils de détection actuels. Les expériences menées ici complètent celles de Jacob *et al.* [66] dans lesquelles le ver MyDoom est manuellement modifié pour en produire différentes versions correspondant à du polymorphisme fonctionnel.

3.2.1 Construction d'une version métamorphe du ver MyDoom

Toutes nos expériences sont construites à partir du ver MyDoom.A [60] découvert en janvier 2004. Le choix de ce code malveillant a été guidé par deux raisons principales :

1. les sources de ce ver [2] sont disponibles en langage C, ce qui nous permet une utilisation directe de notre moteur de métamorphisme ;
2. MyDoom est toujours considéré comme l'une des plus sérieuses attaques connues [158], étant donné sa virulence, c'est-à-dire le nombre de mails générés pour assurer sa propagation.

De manière succincte, MyDoom est un ver se propageant via un client pair à pair (« *Peer to Peer* » ou P2P) et par emails. Sa charge utile est composée de deux parties :

- la première partie effectue un dénis de service (« *Denial of Service* » ou DoS) ciblant un site internet particulier ;
- la seconde partie est une bibliothèque dynamique (« *Dynamic Link Library* » ou DLL) chiffrée qui implémente une porte dérobée. Cette porte dérobée est en écoute sur un port TCP compris entre 3127 et 3198. Cette DLL peut être considérée comme un code malveillant à part entière. Elle est chargée par l'explorateur Windows (`explorer.exe`) pour accepter des commandes d'un attaquant extérieur.

Nous considérons par la suite avoir à notre disposition deux candidats à la détection : MyDoom et sa porte dérobée.

Dans les sources de MyDoom, la fonction en charge de la réplication est `CopyFile` qui, comme son nom l'indique, permet de copier un fichier vers une autre destination. Afin de rendre ce ver métamorphe, nous avons modifié les sources de MyDoom en remplaçant les appels à la fonction `CopyFile` par la fonction de réplication de notre moteur de métamorphisme. La porte dérobée ne constitue pas un code auto-reproducteur. C'est pourquoi le moteur de métamorphisme n'est pas appliqué à cette dernière.

La génération du ver métamorphe est illustrée en figure 3.4 :

La porte dérobée est obscurcie comme expliqué en section 3.1.1 afin d'obtenir la DLL `xproxy.dll` en étape (1). Ensuite, cette DLL est chiffrée comme pour la compilation d'origine du ver. Ce binaire chiffré est ensuite converti en tableau de valeurs hexadécimales dans le fichier source `xproxy.inc` comme le montre l'étape (2). Finalement, le ver métamorphe est produit comme l'illustre

2. les sources de MyDoom.A sont téléchargeable via le lien suivant `http://vx.org.ua/dl/src/mydoom.zip` (dernier accès en décembre 2010).

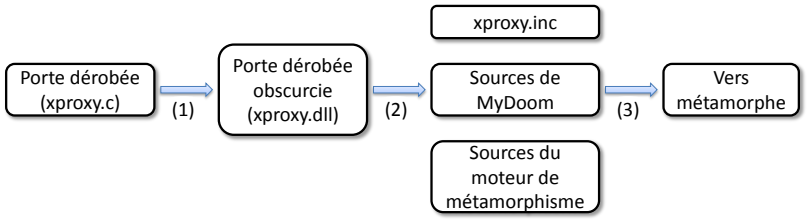


FIGURE 3.4 – Illustration de l'incorporation de la porte dérobée obscurcie (`xproxy`) dans le ver métamorphe MYDOOM.

la figure 3.4 à partir du fichier `xproxy.inc`, du moteur de métamorphisme et des sources de MYDOOM en étape (3).

3.2.2 Plateforme d'évaluation

Afin d'observer le comportement du code malveillant dans un environnement sûr et protégé, une plateforme dédiée est utilisée. La solution adoptée consiste à employer des machines virtuelles pour assurer la sécurité du système d'exploitation. Nous avons au préalable vérifié que MYDOOM ne tente pas de détecter la virtualisation de son environnement afin de modifier son comportement le cas échéant. En outre, l'emploi de machines virtuelles nous permet aussi de facilement revenir à un état sain, indépendamment du résultat de détection, en restaurant l'image du système d'origine.

Notre plateforme d'évaluation utilise VMWare workstation comme environnement de virtualisation. Elle est constituée de deux éléments représentés sur la figure 3.5 : la machine invitée et la machine hôte.

1. **La machine invitée**. Le système d'exploitation installé est Windows XP Pro SP3 à jour au niveau des correctifs. Afin d'observer la propagation du ver, un client e-mail ainsi qu'un client P2P sont installés et configurés. Un compte d'accès à internet est défini avec différents paramètres dont l'adresse d'un serveur SMTP. Cette configuration est clonée pour servir de base à chaque anti-virus testé. Un anti-virus différent est installé pour chacune de ces configurations. Cet anti-virus est paramétré afin d'obtenir les meilleurs résultats de détection possibles.
2. **La machine hôte**. Un pont a été configuré entre les deux machines afin d'établir une liaison réseau. Un faux serveur de système de noms de domaines (« *Domain Name System* » ou DNS) est installé pour rediriger tout le trafic réseau en provenance de la machine invitée vers la machine hôte. De même, un faux serveur SMTP, en écoute sur le port 25, est en charge de la collecte des e-mails de propagation en provenance du ver.

Dans le but d'être le plus représentatif possible, nos expériences sont menées avec 16 anti-virus [3] parmi les plus utilisés du marché indépendamment des

3. Les anti-virus utilisés sont les suivants : AntiVir personal edition 8.1.0.367, Avast ! 4.8 an-

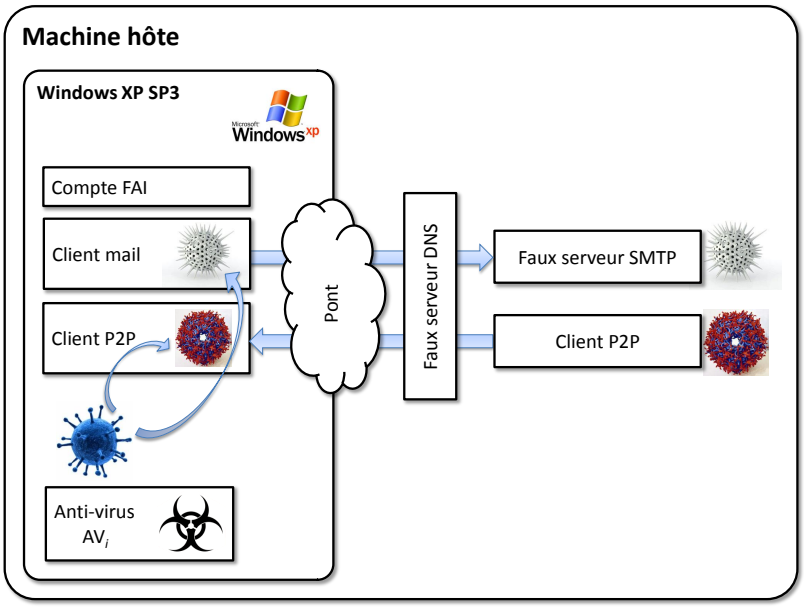


FIGURE 3.5 – Plateforme d'évaluation.

techniques de détection proposées. En termes de licence plusieurs anti-virus restreignent l'évaluation « boîte noire » : « *You shall not use this Software in automatic, semi-automatic or manual tools designed to create virus signature, virus detection routines, any other data or code for detecting malicious code or data* ». Dans un but de neutralité tous nos résultats seront alors donnés de manière anonyme sous la forme AV1 à AV16.

3.2.3 Protocole expérimental

Afin de valider le bon fonctionnement des vers obtenus et d'apporter des résultats représentatifs, nous présentons maintenant notre protocole expérimental. Deux types de vers sont en fait considérés : des vers obscurcis et des vers métamorphes. La distinction entre ces deux types de vers réside dans leur réplication. Les vers métamorphes emploient notre moteur de métamorphisme pour se propager alors que ceux obscurcis utilisent la fonction `xcopy` d'origine pour se répliquer. Ces derniers conservent donc une forme constante. L'objectif ici est d'identifier si une copie à l'identique d'un fichier exécutable (un « *Portable Executable* » (PE)) constitue un critère de détection ou non. Le protocole ex-

tivirus, BitDefender Antivirus 2009, DrWeb antivirus 4.44.1.11240, F-Prot antivirus, F-Secure Anti-Virus 2009, Kaspersky Anti-Virus 2009, McAfee SecurityCenter, Eset Nod32 Antivirus Version 3.0.669.0, Norman Security Suite X, Norton 360 Version 2.2.0.2, Panda Antivirus Pro 2009, Sophos Anti-Virus 7.6.2, Sunbelt 3.1.2416, PC Tools AntiVirus 5.0,Trend Micro Internet Security Pro Version 17.00.

périmental est alors le suivant :

1. un ver est installé sur une machine invitée qui contient un anti-virus. De fait, il y a autant de machines invitées qu'il y a d'anti-virus impliqués dans nos expériences (ici 16). Par défaut, le moteur de métamorphisme est configuré pour produire des blocs de code comprenant de 1 à 5 instructions et avec 1 seule itération pour le τ-obscurcissement de code [4]. Si le ver est détecté, le résultat est alors consigné et cette première étape est réitérée jusqu'à atteindre le nombre de tests souhaités (ici 100) ;
2. si le ver métamorphe installé n'est pas détecté, alors ce dernier est ensuite exécuté sur la machine invitée jusqu'à l'obtention de deux nouvelles formes mutées (s'il n'est pas détecté avant par l'anti-virus utilisé). Ces deux vers sont récupérés sur la machine hôte au moyen du client P2P, qui télécharge un nouvel exécutable, et du faux serveur SMTP, qui réceptionne le nouveau ver sous la forme d'un pièce jointe à un e-mail reçu ;
3. l'environnement virtuel est finalement restauré dans l'état initial et les expériences sont réitérées (étape 2) avec les 2 nouveaux codes malveillants obtenus jusqu'à ce que le nombre d'échantillons désiré soit atteint. Cette façon de procéder permet de valider le bon fonctionnement du ver et notamment sa réplication.

3.2.4 Résultats de détections obtenus

Les résultats de détection concernant les deux échantillons de codes malveillants soumis (obscurcis et métamorphes) sont présentés dans le tableau 3.1 ainsi que les techniques de détection qui sont déduites de l'observation des anti-virus. On remarquera que les expériences menées ici sont plus précises que celles conduites au chapitre 2 section 2.3. En effet, les résultats obtenus précédemment ne comportent pas d'avertissements liés à l'exécution des codes malveillants soumis, contrairement aux observations notées ici.

Au regard des résultats fournis par le tableau 3.1, quatre catégories de techniques de détection sont identifiées :

1. les logiciels analysant le comportement d'un programme, AV1 et AV2 ;
2. les logiciels bloquant certaines actions spécifiques, AV3 à AV8 ;
3. les outils de détection par heuristiques, AV9 ;
4. les anti-virus que nous qualifions de purement syntaxiques, incapables de détecter le moindre ver obscurci ou métamorphe, AV10 à AV16.

Nous notons toutefois que l'action de la porte dérobée, c'est-à-dire l'écoute sur un port TCP spécifique, est signalée dans tous les cas par le pare-feu du système. La sécurité repose alors sur le choix de l'utilisateur d'accepter ou non cette action.

4. Cette configuration est modifiée dans un seul cas, voir 3.2.4.3

1. Tous les vers, aussi bien obscurcis que métamorphes, ont été bloqués à cause de l'installation d'une DLL qualifiée de suspecte par AV2. Comme cette DLL représente un code malveillant en soit, c'est-à-dire la porte dérobée de MyDoom), Pour cet anti-virus, les résultats sont donnés pour des vers sans charge finale (porte dérobée `xproxy.dll`).

Outil	observations		techniques de détection déduites des observations
	vers obscurcis	vers métamorphes	
AV1	100/100 détectés comme « *generic Trojan* »	0/100	analyse comportementale
AV2	100/100 détectés pour des actions suspectes sur des fichiers (auto-copies)[1]	0/100[1]	analyse comportementale
AV3	40/100 bloqués pour actions suspectes sur des fichiers	40/100 bloqués pour actions suspectes sur des fichiers	blocage d'actions portant sur les fichiers
AV4 AV5 AV6	100/100 bloqués pour modifications du registre (résidence)	100/100 bloqués pour modifications dur registre (résidence)	blocage d'actions portant sur le registre
AV7	100/100 bloqués pour accès en écriture au répertoire système	100/100 bloqués pour accès en écriture au répertoire système	blocage d'actions portant sur les fichiers
AV8	100/100 bloqués pour actions suspectes	100/100 bloqués pour actions suspectes	blocage portant sur des actions suspectes
AV9	10/100 détectés comme « *Heur_PE virus* »	10/100 détectés en tant que « *Heur_PE virus* »	analyse de forme par heuristiques
AV10 ⋮ AV16	0/100 ⋮ 0/100	0/100 ⋮ 0/100	pas de détection ⋮ pas de détection

TABLE 3.1 – Résultats de détection obtenus par 16 produits anti-viraux sur 100 vers obscurcis (première colonne) et 100 vers métamorphes (deuxième colonne). Les techniques de détection déduites des observations des anti-virus sont fournies en troisième colonne.

Un résultat identique est obtenu pour toute application de type serveur, en écoute sur un port particulier. Aussi, ce comportement ne semble pas suffisant pour caractériser un code malveillant à lui seul.

Avant de détailler les résultats obtenus, nous remarquons qu'ils confirment ceux des travaux de Jacob *et al.* [89] portant sur la détection d'un moteur de polymorphisme fonctionnel. Nous présentons ci-après les trois premières catégories d'outils qui présentent des vrais positifs.

3.2.4.1 Résultats pour l'observation comportementale

Cette catégorie de détecteurs inclut deux logiciels (AV1 et AV2) capables de détecter tous les vers obscurcis mais pas les vers métamorphes. Ce résultat tend à illustrer que ces deux anti-virus considèrent l'auto-réplication comme une composante essentielle à la détection. Nos résultats montrent que la réplication directe via l'appel à la fonction `CopyFile` est détectée mais pas le processus de réplication du moteur de métamorphisme illustré en figure 3.2. AV1 ne fournit aucune information permettant de déceler la technique de détection précisément employée. Il semble que les évènements sensibles d'un point de vue de la sécurité (la création de fichiers, les modifications de fichiers et de clés de registre, l'auto-réplication, etc.) sont corrélés dans le but d'identifier une catégorie générique de codes malveillants (ici un cheval de Troie selon l'anti-virus). AV2 détecte tous les vers, qu'ils soient obscurcis ou métamorphes durant l'installation de la porte

dérobée. Si cette porte dérobée n'est pas incluse dans les vers testés, alors aucun ver métamorphe n'est détecté.

3.2.4.2 Résultats pour les bloqueurs comportementaux

Tous les logiciels de détection (AV3 à AV8) requièrent l'intervention de l'utilisateur pour chaque action « suspecte » détectée. L'anti-virus AV3 avertit l'utilisateur en cas de détection d'un fichier contenant un programme exécutable avec une extension différente. Cela arrive, avec une probabilité de 40%, lors de la création des mails assurant la propagation du ver. Cette probabilité est présente telle quelle dans le code source original du ver. Plus précisément, le ver compresse (au moyen du « *packer* » UPX) sa nouvelle image binaire dans un répertoire temporaire avec une extension `.tmp` avant de coder (en base64) cette copie qui sera envoyée comme pièce jointe du mail. Par conséquence, tous ces fichiers temporaires sont alors détectés comme suspects par AV3. Les outils AV4 jusqu'à AV6 bloquent toutes les tentatives de mise en résidence du ver par modifications du registre. AV7 bloque tous les accès en écriture au répertoire système. Finalement, le programme AV8 détecte plusieurs comportements risqués et produit les avertissements suivants pour tous les vers métamorphes :

1. modifie votre ordinateur afin qu'un autre ordinateur puisse y accéder ;
2. copie un fichier « exécutable » dans une zone sensible de votre système ;
3. s'enregistre lui-même dans votre liste d'exécution automatique (« *Windows System Startup* ») ;
4. copie un autre programme vers une zone de votre ordinateur qui partage des fichiers avec un autre ordinateur ;
5. se connecte à Internet d'une façon suspecte pour envoyer des mails.

Nous remarquons ici que AV8 n'est pas capable de détecter l'auto-réplication comme le suggèrent les termes « un exécutable » en 2 et « un autre programme » en ligne 4. Pour ce qui est des vers obscurcis, AV8 est capable de détecter l'auto-réplication. Dans tous les cas, un avertissement est généré pour chaque copie de programme ainsi que chaque auto-réplication. Ces différents avertissements ne sont pas corrélés afin d'identifier un comportement spécifique de code malveillant. Le blocage comportemental est une technique de détection pro-active visant à prévenir les actions malveillantes avant qu'elles ne soient exécutées. Chacune de ces actions repose sur un simple appel système. Aussi, le τ-obscurcissement de code apparaît alors inutile pour cette catégorie de détecteurs.

3.2.4.3 Résultats pour la détection heuristique

AV9 détecte tous les codes malveillants directement à partir de leur image binaire et non durant leur exécution. Plus précisément, tous les vers sont détectés sous l'appellation « *Heu_PE virus* », ce qui suggèrent l'emploi d'heuristiques pour la détection. Afin de valider cette hypothèse de détection par heuristiques, nous avons créé trois échantillons avec différentes valeurs pour τ (1, 500, puis 1 000 000 itérations). Chacun de ces échantillons est composé de quatre groupes

de 100 vers avec des tailles de blocs de code différentes. La figure 3.6 présente les taux de détection correspondant. Les résultats montrent que les taux de

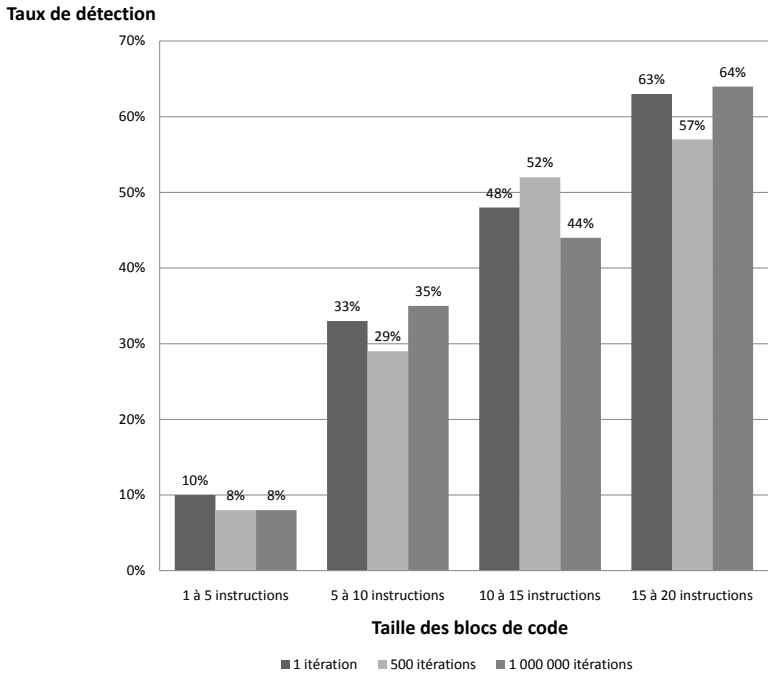


FIGURE 3.6 – Taux de détection de l'anti-virus AV9 en fonction de la taille des blocs de code et des valeurs de τ.

détection sont proportionnels à la taille des blocs de code. De plus, le τ-obscurcissement de code n'a pas d'impact sur la détection. Ces résultats confirment que le produit AV9 emploie des techniques de détection à base d'heuristiques pour reconnaître ces codes malveillants métamorphes. Aucune information supplémentaire n'est fournie par ce produit quant aux heuristiques utilisées.

3.3 Bilan des aspects implémentation et évaluation

Ce chapitre a exposé la contribution de notre moteur de métamorphisme à la méthodologie de classification « boîte noire » des logiciels anti-viraux. À partir du modèle d'obscurcissement de code présenté au chapitre 2, nous avons présenté le fonctionnement de notre moteur de métamorphisme. En particulier, nous avons illustré comment l'adjonction d'un tel moteur permet, à un programme transformé, de remonter à son *archétype* tout en garantissant la difficulté de sa détection sous certaines hypothèses d'analyse statique. La clas-

sification des techniques de détection observées, sur un panel représentatif de logiciels anti-viraux montre la difficulté actuelle de détection de tels programmes métamorphes. En effet, les expériences conduites mettent en avant deux techniques de détection comportementale utilisées par les anti-virus actuels : l'observation comportementale ainsi que le blocage comportemental. Deux anti-virus produisent des résultats intéressants, chacun représentant une technique de détection différente. Un premier produit est capable de corréler plusieurs actions suspectes afin d'identifier un comportement malveillant générique. Toutefois, l'emploi de mécanismes complexes de réplication tels que ceux impliqués dans le métamorphisme peuvent conduire à 100% de faux négatifs lorsque nous soumettons, par exemple, notre ver métamorphe expérimental. Un deuxième anti-virus peut détecter toutes les actions suspectes qui pourraient constituer une représentation du comportement du ver MyDoom. Cependant, ces actions ne sont pas corrélées afin d'identifier ce ver. Dans tous les cas, nos expériences montrent qu'il est trop tôt pour évaluer l'impact du τ-obscurcissement de code sur les outils de détection actuels.

Perspective 2 (Amélioration du moteur de métamorphisme).

Afin d'éviter le recours au τ-obscurcissement de code qui diffère l'exécution de notre programme métamorphe, nous envisageons l'emploie d'autres approches d'obscurcissement de code à fortes résiliences pour la conception d'un moteur de métamorphisme.

En termes de détection, une approche privilégiée pour s'abstraire des transformations syntaxiques utilisées est celle de l'analyse dynamique des programmes. C'est naturellement cet axe que nous adoptons, à partir de maintenant, pour aborder le problème de la détection des codes malveillants et notamment ceux métamorphes. Dans la partie II, nous envisageons l'utilisation de la complexité de Kolmogorov pour la détection des codes malveillants.

Les travaux exposés dans ce chapitre ont fait l'objet des publications suivantes :

- Jean-Marie Borello, Éric Filiol, Ludovic Mé. Are current antivirus programs able to detect complex metamorphic malware ? An empirical evaluation. *18th EICAR Annual Conference*. May 2009.
- Jean-Marie Borello, Éric Filiol, and Ludovic Mé. From the Design of a Generic Metamorphic Engine to a Black Box Classification of Antivirus Detection Techniques. *Journal of Computer Virology*,6(3) :277-287. Springer 2010.

Deuxième partie

Application de la complexité de Kolmogorov à la détection des codes malveillants

Chapitre 4

Mesures de similarités fondées sur la complexité de Kolmogorov

L'objectif de ce chapitre est d'introduire les fondements théoriques sur lesquels s'appuient notre approche de détection de codes malveillants. Pour cela, nous proposons d'utiliser comme socle formel la complexité de Kolmogorov [113] dont nous présentons ici uniquement les éléments essentiels. Succinctement, cette complexité correspond au programme de taille minimale, dans un langage donné, permettant de produire une sortie particulière. Partant de la complexité de Kolmogorov, nous introduisons ensuite les différentes métriques classiques qui ont permis d'aboutir à une première mesure de similarité, dénommée la distance normalisée de compression (« *Normalized Compression Distance* » ou NCD). De manière informelle, cette métrique mesure la distance entre deux objets en termes d'information commune, en exploitant le fait que deux données similaires, une fois concaténées, se compressent mieux que des données différentes. Cette distance (NCD) présente aujourd'hui un champ d'applications varié [41] et a notamment été utilisée avec succès pour la construction automatique d'arbres phylogénétiques sur des génomes complets de mitochondries [111, 112], la construction d'un arbre linguistique pour 50 dialectes eurasiatiques [112], la détection de plagiats dans des programmes d'étudiants [28] ainsi que la classification de musiques [40], etc.

Dans ce chapitre nous proposons une nouvelle mesure de similarité, dénommée degré d'inclusion mutuelle par compression (« *Compression based Mutual Inclusion Degree* » ou CMID), également obtenue par compression sans perte. Afin d'introduire brièvement l'origine de cette mesure, nous revenons sur une propriété de NCD. Comme nous le verrons plus loin, cette distance est en effet normalisée par la taille du plus grand élément compressé. De fait, NCD éloigne naturellement deux objets dont les tailles de compression respectives sont différentes. Or, il peut parfois apparaitre intéressant de mesurer, en termes d'information commune, si un objet se retrouve « inclus » dans un autre. C'est

par exemple le cas pour un comportement malveillant connu, que l'on observe au milieu de celui d'un autre programme supposé sein à l'origine. C'est d'un tel constat qu'est née notre mesure de similarité (CMID), qui représente effectivement cette notion intuitive d'inclusion. Dans ce chapitre, nous démontrons sous quelles conditions, portant sur l'algorithme de compression employé, cette mesure de similarité est effectivement un degré d'inclusion au sens formel du terme. Nous apportons aussi un exemple d'application de ce degré d'inclusion autre que celui de la détection des codes malveillants.

4.1 Les différentes définitions de l'information

Dans cette section, nous introduisons les fondements théoriques sur lesquels s'appuient les deux mesures de similarité évoquées (NCD et CMID). La notion de départ est celle d'information, bien trop vaste pour se résumer en une seule définition. Nous présentons ici deux manières d'envisager le concept d'information : d'une part une description probabiliste proposée par Shannon et présentée en section 4.1.1 ; d'autre par une approche algorithmique initiée par les travaux de Kolmogorov et présentée en section 4.1.2. Cette dernière approche, appelée complexité de Kolmogorov, est celle développée par la suite.

Dans tout le chapitre et sauf mention particulière, notre étude porte sur l'ensemble des chaînes binaires finies noté Ω ($\Omega = \{0,1\}^*$). La chaîne vide de Ω sera notée ϵ. Comme tout objet fini peut se représenter sous la forme d'une suite binaire finie, ce choix ne conditionne pas les résultats obtenus.

4.1.1 Théorie probabiliste de l'information : l'entropie de Shannon

Shannon s'est intéressé à la définition de la quantité d'information moyenne délivrée par une source, à destination d'un récepteur, via un canal de communication [153]. Son approche s'appuie sur les probabilités d'occurrences des évènements observés, c'est-à-dire des messages émis par la source. De manière formelle, on considère une variable aléatoire X pouvant prendre n valeurs distinctes $(X_i)_{i\in[\![1,n]\!]}$. Chacune de ces valeurs possède une probabilité d'occurrence notée p_i. L'entropie de la variable X est alors définie par Shannon comme :

$$\mathcal{H}(X) = -\sum_{i=1}^{n} p_i \log(p_i)$$

L'entropie $\mathcal{H}(X)$ de la variable X permet de mesurer la quantité d'information moyenne associée à un ensemble d'évènements (X_i). L'intérêt de l'entropie est de pouvoir caractériser la taille d'un codage optimal associé aux évènements observés. Il s'agit du théorème de codage de source obtenu par Shannon, que nous présentons en théorème 12. Afin d'introduire ce résultat fondamental, quelques définitions préalables sont nécessaires.

Définition 27. *(Fonction de codage [113, p.13]). Une fonction de codage est une application $h : \Omega \to \Omega$ qui à tout mot source associe le mot code correspondant.*

À titre d'exemple, on peut considérer une fonction de codage simple, qui à toute représentation ASCII d'un caractère alphabétique majuscule, associe son index binaire dans l'alphabet. Un tel codage est donné dans le tableau 4.1.

Caractère ASCII	**représentation binaire**	**index alphabétique**
A	01000001	0
B	01000010	1
C	01000011	10
⋮	⋮	⋮
Z	01011010	11001

TABLE 4.1 – Exemple de codage non uniquement décodable.

Un problème se pose alors au moment du décodage puisque plusieurs possibilités se présentent pour une même suite binaire. Par exemple, le codage caractère par caractère de la chaîne CAB donne 1001. Or, ce code correspond à plusieurs chaînes possibles : BAAB (1-0-0-1), CAB (10-0-1), EB (100-1) et J (1001). C'est afin d'éviter ce problème de décodage multiple que la propriété du préfixe été définie.

Définition 28. *(propriété du préfixe pour une fonction de codage [113, p.13]). Une fonction de codage possède la propriété du préfixe si aucun de ses mots code n'est un préfixe d'un autre de ses mots code.*

À titre d'illustration et pour obtenir un résultat nécessaire par la suite (voir la relation 4.1), nous présentons une façon simple de coder un entier x avec cette propriété du préfixe. Ce codage auto-délimitant, noté λ, est constitué de x symboles 1 consécutifs suivis d'un 0 terminal :

$$\lambda(x) = \underbrace{11\ldots11}_{x \text{ fois}}0 = 1^x0.$$

Ce codage est cependant loin d'être efficace car sa longueur est de $x + 1$ bits. Par contre, en appliquant λ à la longueur de x, notée $l(x)$, on peut alors créer un nouvelle fonction de codage λ' telle que $\lambda'(x) = \lambda(l(x))x = 1^{l(x)}0x$ et pour laquelle $l(\lambda'(x)) = 2l(x) + 1$. On remarque alors que ce processus peut être réitéré par application de λ' sur $l(x)$ et ainsi de suite. Il en résulte la définition récursive d'une famille $(E_i)_{i\in\mathbb{N}}$ de fonctions de codage définie par :

$$\forall x \in \mathbb{N}, E_i(x) = \begin{cases} 1^x0 & \text{si } i = 0, \\ E_{i-1}(l(x))x & \text{si } > 0. \end{cases}$$

On s'intéresse alors aux tailles des éléments codés :

$$\begin{array}{ll} E_0(x) = 1^x 0 & l(E_0(x)) = x+1, \\ E_1(x) = E_0(l(x))x = 1^{l(x)}0x & l(E_1(x)) = 2l(x)+1, \\ E_2(x) = E_1(l(x))x = 1^{l(l(x))}0l(x)x & l(E_2(x)) = l(x)+2l(l(x))+1 \\ \vdots & \vdots \\ E_n(x) = 1^{l^n(x)}0\left(\prod_{i=1}^{n} l^i(x)\right)x & l(E_n(x)) = 1+\sum_{i=1}^{n}(l^i(x)+1). \end{array}$$

En remarquant que $l(0) = 1$ et que pour tout entier $x > 1$, $l(x) = \log x + 1$, on obtient la relation suivante :

$$\forall i > 1, l(E_i(x)) \leq l(x) + \mathcal{O}(\log l(x)) \tag{4.1}$$

Ce codage auto-délimitant est universel et asymptotiquement optimal [113, p.79].

Considérons maintenant une fonction de codage binaire notée h ayant la propriété du préfixe. La fonction h associe à chaque symbole X_i un mot code noté x_i de longueur l_i composé de 0 et de 1. On définit alors la longueur moyenne du code h, notée $L(h)$ comme l'espérance la variable aléatoire (l_i) :

$$L(h) = \sum_{i=1}^{n} p_i l_i$$

Théorème 12. *(Codage de source ou premier théorème de Shannon [153]). Soit $L_{inf} = \inf_h L(h)$ la longueur moyenne minimale des codages h ayant la propriété du préfixe, alors $\mathcal{H}(X) \leq L_{inf} \leq \mathcal{H}(X) + 1$.*

Cette approche probabiliste présente toutefois un inconvénient que nous illustrons en considérant, par exemple, une source S qui produit en sortie les décimales du nombre π, soit $S \to 31415926535897932384626433\ldots$ Bien qu'il ne soit pas démontré que π est un nombre normal [172], c'est-à-dire que sa représentation dans une base quelconque présente des digits répartis de manière uniforme, les 29 premiers millions de digits décimaux qui le composent sont effectivement uniformément distribués [6]. Dans ce cas, l'entropie de chaque décimale est d'environ $3,322$ bits d'information. Le théorème 12, nous permet alors d'estimer la taille d'un codage optimal ayant la propriété du préfixe des n premières décimales de π à $3,322 \times n$ bits. Or, ces n premières décimales sont calculables au moyen d'un algorithme simple, dont la taille est inférieure à $3,322 \times n$ bits pour n à partir d'un certain rang. Par exemple, le programme 4.1 de taille minimaliste dû à D. Winter [1] permet de calculer les 32 372 premières décimales de π. Le codage entropique de ces décimales nécessite pas loin de $32\,372 \times 3,322 = 13\,443$ octets alors que le programme fourni requiert uniquement 152 octets [2]. Or, ce programme minimaliste représente pourtant une description

1. Ce programme provient du site : `http://www.matpack.de/Info/Mathematics/Pi.html` (dernier accès en décembre 2010).

2. Cette taille de 152 octets est obtenue en supprimant les retours à la ligne ainsi que les espaces rajoutés pour un minimum de lisibilité sur le programme 4.1.

```
unsigned a=1e4,b,c=113316,d,e,f[113316],g,h,i;

main() {
  for(;b=c,c-=14;i=printf("%04d",e+d/a),e=d%a)
    while(g=--b*2)
      d=h*b+a*(i?f[b]:a/5),
      h=d/--g,
      f[b]=d-g*h;
}
```

Listing 4.1 – Programme C de 152 octets permettant de calculer les 32 372 premières décimales de π.

de l'information délivrée par la source S. C'est d'un tel constat qu'est née la complexité de Kolmogorov [100].

4.1.2 Théorie algorithmique de l'information : la complexité de Kolmogorov

De manière simplifiée, la complexité de Kolmogorov également connue sous le nom de complexité algorithmique ou encore de complexité calculatoire, représente le plus petit programme produisant une sortie souhaitée, pour un langage de programmation ainsi qu'une entrée donnée. Formellement, soit M une machine de Turing. On note $\mathcal{P}_M$ l'ensemble des programmes écrits pour la machine M. Pour un programme $p \in \mathcal{P}_M$, on note $l(p)$ la longueur du programme p exprimée en nombre d'instructions pour la machine M et $s(p,x)$ sa sortie pour l'entrée x. La complexité de Kolmogorov pour l'entrée y et la sortie x est définie par :

$$\mathcal{K}_M(x|y) = \begin{cases} \min_{p\in\mathcal{P}_M} \{l(p) \text{ tel que } s(p,y) = x\}, \\ \infty \text{ si un tel } p \text{ n'existe pas.} \end{cases}$$

Le programme p peut s'interpréter comme une description de la sortie y pour la machine M à partir de l'entrée x. Autrement dit, p représente l'information minimale nécessaire à la construction de la suite y exprimée pour la machine M à partir de la donnée x. Il s'agit là de la définition la plus générale de la complexité de Kolmogorov, appelée aussi complexité de Kolmogorov conditionnelle.

Bien entendu, il est possible de définir la complexité de Kolmogorov d'une donnée x de manière inconditionnelle en tant que plus petit programme, qui pour une machine de Turing donnée M et pour l'entrée vide, produit x en sortie. Autrement dit, $\mathcal{K}_M(x) = \mathcal{K}_M(x|\lambda_M)$ où λ_M désigne l'entrée vide pour M :

$$\mathcal{K}_M(x) = \begin{cases} \min_{p\in\mathcal{P}_M} \{l(p) \text{ tel que } s(p,\lambda_M) = x\}, \\ \infty \text{ si un tel } p \text{ n'existe pas.} \end{cases}$$

L'inconvénient de cette définition est sa dépendance par rapport à une machine de Turing M donnée. Toutefois, le théorème suivant montre que, à une constante additive près, la complexité de Kolmogorov est en réalité indépendante de la machine de Turing considérée.

Théorème 13. *(Invariance de la complexité de Kolmogorov [100]). Soient M et L deux machines de Turing. Soient $\mathcal{K}_M$ et $\mathcal{K}_L$ les complexités de Kolmogorov respectives des machines M et L. Alors il existe un constante C, ne dépendant que de M et de L telle que :*

$$\forall x, |\mathcal{K}_M(x) - \mathcal{K}_L(x)| \leq C$$

Dès lors, ce théorème nous permet de faire abstraction de la machine de Turing considérée. Pour la suite, nous nous fixons une machine de Turing universelle qui définit alors le langage de référence dans lequel est décrit le plus petit programme p permettant d'obtenir une sortie donnée (si toutefois un tel programme existe). Aussi, la complexité de Kolmogorov de la variable x sera dorénavant notée $\mathcal{K}(x)$ au lieu de $\mathcal{K}_M(x)$.

4.2 Distances fondées sur la complexité de Kolmogorov

La complexité de Kolmogorov représente une mesure absolue en termes d'information contenue dans un objet. À partir de cette mesure, il apparaît alors naturel de vouloir définir une distance absolue entre deux objets par rapport à l'information qu'ils contiennent.

Définition 29. *(Métrique). Soit X un ensemble, une application $D : X \times X \mapsto \mathbb{R}^+$ est une distance (métrique) sur X si pour tous x, y et z de X, elle vérifie :*

$$\text{(Identité)} \qquad (D(x,y) = 0) \Leftrightarrow (x = y), \tag{4.2}$$

$$\text{(Symétrie)} \qquad D(x,y) = D(y,x), \tag{4.3}$$

$$\text{(Inégalité triangulaire)} \qquad D(x,y) + D(y,z) \geq D(y,z). \tag{4.4}$$

Certaines métriques peuvent toutefois apparaître comme « non-réalistes ». C'est par exemple le cas pour une distance D qui, à tout couple (x, y) pour lequel $x \neq y$, associe 1. Afin d'écarter ce type de métriques jugées inappropriées, une classe particulière de distances, dites distances *admissibles*, a été définie en restreignant le nombre d'objets situés à une certaine distance d'un objet donné :

Définition 30. *(Distance admissible [113, p.541-542]). Une application $D : X \times X \mapsto \mathbb{R}^+$ est une distance admissible si pour tout couple (x, y) de X, la distance calculable $D(x, y)$ vérifie :*

$$\text{(Densité)} \qquad \sum_{y} 2^{-D(x,y)} \leq 1 \tag{4.5}$$

Cette définition est à mettre en rapport avec l'inégalité de Kraft [113, p.74] qui stipule que tout codage binaire h ayant la propriété du préfixe vérifie la relation suivante :

$$\sum_{i=1}^{k} 2^{-l_i} \leq 1,$$

où les $(l_i)_{i\in[\![1,k]\!]}$ désignent les longueurs des mots de code du codage h. En effet, si D est une distance admissible, alors pour tout x, l'ensemble $\{D(x,y), y \in \Omega\}$ représente l'ensemble des longueurs des codes ayant la propriété du préfixe. De manière équivalente, si une distance est la longueur d'un code ayant la propriété du préfixe alors elle satisfait 4.5 (voir [49]).

4.2.1 Distance informationnelle normalisée (NID)

Nous disposons à ce stade d'une mesure de complexité, permettant de quantifier l'information nécessaire à la description d'un objet représenté sous forme binaire. La distance informationnelle E entre deux objets x et y a été définie dans les travaux de Bennett *et al.* [15] au moyen de la complexité de Kolmogorov comme le plus petit programme permettant de produire x à partir de y aussi bien que y à partir de x.

$$E(x,y) = \min_{p\in\mathcal{P}} \{l(p) \text{ tel que } s(p,x) = y \text{ et } s(p,y) = x\}$$

On remarquera que cette distance peut aussi s'écrire sous la forme $E(x,y) = max\{\mathcal{K}(x|y), \mathcal{K}(y|x)\}$. L'un des principaux résultats démontrés dans [15] est que cette fonction est effectivement une métrique admissible :

Théorème 14. *(Admissibilité de la distance informationnelle [15]). La distance informationnelle est une métrique admissible.*

Bien que la plupart des distances admissibles soient absolues, il est souvent souhaitable, pour exprimer un degré de similarité, de disposer de distances relatives. Par exemple, si deux chaînes de 10^6 bits diffèrent seulement de 10^2 bits, nous sommes alors naturellement enclins à considérer ces chaînes comme relativement similaires. Par contre, si une chaîne de 2.10^2 bits diffère d'une autre de 10^2 bits alors elles apparaissent réciproquement peu similaires. Pour remédier à ce biais, cette distance est normalisée afin obtenir la distance informationnelle normalisée (« *Normalized Information Distance* » ou NID) définie comme suit :

$$NID(x,y) = \frac{\max\{\mathcal{K}(x|y), \mathcal{K}(y|x)\}}{\max\{\mathcal{K}(x), \mathcal{K}(y)\}}$$

En remarquant que la relation $\mathcal{K}(x|y) = \mathcal{K}(xy) - \mathcal{K}(y)$ est valide à une constante additive près, on obtient alors :

$$NID(x,y) = \frac{\mathcal{K}(xy) - \min\{\mathcal{K}(x), \mathcal{K}(y)\}}{\max\{\mathcal{K}(x), \mathcal{K}(y)\}}$$

4.2.2 Distance normalisée de compression (NCD)

Nous disposons à ce stade de tous les éléments nous permettant de comparer de manière universelle deux objets binaires. Toutefois, une interrogation persiste : comment calculer cette complexité de Kolmogorov ? Malheureusement, ce calcul n'est pas possible dans le cas général.

Théorème 15. *(Incalculabilité de la complexité de Kolmogorov [113, p.121]). La complexité de Kolmogorov n'est pas calculable.*

Bien que la complexité de Kolmogorov ne soit pas calculable, elle représente un outil mathématique puissant permettant notamment d'apporter des preuves plus simples à des théorèmes connus. Divers exemples sont donnés dans [113], comme la preuve du théorème d'incomplétude de Gödel ou encore l'infinité des nombres premiers.

Afin d'approcher cette mesure idéale qu'est la complexité de Kolmogorov, Li *et al.* [112] ont proposé de l'approximer au moyen d'un algorithme de compression de données sans perte. Cette approche produit une sur-approximation de la complexité de Kolmogorov en considérant que la version compressée d'une chaîne binaire x est un programme qui génère x sur une machine de « décompression ». Étant donnée que cette complexité n'est pas calculable, il est impossible de qualifier précisément la validité de cette approximation.

Comme dans [41], nous définissons un algorithme de compression comme un codage ayant la propriété du préfixe, défini sur Ω et à valeurs dans Ω. L'avantage de la propriété du préfixe est d'assurer que chaque séquence de code est décodable de manière unique. À chaque compresseur $\mathcal{C}$, nous associons la fonction correspondante $C : \Omega \mapsto \mathbb{N}$, qui a tout élément x de Ω associe la taille, notée $C(x)$, du résultat de la compression de x par $\mathcal{C}$ ($C(x) = |\mathcal{C}(x)|$). Nous considérons ici seulement des compresseurs ayant propriété du préfixe et vérifiant la relation 4.1, c'est-à-dire, tels que $C(x) \leq |x| + \mathcal{O}(\log |x|)$. Cette condition est effectivement remplie par les algorithmes standards de compression [41].

De manière intuitive, il apparait évident que les résultats obtenus sont directement liés à l'efficacité du compresseur considéré. Pour formaliser cette remarque, la propriété de *normalité* des compresseurs a été définie comme suit :

Définition 31. *(Compresseur normal [41]). Un compresseur $\mathcal{C}$ défini sur un ensemble Ω est normal si, pour tout x, y et z de Ω, il satisfait à une précision logarithmique près*[3] *:*

$$\begin{aligned}
&\textit{(Idempotence)} && C(xx) = C(x) \text{ et } C(\epsilon) = 0, && (4.6)\\
&\textit{(Monotonicité)} && C(x) \leq C(xy), && (4.7)\\
&\textit{(Symétrie)} && C(xy) = C(yx), && (4.8)\\
&\textit{(Distributivité)} && C(xy) + C(z) \leq C(xz) + C(yz). && (4.9)
\end{aligned}$$

Il existe une propriété de distributivité plus forte portant sur la complexité de Kolmogorov. Cette propriété est aussi valide dans le cas d'un compresseur normal :

$$\forall (x, y, z) \in \Omega^3, C(xyz) + C(z) \leq C(xz) + C(yz) \tag{4.10}$$

3. la précision mentionnée est de l'ordre de $\mathcal{O}(\log n)$, où n désigne la longueur du plus grand élément de Ω concerné par la relation décrite. Cette précision correspond à la relation 4.1 vérifiée par le compresseur employé. Dorénavant, cette expression désignera toujours cette même précision logarithmique.

Étant donné un compresseur $\mathcal{C}$, la distance normalisée de compression (« *Normalized Compression Distance* » ou NCD) est définie par :

$$NCD(x,y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}$$

Théorème 16. *(NCD, métrique admissible sur Ω [41]). Si un compresseur $\mathcal{C}$ est normal sur Ω alors la fonction NCD associée est une distance admissible sur Ω.*

De plus cette distance est universelle, au sens où deux objets sont similaires, par rapport à une propriété donnée, s'ils sont aussi similaires respectivement à NCD.

Théorème 17. *(Universalité de NCD [41]). Soit d une distance calculable. Étant donnée une constante $a > 0$, soient x et y des objets tels que $C(xy) - \mathcal{K}(xy) \leq a$. Alors,*

$$NCD(x,y) \leq d(x,y) + \frac{(a + \mathcal{O}(1))}{\max\{C(x), C(y)\}}$$

On remarquera que cette distance correspond en fait à une mesure de dissimilarité entre objets. En effet, si $x = y$ alors d'après 4.6, $C(xy) = C(xx) = C(x)$ et donc, $NCD(x,y) = 0$. Maintenant, si x et y sont totalement différents, alors $C(xy) = C(x) + C(y)$ et donc, $NCD(x,y) = 1$. La mesure de similarité correspondante est appelée la similarité normalisée de compression (« *Normalized Compression Similarity* » ou NCS) et se définit par :

$$NCS(x,y) = 1 - NCD(x,y) = \frac{C(x) + C(y) - C(xy)}{\max\{C(x), C(y)\}}$$

4.3 Vers une nouvelle mesure de similarité par compression

Nous exposons maintenant une limitation potentielle de NCD (et donc de NCS) à l'origine de notre nouvelle mesure de similarité. Considérons pour cela deux éléments x et y tels que l'information contenue dans x se « retrouve » dans celle de y. Intuitivement, nous pouvons supposer que cette propriété d'« inclusion d'information » peut s'exprimer au moyen d'un algorithme de compression normal $\mathcal{C}$ par la relation $C(xy) = C(y)$ (ce résultat est prouvé en section 4.3.2 proposition 2). Il est alors facile de voir, par monotonicité 4.7 de $\mathcal{C}$, que $C(x) \leq C(y)$ et donc que :

$$NCS(x,y) = \frac{C(x) + C(y) - C(xy)}{C(y)} = \frac{C(x)}{C(y)}$$

Dans ce cas, la taille du compressé de y impacte directement sur le résultat de la mesure de similarité.

Nous illustrons maintenant cette remarque par un exemple simple. Supposons qu'un élève Oscar doive rédiger une poésie pour son cours de français. Peu motivé, ce dernier décide de s'inspirer d'une fable de la Fontaine qu'il affectionne particulièrement : « le Lion et le Moucheron. » Dans un excès de paresse, Oscar recopie directement quelques vers pour en finir avec sa composition. Les deux fables correspondantes sont données en annexe D, où le plagiat d'Oscar correspond aux lignes grisées dans la fable originale. En utilisant NCS [4], nous obtenons un taux de similarité de 43,02% entre ces deux fables. Pour ce qui est de notre mesure d'inclusion présentée en section 4.3.3 nous obtenons un taux de similarité de $96,65\%$. Ces résultats montrent qu'une distance n'est pas toujours appropriée pour la comparaison de deux objets. C'est à partir de ce constat qu'une nouvelle mesure de similarité a été proposée à partir de la notion de degré d'inclusion.

4.3.1 Degré d'inclusion

Le concept de degré d'inclusion est facilement illustrable en théorie des ensembles. Soit U un ensemble fini, $F = \{X | X \subseteq U\}$ où $\subseteq$ désigne la relation d'inclusion classique sur les ensembles. Pour tous X et Y de F, nous définissons une fonction $D_0 : F \times F \mapsto \mathbb{R}^+$ par :

$$D_0(X,Y) = \begin{cases} \frac{|X \cap Y|}{|X|} & \text{si } X \neq \emptyset, \\ 1 & \text{si } X = \emptyset, \end{cases}$$

où $|X|$ désigne la cardinalité de X. La fonction D_0 correspond alors effectivement à la notion un degré d'inclusion sur F.

Comme en témoigne cet exemple, la définition d'un degré d'inclusion nécessite de se munir d'une relation d'ordre partiel.

Définition 32. *(Ordre partiel sur un ensemble). Un ordre partiel sur un ensemble E est une relation binaire $\preceq$ vérifiant, pour tous x, y et z de E :*

$$\text{(Réflexivité)} \qquad x \preceq x, \tag{4.11}$$

$$\text{(Antisymétrie)} \qquad ((x \preceq y) \text{ et } (y \preceq x)) \Rightarrow (x = y)), \tag{4.12}$$

$$\text{(Transitivité)} \qquad ((x \preceq y) \text{ et } (y \preceq z)) \Rightarrow (x \preceq z). \tag{4.13}$$

Le concept de degré d'inclusion fondé sur un ordre partiel a été proposé par Zhang *et al.* [191] pour le raisonnement approximatif. Nous reprenons ce formalisme pour définir un degré d'inclusion, en termes d'information contenue dans deux objets, au moyen d'un algorithme de compression sans perte.

Définition 33. *(Degré d'inclusion [191]). Soit $(E, \preceq)$ un ensemble partiellement ordonné. Un degré d'inclusion est une application $D : E \times E \to \mathbb{R}$ vérifiant,*

4. Les mesures de similarités utilisée ici sont calculés au moyen de LZMA comme algorithme de compression. Ce choix est justifié au prochain chapitre en section 5.3.1

pour tous x, y et z de E :

$$\textit{(Normalisation)} \qquad 0 \leq D(x,y) \leq 1, \tag{4.14}$$

$$\textit{(Consistance)} \qquad (x \preceq y) \Rightarrow (D(x,y) = 1)), \tag{4.15}$$

$$\textit{(Monotonicité 1)} \qquad (x \preceq y \preceq z) \Rightarrow (D(z,x) \leq D(y,x)), \tag{4.16}$$

$$\textit{(Monotonicité 2)} \qquad (x \preceq y) \Rightarrow (D(z,x) \leq D(z,y)). \tag{4.17}$$

Cette définition correspond bien à la notion intuitive de « degré d'inclusion ». En particulier et contrairement à NCS, la propriété de consistance nous garantit une valeur maximale de la fonction en cas d'inclusion.

4.3.2 Degré d'inclusion par compression (CID)

La définition d'un degré d'inclusion nécessite la donnée d'un ordre partiel. Nous définissons alors une relation binaire notée $\lhd$ sur Ω comme suit :

$$\big(\forall (x,y) \in \Omega^2, (x \lhd y)\big) \Leftrightarrow \big(\exists (a,b) \in \Omega^2, y = a.x.b\big) \tag{4.18}$$

La notation $a.x$ désigne la concaténation de la chaîne a avec la chaîne y. Pour simplifier l'écriture nous noterons par la suite ax au lieu de $a.x$. Cette relation définit bien un ordre partiel sur Ω.

Preuve. *Nous rappelons que ϵ désigne la chaîne vide sur Ω. $\forall x \in \Omega, x = \epsilon x \epsilon$, soit $x \lhd x$, aussi $\lhd$ est réflexive. Supposons maintenant que $x \lhd y$ et $y \lhd x$ alors $\exists (a,b,c,d) \in \Omega^4, y = axb$ et $x = cyd$, donc $y = acydb$. Dans ce cas, $a = c = d = b = \epsilon$ et $x = y$ donc $\lhd$ est antisymétrique. Finalement, supposons que $x \lhd y$ et que $y \lhd z$, alors $\exists (a,b,c,d) \in \Omega^4, y = axb$ et $z = cyd$. Aussi, $z = (ca)x(bd)$ soit $x \lhd z$, $\lhd$ est transitive.* □

À partir de la relation d'ordre partiel $\lhd$, nous définissons notre fonction d'inclusion par compression.

Définition 34. *(Fonction d'inclusion par compression). Soit $\mathcal{C}$ un compresseur sur l'ensemble $(\Omega, \lhd)$. Nous définissons la fonction de degré d'inclusion par compression (« Compression based Inclusion Degree » ou CID) notée CID : $\Omega \times \Omega \to \mathbb{R}$ comme suit :*

$$CID(x,y) = 1 - \frac{C(xy) - C(y)}{C(x)} = \frac{C(x) + C(y) - C(xy)}{C(x)} \tag{4.19}$$

Nous montrons maintenant que cette fonction d'inclusion par compression est bien un degré d'inclusion sous l'hypothèse que le compresseur utilisé est normal. Pour celà, nous commençons par introduire plusieurs résultats intermédiaires.

Lemme 1. *Soit $\mathcal{C}$ un compresseur normal sur Ω alors, pour tous éléments x et y de Ω, les relations suivantes sont vérifiées à une précision logarithmique près :*

$$C(xy) \leq C(x) + C(y) \tag{4.20}$$

$$C(xxy) = C(xy) \tag{4.21}$$

Preuve. *Le premier résultat (4.20) découle directement de la propriété de distributivité (4.9) en substituant la chaîne vide ϵ à l'élément z, soit $C(xy)+C(\epsilon) \leq C(x\epsilon)+C(y\epsilon)$, ce qui donne le résultat attendu : $C(xy) \leq C(x)+C(z)$.*

Pour le second résultat (4.21), nous partons de la propriété de symétrie (4.8) $C(xxy) = C(xyx)$. Par distributivité (4.9) on a $C(xyx) \leq C(xx)+C(yx)-C(x)$. Par idempotence (4.6), $C(xx) = C(x)$ nous donne $C(xyx) \leq C(yx)$. Or, par monotonicité (4.7), $C(xy) \leq C(xyx)$, ce qui nous donne le résultat attendu : $C(xxy) = C(xy)$ □

Ce résultat nous permet d'établir la démonstration de la position suivante :

Proposition 2. *(Consistance d'un compresseur normal). Un compresseur normal $\mathcal{C}$ défini sur l'ensemble partiellement ordonné $(\Omega, \lhd)$ satisfait la propriété suivante à une précision logarithmique près :*

$$\textit{(Consistance)} \qquad \big(\forall (x,y) \in \Omega^2, (x \lhd y)\big) \Rightarrow (C(xy) = C(y)) \qquad (4.22)$$

Preuve. *Supposons que $x \lhd y$, alors par définition 4.18, $\exists (a,b) \in \Omega^2, y = axb$. Dans ce cas, $C(xy) = C(xaxb)$. Par distributivité (4.10), nous avons $C(x.a.(xb)) \leq C(xxb)+C(axb)-C(xb)$. D'après 4.21, $C(xxb) = C(xb)$, aussi, $C(xaxb) \leq C(axb)$. Or, par monotonicité 4.7, nous avons que $C(axb) \leq C(x(axb))$. Donc, $C(xaxb) = C(axb)$ qui est le résultat escompté : $C(xy) = C(y)$.* □

Nous disposons maintenant de tous les résultats nécessaires à la démonstration de la proposition suivante :

Proposition 3. *(Degré d'inclusion par compression). Si un compresseur $\mathcal{C}$ est normal sur l'ensemble partiellement ordonné $(\Omega, \lhd)$, alors la fonction CID associée, présentée en définition 34, est un degré d'inclusion.*

Preuve. *Nous supposons que le compresseur $\mathcal{C}$ est normal sur l'ensemble partiellement ordonné $(\Omega, \lhd)$.*

1. ***Normalisation**. Par distributivité 4.9, $\forall (x,y,z) \in \Omega^3, C(xy)+C(z) \leq C(xz)+C(yz)$. D'après (4.20), $0 \leq C(x)+C(y)-C(xy)$ et donc $0 \leq CID(x,y)$. Par monotonicité (4.7), nous avons $\forall (x,y) \in \Omega^2, C(y) \leq C(yx) = C(xy)$ par symétrie (4.8), aussi $C(y)-C(xy)+C(x) \leq C(x)$ et donc $CID(x,y) \leq 1$;*
2. ***Consistance**. Considérons x et y de Ω tels que $x \lhd y$. D'après (4.22), nous avons $C(xy) = C(y)$ et donc,*

$$CID(x,y) = 1 - \frac{C(xy)-C(y)}{C(x)} = 1 - \frac{C(y)-C(y)}{C(x)} = 1\,;$$

3. ***Monotonicité 1**. Considérons x, y et z de Ω tels que $x \lhd y \lhd z$. D'après (4.22), comme $x \lhd z$, $C(xz) = C(z)$. Pour les mêmes raisons, $x \lhd y$ donne $C(xy) = C(y)$. Aussi, $C(z)-C(xz) = C(y)-C(xy) = 0$ et donc $C(x)+C(z)-C(xz) = C(x)+C(y)-C(xy)$. Par monotonicité (4.7) et par définition de $\lhd$ (4.18), nous obtenons que $C(y) \leq C(z)$. D'où finalement,*

$$\frac{C(x)+C(z)-C(xz)}{C(z)} \leq \frac{C(x)+C(y)-C(xy)}{C(y)}$$

ce qui correspond au résultat attendu, $CID(z,x) \leq CID(y,x)$.

4. ***Monotonicité 2****. Considérons* x *et* y *de* Ω *tels que* $x \triangleleft y$*. Pour tout* z *de* Ω*, la distributivité (4.9) donne* $C(yz)+C(x) \leq C(yx)+C(zx)$*. Aussi, par symétrie (4.8),* $C(x) - C(xz) \leq C(xy) - C(yz)$*. En ajoutant* $C(z)$*, nous obtenons que* $C(x) + C(z) - C(xz) \leq C(xy) - C(yz) + C(z)$*. En divisant par* $C(z)$*,*

$$CID(z,x) \leq \frac{C(xy) + C(z) - C(yz)}{C(z)}$$

Finalement, (4.22) donne $C(xy) = C(y)$*, ce qui conduit au résultat attendu,* $CID(z,x) \leq CID(z,y)$.

□

4.3.3 Degré d'inclusion mutuelle par compression (CMID)

Considérons maintenant deux éléments x et y de Ω. Il apparaît plus « naturel » de mesurer le degré d'inclusion de l'élément contenant le moins d'information par rapport à celui qui en contient le plus, c'est-à-dire, de calculer $CID(x,y)$ dans le cas où $C(x) \leq C(y)$. En effet, dans le cas contraire où $C(x) \geq C(y)$:

$$CID(x,y) = \frac{C(x) + C(y) - C(xy)}{C(x)} = \frac{C(x) + C(y) - C(xy)}{\max\{C(x), C(y)\}} = NCS(x,y)$$

Dans ce cas, la limitation présentée en début de section 4.3 tient toujours. Pour cette raison, nous définissons un degré d'inclusion mutuelle par compression (« *Compression based Mutual Inclusion Degree* » ou CMID) entre deux objets x et y comme le plus grand degré d'inclusion entre $CID(x,y)$ et $CID(y,x)$. Ce qui nous donne :

$$\forall (x,y) \in \Omega^2, CMID(x,y) = \frac{C(x) + C(y) - C(xy)}{\min\{C(x), C(y)\}}$$

4.3.4 Exemple d'application

Pour illustrer l'intérêt de cette mesure de similarité, nous proposons un scénario d'application pratique. Dans [28], Chen *et al.* utilisent une distance de compression afin de détecter des cas de plagiats dans des programmes développés par des étudiants. Le principe consiste à parser les sources de chaque programme afin d'obtenir n éléments syntaxiques de comparaison. Ensuite toutes les distances entre éléments syntaxiques sont calculées. On obtient alors une matrice carré de $n \times n$ distances, ce qui permet de déterminer quels sont les éléments copiés et quels sont les coupables de plagiats.

Notre scénario concerne aussi un cas de plagiat mais sur un programme binaire pour lequel un découpage syntaxique n'est pas autorisé (dans le strict respect des lois). Considérons pour cela une entreprise A qui vend les sources d'une bibliothèque applicative b à ses clients. A suspecte une autre entreprise E d'utiliser la bibliothèque b, qu'elle n'a pas achetée, dans un de ses produits

commerciaux l. A souhaiterait alors confirmer ses doutes quant à la présence de sa propre bibliothèque b dans le binaire du logiciel l.

Dans le cadre de la législation en cours, A n'a absolument pas le droit de rétro-concevoir l'application l. Heureusement pour A, l'utilisation de CMID va lui permettre de confirmer ses doutes en révélant le fort niveau d'inclusion de son code dans l'application l. Afin d'illustrer ce scénario, nous considérons que A fournit le kit de développement logiciel (« *Software Development Kit* » ou SDK) de l'algorithme « *Lempel-Ziv-Markov chain-Algorithm* » (LZMA) [5]. E commercialise le logiciel TrueCrypt [6] soupçonné d'utiliser les fonctions `LzmaCompress` et `LzmaUncompress` de la bibliothèque LZMA sans l'avoir au préalable achetée auprès de A. Nous précision ici que cet exemple est bien sûr fictif. Le SDK de LZMA appartient au domaine public et la fondation TrueCrypt n'utilise pas cette bibliothèque.

A utilise alors un programme du type PEid [7] afin de déterminer le compilateur utilisé pour produire l. À partir de cette information, A peut compiler sa bibliothèque en modifiant les principales options d'optimisation qui peuvent influencer le résultat (optimisation privilégiant soit la taille, soit la vitesse d'exécution du code produit avec Visual Studio par exemple). Pour valider cette approche, nous donnons dans le tableau 4.2 les résultats de similarité obtenus avec NCS et CMID sur plusieurs versions du programme TrueCrypt. Tous ces résultats sont donnés avec Visual Studio 2008 comme compilateur. La première version d'origine est obtenue par recompilation. La seconde version correspond aux sources d'origine avec ajout des fonctions `LzmaCompress` et `LzmaUncompress`. Ces résultats sont donnés pour une optimisation privilégiant la vitesse du code mais les autres options sont différentes (`/O2` pour TrueCrypt et `/Ox /Ob2 /Oi /Ot /Oy /GT /GL` pour la bibliothèque b). La dernière ligne représente le cas où les options de compilation des deux projets sont identiques (`/O2` pour les deux). On remarque que dans les trois cas NCS présente un taux de similarité infé-

Programme l	$NCS(b,l)$	$CMID(b,l)$
TrueCrypt original	0,17 %	5,41 %
TrueCrypt+LZMA	2,05 %	**69,21** %
TrueCrypt+LZMA mêmes options	2,95 %	**94,17** %

TABLE 4.2 – Mesures de similarité obtenues entre TrueCrypt et la bibliothèque LZMA pour différentes options de compilation.

rieur à 3%. Par contre, CMID offre un taux de similarité de plus de 94% en cas d'inclusion pour seulement 5,41% avec la version originale de TrueCrypt si les deux projets sont compilés avec les mêmes options. Ce taux chute à 69 % si la seule option commune est l'optimisation en faveur d'un code rapide ou de petite

5. La version utilisée est la 9.20 du 18/11/2010, téléchargeable à l'URL suivante : `http://www.7-zip.org/sdk.html` (dernier accès en décembre 2010).

6. La version utilisée est la 7.0a, téléchargeable à l'URL suivante : `http://www.truecrypt.org` (dernier accès en décembre 2010).

7. Téléchargeable à l'URL suivante : `http://www.peid.info` (dernier accès en décembre 2010).

taille. Ce tableau conforte donc les soupçons de A qui peut alors légitimement traduire l'entreprise E en justice.

4.4 Discussion

Revenons maintenant sur les différentes mesures de similarité étudiées ainsi que les diverses façons d'envisager la notion d'information pour montrer les relations qui les lient. La construction théorique de NCS et CMID repose sur la complexité de Kolmogorov. Cette complexité absolue n'étant pas calculable, nous sommes alors ramenés à l'approximer au moyen d'un compresseur entropique. En notant $ID(x,y) = C(x) + C(y) - C(xy)$, on remarque que cette fonction correspond à l'approximation de la distance informationnelle présentée en section 4.2.1 au moyen d'un algorithme de compression sans perte. Or, l'information mutuelle de deux variables aléatoires X et Y se définit au moyen de l'entropie de Shannon par $I_{\mathcal{H}}(X,Y) = \mathcal{H}(X)+\mathcal{H}(Y)-\mathcal{H}(X,Y)$. Elle correspond à la diminution de l'incertitude sur X dûe à la connaissance de Y. On constate alors que ces deux mesures de similarité correspondent à deux manières différentes de normaliser une approximation de l'information mutuelle obtenue au moyen d'un algorithme de compression sans perte. En effet, pour tout élément x et y de Ω, on a la relation :

$$0 \leq NCS(x,y) = \frac{ID(x,y)}{\max\{C(x),C(y)\}} \leq CMID(x,y) = \frac{ID(x,y)}{\min\{C(x),C(y)\}} \leq 1$$

Ce résultat signifie que deux éléments quelconques x et y sont plus éloignés au sens de NCS qu'ils ne sont mutuellement inclus l'un dans l'autre au sens de CMID.

4.5 Bilan des mesures de similarité

Dans ce chapitre, nous avons présenté les bases théoriques sur lesquelles s'appuie notre approche de détection de codes malveillants exposée dans le chapitre suivant. C'est ainsi que la complexité de Kolmogorov a été utilisée comme socle théorique pour la définition de deux mesures de similarité. Cependant, cette complexité n'est pas calculable. Une approximation est alors nécessaire afin de l'approcher. C'est pourquoi il a été proposé de remplacer la complexité de Kolmogorov par un algorithme de compression sans perte.

La première mesure de similarité proposée (NCD) correspond à une distance universelle entre deux objets binaires du moment que le compresseur utilisé est *normal*. Comme nous l'avons présenté, cette distance éloigne deux objets en fonction des tailles de leurs compressés respectifs. Or, nous avons aussi illustré l'intérêt d'une mesure universelle d'inclusion entre deux objets.

La seconde mesure de similarité, CMID, constitue la principale contribution de ce chapitre, l'autre contribution étant l'origine de cette mesure, c'est-à-dire la limitation de NCD. Cette mesure correspond au degré d'inclusion d'information

mutuelle d'un objet dans un autre. Nous avons démontré que cette fonction CMID est effectivement un degré d'inclusion au sens formel, si le compresseur employé est *normal*. D'un point de vue pratique, nous avons illustré l'avantage de cette mesure dans le cadre de la protection intellectuelle à travers un exemple concret.

Perspective 3 (Applications de CMID).

Le champ d'applications possibles de CMID reste encore à explorer dans de nombreux domaines où le concept de degré d'inclusion est plus approprié que la notion de distance.

Dans le prochain chapitre, nous employons ces deux mesures de similarité dans le cadre de la détection de codes malveillants.

Chapitre 5

Architecture générique de détection de codes malveillants fondée sur la similarité comportementale

Ce chapitre présente le fonctionnement de notre système de détection dynamique fondé sur l'hypothèse que des variantes de codes malveillants se comportent de façon similaire. Afin de formaliser cette notion de similarité, nous nous appuyons sur les mesures NCD et CMID définies au chapitre 4. L'avantage de ces mesures algorithmiques, fondées sur la complexité de Kolmogorov, résilde dnas leur universalité. En effet, si deux objets sont proches pour une distance quelconque alors ils le seront aussi pour NCD (voir le théorème 17 page 115). C'est pourquoi la distance NCD a déjà été utilisée avec succès dans le cadre de la lutte contre les codes malveillants. Par exemple, les travaux de Wehner [181] emploient la compression de données directement sur des exécutables de vers, non compressés et non chiffrés, afin d'identifier ceux appartenant à une même famille. De même, les travaux de Bailey *et al.* [7] utilisent NCD dans le cadre de la classification non-supervisée de rapports d'activité de codes malveillants.

Notre approche de détection comporte trois composantes illustrées en figure 5.1 :

1. Le premier élément est le générateur de profils comportementaux. Chaque programme soumis à détection est exécuté durant une certaine période. Au cours de cette exécution, les appels systèmes sont observés et synthétisés sous la forme d'une trace d'exécution. Chaque trace est ensuite analysée pour en abstraire au maximum les informations contextuelles propres à l'environnement d'exécution. Le résultat de ce processus est un profil

comportemental (ou plus simplement profil) qui représente l'élément de comparaison de base dans notre approche ;

2. Le deuxième composant est la base de données regroupant des profils comportementaux de codes malveillants déjà identifiés. Cette base contient les profils correspondant aux souches originales à partir desquelles les variantes doivent être détectées ;
3. Le dernier élément est le comparateur de profils comportementaux qui, à partir de la base de profils malveillants et du profil testé, fournit un résultat de détection. Ce résultat est obtenu au moyen de l'une des deux mesures de similarité présentées au chapitre précédent, NCS ou CMID.

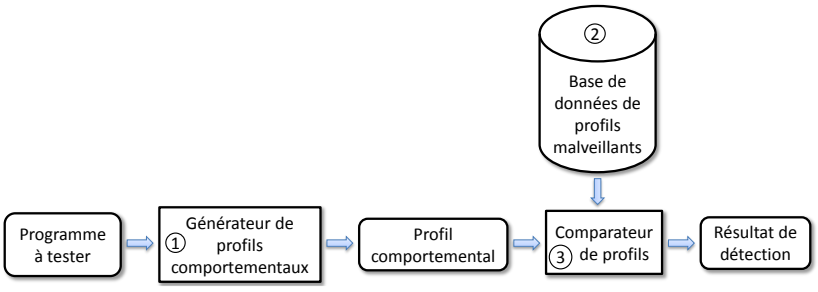


FIGURE 5.1 – Architecture du système de détection

Ce chapitre s'organise conformément à cette architecture. Dans la section 5.1 nous présentons de quelle manière les profils comportementaux sont générés à partir de l'exécution d'un programme. Dans la section 5.2 nous expliquons comment construire une base de détection de taille minimale. Dans la section 5.3 nous présentons les expériences menées afin d'évaluer notre approche. Cette évaluation concerne à la fois les performances de détection à proprement parlé, ainsi que les résultats respectifs de NCS et de CMID dans le cadre de la détection de codes malveillants.

5.1 Génération des profils comportementaux

Nous avons vu en section 1.3.3.1 que le programme d'observation constitue un élément essentiel pour la détection dynamique de codes malveillants puisque sa précision impacte directement les résultats obtenus. Deux choix sont alors possibles : soit utiliser un environnement d'analyse dédié, qui autorise alors une observation plus fine et plus précise ; soit un outil directement déployable sur un poste client. Du point de vue de l'utilisateur final, notre approche se veut la plus transparente possible. Ce choix exclut la modification dynamique de binaire ainsi que les environnements dédiés. En effet, des codes malveillants peuvent vérifier leur intégrité, ou encore modifier leur propre code notamment dans le cas du métamorphisme.

Choix 4 (Générateur de profils comportementaux).

Nous avons fait le choix de développer notre propre système de détection, à commencer par le générateur de profils comportementaux.

5.1.1 Obtention d'une trace d'exécution

Afin d'observer l'activité d'un programme, nous interceptons les appels systèmes avec leurs paramètres. Pour être plus précis, nous ne monitorons pas chaque appel système indépendamment mais nous regroupons tous ceux qui concourent à la même action finale. En effet, les appels systèmes peuvent considérablement varier entre des programmes qui montrent le même comportement. Par exemple, la lecture d'un fichier peut s'envisager de différentes façons : le plus simplement possible, en lisant directement le fichier dans son intégralité, ou encore en le lisant de manière linéaire en plusieurs fois. Dans ces deux cas, le même but est atteint. Cependant, dans le premier cas, la trace d'exécution correspondante ne contient qu'un seul et unique appel système alors que le second cas est composé de plusieurs accès en lecture. Pour cette raison, nos traces d'exécution ne mentionnent que les actions terminales sur le système d'exploitation indépendamment de leur enchaînement. La syntaxe simplifiée de nos traces d'exécution est la suivante :

```
ExecutionTrace ::= TraceEntry ExecutionTrace | TraceEntry;
TraceEntry ::= ObjectType ObjectOperation OperationAttributes;
ObjectType ::= DRIVER|FILE|IMAGE|KEY|KEYVALUE|NETWORK|PROCESS|
    REGISTRY|SERVICE|SYNCHRONIZATION|THREAD;
ObjectOperation ::= CONNECT|LISTEN|LOAD|NEW|READ|WRITE;
OperationAttributes ::= String OperationAttributes | String;
```

où `ObjectType` représente tous les types d'objets manipulés par les appels systèmes, `ObjectOperation` désigne les différents types d'actions menées par un appel système sur un objet spécifique et `OperationAttributes` décrit une chaîne de caractères représentant les paramètres d'un appel système.

Nous présentons une illustration de fonctionnement global de notre générateur de profils à travers l'exemple du cheval de Troie Trojan-PSW.Win32.-Lmir.zr (d'après la convention de nommage de l'antivirus Kaspersky). La trace d'exécution correspondante est donnée en figure 5.1.1.

Dans cet exemple, le code malveillant commence par écrire un binaire nommé `KVXP.exe` dans le répertoire système. Ensuite, en ligne 2, ce nouveau binaire est enregistré pour démarrer automatiquement. En ligne 3, `KVXP.exe` est lancé, ce qui conduit à la création d'une nouvelle DLL `KVXP.dll`. Cette DLL est écrite dans le répertoire système en ligne 4 et chargée par le processus courant en ligne 5. Finalement, un script batch, écrit dans le répertoire Windows en ligne 6, est lancé en ligne 7 par la commande `cmd`.

Au niveau du système d'exploitation, les chemins peuvent se présenter sous divers formes : soit sous forme native, soit sous forme utilisateur. Par exemple, le chemin natif `\Device\HarddiskVolume1\WINDOWS\system32` (ligne 1)

```
1 FILE WRITE     "\Device\HarddiskVolume1\WINDOWS\system32\KVXP.exe"
2 REGISTRY WRITE "\REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows\
      CurrentVersion\Run" REG_SZ "Windows" "C:\WINDOWS\System32\KVXP.exe"
3 PROCESS NEW    "C:\WINDOWS\system32\KVXP.exe"
4 FILE WRITE     "\Device\HarddiskVolume1\WINDOWS\System32\KVXP.dll"
5 IMAGE LOAD     "\WINDOWS\System32\KVXP.dll"
6 FILE WRITE     "\Device\HarddiskVolume1\WINDOWS\Deleteme.bat"
7 PROCESS NEW    "cmd /c \Device\HarddiskVolume1\WINDOWS\Deleteme.bat"
```

FIGURE 5.2 – Trace d'exécution du cheval de Troie TROJAN-PSW.WIN32.-LMIR.ZR.

représente le chemin `C:\WINDOWS\system32` (ligne 2), qui apparait lui même sous la forme `\WINDOWS\system32` (ligne 5). Dans tous les cas, ces chemins désignent le même répertoire système dont la valeur dépend à la fois du type de système d'exploitation et de l'installation.

5.1.2 Abstraction d'une trace d'exécution

Afin d'éviter au maximum la présence d'informations contextuelles à l'environnement d'exécution, ce qui pourrait conduire à un biais en terme de similarité, nous abstrayons les traces d'exécution obtenues en section précédente. Plusieurs sources d'informations sont exploitées aussi bien au niveau du système d'exploitation que de la session utilisateur. Pour l'instant, notre implémentation prend en compte les variables d'environnement du processus courant, les noms de répertoires connus, les clés de registre connues ainsi que les différents identifiants de sécurité (« *Security Identifiers* » ou SIDs) sur le poste courant. Plusieurs actions suspectes sont aussi mises en avant pour la détection. Actuellement, sont répertoriés les injections de « *threads* », la mise en résidence à travers la modification de fichiers et des actions sur le registre, et des cas d'auto-reproduction lorsque les binaires produits apparaissent suffisamment similaires au programme d'origine par rapport aux mesures présentées précédemment.

Pour illustrer ce processus d'abstraction, nous donnons le profil comportemental du même cheval de Troie TROJAN-PSW.WIN32.LMIR.ZR en figure 5.1.2 dont la trace d'exécution correspond à la figure 5.1.1.

Dans cet exemple, tous les chemins sont abstraits en fonction des variables d'environnement et des noms de répertoires connus. Les chemins `\Device\HarddiskVolume1\WINDOWS\system32\` (ligne 1), `C:\WINDOWS\system32\` (ligne 2) et `\WINDOWS\System32` (ligne 5) de la figure 5.1.1 sont convertis via l'identifiant `CSIDL_SYSTEM` en figure 5.1.2. De manière équivalente, le chemin `\%SystemRoot\%` en lignes 6 et 7 de la figure 5.1.2 remplace `\Device\HarddiskVolume1\WINDOWS\` de la figure 5.1.1. La mise en résidence est illustrée en deuxième ligne de la figure 5.1.2 avec l'ajout du tag `AUTORUN` à l'entrée `REGSITRY WRITE` de la figure 5.1.1. L'auto-reproduction est représentée à travers la première ligne de la figure 5.1.2 pour laquelle l'entrée `FILE WRITE` de la figure 5.1.1 est remplacée par `IMAGE SELF_COPY`.

```
1 IMAGE SELF-COPY          "%CSIDL_SYSTEM%\KVXP.exe"
2 REGISTRY WRITE AUTORUN   "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion
      \Run" REG_SZ "Windows" "%CSIDL_SYSTEM%\KVXP.exe"
3 PROCESS NEW              "%CSIDL_SYSTEM%\KVXP.exe"
4 IMAGE WRITE              "%CSIDL_SYSTEM%\KVXP.dll"
5 IMAGE LOAD               "%CSIDL_SYSTEM%\KVXP.dll"
6 FILE WRITE               "%SystemRoot%\Deleteme.bat"
7 PROCESS NEW              "cmd /c %SystemRoot%\Deleteme.bat"
```

FIGURE 5.3 – profil comportemental du cheval de Troie TROJAN-PSW.WIN32.-LMIR.ZR.

5.2 Construction d'une base optimale de détection comportementale

Dans cette section, nous supposons disposer d'une mesure de similarité qui constitue la base de notre approche de détection. En effet, nous considérons un profil comme malveillant s'il apparait suffisamment similaire à un autre contenu dans une base de profils malveillants de référence. Étant donnée une fonction de similarité, la définition d'une base de profils malveillants est alors conditionnée par le seuil de similarité fixé pour définir la correspondance entre deux profils. Nous nous intéressons ici à la définition d'une base de détection optimale d'un point de vue de sa taille, c'est à dire contenant le moins possible d'éléments pour une mesure de similarité et un seuil donnés.

5.2.1 Définition d'une base de détection

Pour définir une base de détection, nous devons au préalable adapter la fonction de détection présentée en section 1.3.1. De manière classique, nous dirons qu'un profil p est détecté comme malveillant si son taux de similarité par rapport à un profil m, identifié comme malveillant, est supérieur à un seuil donné t.

Définition 35. *(Fonction de détection). Soit P un ensemble de profils et S un mesure de similarité définie sur P, c'est-à-dire $S : P \times P \mapsto [0,1]$. Soit $M \subset P$ un ensemble de profils malveillants. Pour un seuil donné $t \in [0,1]$, nous définissons une fonction de détection $\mathcal{D}_t : P \times \mathcal{M} \mapsto \{0,1\}$ comme suit :*

$$\forall (p,m) \in P \times \mathcal{M}, \mathcal{D}_t(p,m) = \begin{cases} 0 \text{ si } S(p,m) < t, \\ 1 \text{ si } S(p,m) \geq t. \end{cases}$$

On remarquera que $\mathcal{D}$ est en fait une fonction de trois paramètres (t,p,m). Afin de simplifier la lecture, nous emploierons uniquement la notation indexée par t. De fait, toutes les définitions de cette section sont données pour valeur de t fixée. L'influence de ce seuil de similarité sera évaluée plus loin.

À partir d'une telle fonction de détection, nous définissons alors une base de détection $\mathcal{A}_t$ relativement à un ensemble de profils malveillants $\mathcal{M}$ comme un

sous-ensemble de $\mathcal{M}$, qui permet la détection de tout élément de $\mathcal{M}$ pour un t fixé.

Définition 36. *(Base de détection). Soit $\mathcal{D}_t$ une fonction de détection comme présentée en définition 35. Nous disons qu'un ensemble $\mathcal{A}_t \subset \mathcal{M}$ est une base de détection de $\mathcal{M}$ si*

$$\forall p \in \mathcal{M}, \exists a \in \mathcal{A}_t, \mathcal{D}_t(p, a) = 1$$

Nous considérons l'ensemble des bases de détection possibles de $\mathcal{M}$ en accord avec la définition 36. Cet ensemble contient $\mathcal{M}$ lui-même comme élément maximal par rapport à l'inclusion. Il contient aussi un élément minimal, qui correspond à la base de détection de $\mathcal{M}$ de taille minimale. Cet élément minimale est précisément la base de détection optimale recherchée. En effet, cette base possède le minimum de profils permettant la détection de tous les éléments malveillants considérés ($\mathcal{M}$). Elle offre donc les meilleurs performances en termes de temps de détection.

5.2.2 Détermination de la base de détection optimale

Nous cherchons ici à déterminer la base de détection de taille minimale pour un ensemble de profils malveillants $\mathcal{M}$. À partir des notations et définitions précédentes, nous sommes amenés à rechercher un sous ensemble de $\mathcal{M}$ noté $\mathcal{B}_t$ vérifiant :

$$\mathcal{B}_t = \min_{\mathcal{A}_t \subset \mathcal{M}} \{|\mathcal{A}_t|, \forall p \in \mathcal{M}, \exists a \in \mathcal{A}_t, \mathcal{D}_t(p, a) = 1\} \tag{5.1}$$

Malheureusement, la détermination de $\mathcal{B}_t$ est une tâche difficile.

Proposition 4. *(Complexité de la détermination d'une base de détection optimale). Déterminer $\mathcal{B}_t$ comme définie (5.1) est équivalent au problème du « Minimum Set Cover » (MSC), connu pour être NP-difficile [90].*

Preuve. *Soit $U = [\![1, n]\!]$ et $\mathcal{M} = (x_i)_{1 \leq i \leq n}$, nous définissons une matrice $D = (d_{ij})_{1 \leq i,j \leq n}$ telle que $\forall (i, j) \in U^2, d_{i,j} = \mathcal{D}_t(x_i, x_j)$. Nous notons $(S_i)_{1 \leq i \leq n}$ une famille de sous-ensembles de U telle que $\forall i \in U, S_i = \{j \in U / d_{ij=1}$. Maintenant, $\forall \mathcal{A}_t \subset \mathcal{M}, \exists J \subset U$ tel quel $\mathcal{A}_t = \bigcup_{j \in J} x_j$.*

$$\begin{aligned}
(A_t \text{ est une base de détection de } \mathcal{M}) &\Leftrightarrow (\forall p \in \mathcal{M}, \exists a \in \mathcal{A}_t, \mathcal{D}_t(p, a) = 1) \\
&\Leftrightarrow (\forall i \in U, \exists j \in J, d_{ij} = 1) \\
&\Leftrightarrow \left(\bigcup_{j \in J} S_j = U \right) \\
&\Leftrightarrow (\{S_j\}_{j \in J} \text{ couvre } U)
\end{aligned}$$

Nous avons donc prouvé que déterminer une base de détection de $\mathcal{M}$ est équivalent à trouver un ensemble couvrant U étant donné une famille (S_i) de sous-ensembles de U. Comme $\mathcal{B}_t$ est la base de détection de taille minimale pour $\mathcal{M}$, sa détermination est équivalente au problème MSC. □

Choix 5 (Approximation de la base de détection optimale).

Afin d'approximer la base de détection optimale $\mathcal{B}_t$, nous utilisons un algorithme glouton classique [90] décrit en figure 5.1.

Cet algorithme présente l'avantage d'avoir une complexité temporelle polynomiale.

```
1  input : Famille F = S_1, S_2, ..., S_n de sous-ensembles d'un ensemble fini U .
2  output :  J ⊆ [[1, n]] tel que ⋃_{j∈J} S_j = U .
3  X := U ;
4  J := ∅ ;
5  while X ≠ ∅ do
6     choose S_j = max_{S_i∈F}{|S_i ∩ X|};
7     X := X\S_j ;
8     J := J ∪ j ;
9  end (while) ;
10 output J ;
```

Listing 5.1 – Algorithme glouton pour le problème du MSC.

Le principe de cet algorithme est le suivant : il prend en entrée (ligne 1), une famille $\mathcal{F}$ de sous-ensembles $(S_i)_{i\in[\![i,n]\!]}$ d'un ensemble fini U. Partant de U (ligne 3), à chaque itération, l'élément S_j de $\mathcal{F}$ qui maximise la couverture de l'ensemble courant (ligne 6) est sélectionné. L'ensemble courant désigne U privé des S_i préalablement sélectionnés par l'algorithme (représenté en ligne 7). L'algorithme termine lorsque U est totalement couvert et retourne l'ensemble J des indexes correspondants aux S_j sélectionnés (en ligne 2).

Pour simplifier, nous appelons $\mathcal{B}_t$ la base de détection optimale de $\mathcal{M}$. De manière similaire, nous appelons base de détection de référence de $\mathcal{M}$, l'approximation de $\mathcal{B}_t$, notée $\widetilde{\mathcal{B}}_t$, obtenue par l'algorithme glouton.

5.3 Évaluation de l'approche

L'objectif de cette section est d'évaluer expérimentalement notre approche. Cette évaluation revient à répondre aux trois questions suivantes :

1. quelle mesure de similarité est la plus adaptée à la détection comportementale de codes malveillants parmi celles présentées au chapitre 4 : NCS ou CMID ?
2. quel est l'impact du nombre de profils d'applications malveillantes dans le base de détection sur les résultats obtenus ?
3. quel est le seuil de similarité t permettant d'obtenir les meilleurs résultats de détection ?

5.3.1 Choix d'un algorithme de compression pour les mesures de similarité

Les deux mesures de similarité présentées au chapitre 4 reposent sur la propriété de *normalité* d'un algorithme de compression (voir théorème 16 et proposition 3). De plus, d'un point de vue pratique, l'efficacité d'un compresseur impacte directement la précision de la similarité associée. L'outil Complearn [39] propose à ce titre différents algorithmes classiques afin de calculer la distance NCD : gzip, bzip2 et PPMZ.

- gzip est un logiciel de compression dont le format de fichier est spécifié dans le RFC1952. Ce logiciel repose sur la bibliothèque ZLIB (RFC1950) qui implémente la méthode de compression DEFLATE (RFC1951). DEFLATE utilise l'algorithme LZ77 [193] suivi d'un codage de Huffman [85]. LZ77 est un algorithme de compression par dictionnaire utilisant une fenêtre de taille fixe ;
- le compresseur bzip2 utilise la transformée de Burrows-Wheeler [24] suivie d'un codage de Huffman ;
- le compresseur PPMZ [16] est un algorithme de prédiction par correspondance partielle (« *Prediction by Partial Matching* » ou PPM) qui appartient à la famille des compresseurs sans perte, statistiques et adaptatifs.

Le choix d'un algorithme de compression est guidé par un besoin spécifique en fonction de ses caractéristiques [26]. Dans notre cas, il convient de vérifier la propriété de *normalité* pour les compresseurs considérés. Le compresseur gzip utilise une fenêtre de compression de 32 KiB alors que la taille des blocs utilisés par bzip2 est limitée à 900 KiB. PPMZ ne présente ni limite de taille de fenêtre, ni limite de taille de bloc. Cependant, il s'avère beaucoup plus lent que les autres (voir le tableau 5.1 pour un comparatif des performances).

Choix 6 (Algorithme de compression).

De manière pratique nous utilisons l'algorithme de compression « Lempel-Ziv-Markov chain-Algorithm » *(LZMA) comme compromis acceptable entre la taille des blocs et la vitesse de compression.*

Sans rentrer dans les détails de l'algorithme, il s'agit d'une amélioration et d'une optimisation de LZ77 qui permet de sélectionner des tailles de bloc jusqu'à 128 MiB tout en conservant une vitesse de compression correcte.

Afin de résumer les performances de ces différents compresseurs, le tableau 5.1 montre les résultats obtenus sur un corpus de 5 000 profils de codes malveillants, chaque échantillon étant compressé 100 fois. La somme des tailles des profils considérés atteint 32 975 375 octets. Tous ces tests ont été conduits sur un DELL precision 690 équipé d'un processeur Intel Quad-Core Xeon cadencés à 2,66 GHz et comprenant 4 Go de RAM. Le système d'exploitation est un Windows 7 64 bits version Entreprise. Seules les bibliothèques bzlib2 et LZMA offrent des options de compression : bzlib2 est configuré avec une taille de blocs maximale (900 KiB) et LZMA est utilisé avec les options par défaut, si ce n'est la

taille des blocs qui est définie en fonction de la taille des données à compresser. Toutes les bibliothèques de compression sont compilées pour produire des exécutables 32 bits via le compilateur de Visual Studio 2008, `cl.exe` version 15.00.30729.01 avec des options d'optimisation de vitesse (`/O2` et `/Ot`).

	zlib 1.2.4	bzlib2 1.0.5	LZMA 4.65	PPMZ2 v0.81
Vitesse de compression (KiB/s)	30 060	4 333	4 039	313
Taux de compression	81,04%	78,07%	81,49%	83,19%
Fenêtre/taille de bloc maximale	32 KiB	800 KiB	128 MiB	∞

TABLE 5.1 – Résultats de compression pour différentes bibliothèques (zlib, bzlib2, LZMA et PPMZ2).

Revenons maintenant sur la *normalité* d'un compresseur. La propriété la plus importante à nos yeux est l'idempotence puisqu'il apparait nécessaire de déterminer si un code malveillant déjà identifié sera bien reconnu comme similaire à lui même. C'est pour cette raison que nous nous focalisons sur l'auto-similarité, c'est-à-dire, la similarité d'un profil avec lui-même. Cette mesure d'auto-similarité nous donne, de plus, une borne maximale sur le seuil de similarité t atteignable. En effet, pour un seuil trop élevé, nous serions dans l'incapacité de déterminer pour deux fichiers identiques s'ils sont détectés. Par conséquent, la figure 5.4 présente la distribution d'auto-similarité pour les trois algorithmes de compression considérés (zlib, bzlib2 et LZMA) appliqués à chacun de nos profils malveillants. Nous écartons volontairement la bibliothèque PPMZ2 à cause de sa vitesse de compression plus de 10 fois plus lentes que les autres.

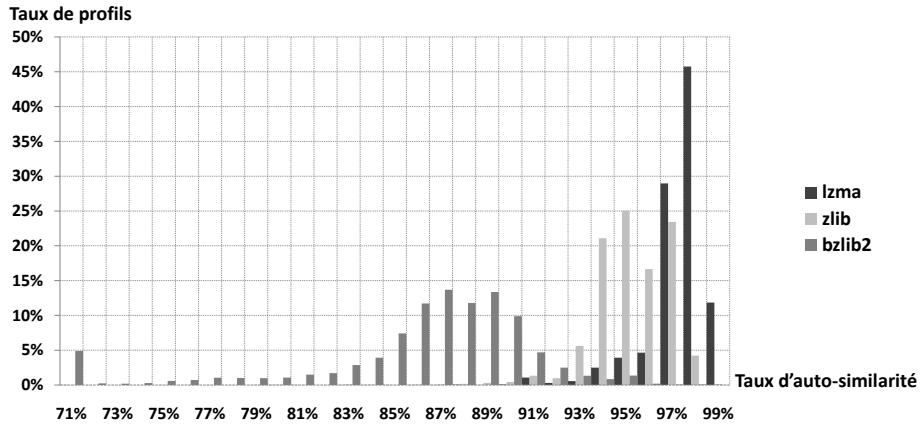


FIGURE 5.4 – Distribution des taux d'auto-similarité obtenus avec un échantillon de 5 000 profils malveillants.

Cette figure nous conforte dans notre choix d'utiliser LZMA comme algorithme de compression pour notre mesure de similarité. Concernant bzlib2, la

distribution est centrée sur 88%, ce qui est dû à une charge supplémentaire dans la région des 50 octets. Pour LZMA et zlib, les distributions sont respectivement centrées sur 98% et 96%. D'après la figure 5.4, si nous voulons conserver 90% des profils, le seuil de similarité acceptable peut atteindre 96% pour LZMA alors qu'il est limité à 94% pour zlib et 72% pour bzlib2.

5.3.2 Collecte des profils comportementaux

Afin d'évaluer notre approche, nous devons constituer notre base de détection telle que définie en section 5.2. Pour cela, nous collectons des profils à caractères malveillants dont une partie servira comme base de détection. L'autre partie constituera un panel de profils de test, c'est-à-dire des profils inconnus du système de détection, qui permettra d'évaluation l'efficacité de l'approche en termes de faux négatifs. Afin d'évaluer les faux positifs, qui pourraient survenir en cas d'utilisation de logiciels légitimes, nous collectons aussi des profils d'applications inoffensives. Cette section présente de quelle manière ces deux types de profils sont collectés.

5.3.2.1 Plateforme de collecte

Pour des raisons de sécurité, nous utilisons un environnement de virtualisation reposant sur VirtualBox afin de collecter les profils obtenus et plus particulièrement ceux à caractère malveillant. L'architecture de notre plateforme est schématisée sur la figure 5.5. Nous avons choisi Windows XP (sans « service pack ») comme système d'exploitation invité afin d'offrir des vulnérabilités potentielles aux codes malveillants. Ce système d'exploitation a été configuré avec des services standards : compte d'accès à internet, un client mail et P2P. Nous utilisons un faux serveur DNS pour dérouter tout le trafic réseau vers une autre machine virtuelle afin d'observer les tentatives de connexion. Le fonctionnement de notre collecteur de profils comportementaux est le suivant :

1. la machine hôte envoie un programme à observer sur la machine d'exécution. Ce programme peut être un code malveillant ou un programme inoffensif en fonction du type de profil requis ;
2. le programme est monitoré au cours de son exécution dont la durée dépend de l'expérience considérée comme nous le décrivons dans les sections suivantes. Ce programme dispose alors de l'ensemble des services de la machine hôte dont le trafic est orienté vers la machine cible. Pour nos expériences, la machine cible ne comporte pas de service particulier.
3. Une fois l'exécution terminée, le profil comportemental obtenu est transmis à la machine hôte ;
4. la machine d'exécution est alors restaurée dans sa configuration initiale pour accueillir un nouveau programme comme défini en étape 1.

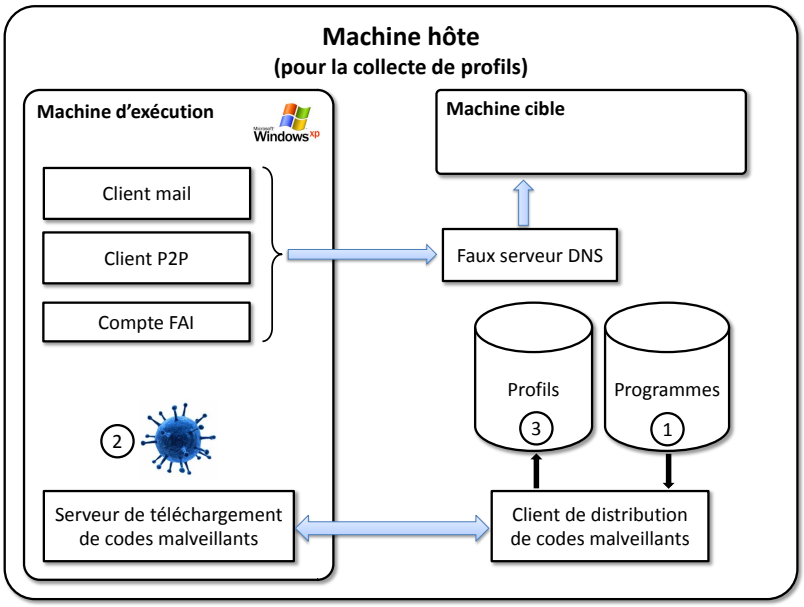


FIGURE 5.5 – Plateforme de collecte de codes malveillants.

5.3.2.2 profils d'applications malveillantes

Le comportement d'un code malveillant dépend directement de son environnement d'exécution. Même à environnement donné, le temps d'exécution d'un programme conditionne son profil comportemental.

Choix 7 (Durées d'exécution différentes).

Afin de prendre en compte l'impact du temps d'exécution sur les résultats de détection, chaque programme est exécuté deux fois :

1. *une première fois pendant une minute sur le système d'exploitation invité sans intervention particulière de l'utilisateur ;*
2. *une deuxième fois dans les mêmes conditions mais pendant une durée de deux minutes.*

Ces deux durées d'exécution nous permettent par ailleurs de nous placer dans des conditions pour lesquelles la limitation de NCD peut apparaître. Les profils obtenus après une minute sont utilisés comme base de détection alors que ceux obtenus après deux minutes sont utilisés pour évaluer les faux négatifs.

Nous reconnaissons que certains codes malveillants (par exemple certains chevaux de Troie) requièrent des actions de la part de l'utilisateur puisqu'ils imitent des programmes légitimes. Cet aspect là n'est pas pris en compte dans nos expériences. Même si des recherches préliminaires ont portées sur la détec-

tion des conditions temporelles de déclenchement [50], les autres outils d'analyse dynamique existants partagent cette limitation. Dans tous les cas, une fois le profil comportemental collecté, l'environnement virtuel est restauré dans sa configuration initiale afin de pouvoir exécuter un nouveau code malveillant. Ce processus est reconduit jusqu'à ce que les profils correspondant à tous les échantillons initiaux soient collectés.

Nos expérimentations sont conduites sur la collection librement disponible sur le site VX Heavens [171]. Même si cette base de codes malveillants est principalement composées de programmes assez anciens (jusqu'à 2006), son accessibilité représente un sérieux argument en termes de reproductibilité des expériences menées. Cette collection comprend 35 471 fichiers exécutables au format PE correspondant à des codes malveillants connus et à des kits de générations de code. Parmi ces binaires, nous avons obtenus 22 539 profils comportementaux. Les autres binaires ne présentent aucune activité, ce qui peut être expliqué de différentes manières : ils peuvent cibler un autre système d'exploitation ou un logiciel particulier qui n'est pas présent. Il se peut aussi que certains programmes nécessitent l'intervention de l'utilisateur. Conformément aux capacités du compresseur LZMA présenté en section 5.3.1, nous avons sélectionné des profils offrant un taux d'auto-similarité supérieur à 96%. Ce choix nous permet de faire varier le seuil de similarité t jusqu'à 96% tout en conservant 90% des profils. Finalement, nos expériences sont menées sur 5 000 profils comportementaux pour lesquels l'aspect malveillant a été manuellement vérifié. Cet ensemble de profils contient 54% de chevaux de Troie, 26% de portes dérobées, 13% de vers et 7% de virus d'après la classification fournie par l'anti-virus Kaspersky.

5.3.2.3 Profils d'applications inoffensives

Afin d'obtenir des profils inoffensifs, nous considérons deux types d'applications différentes : les installateurs d'applications, ainsi que des logiciels directement exécutables (portables). Le cas des logiciels portables correspond à la plupart des codes malveillants pour lesquels l'installation ne nécessite aucune intervention de la part de l'utilisateur, si ce n'est souvent le lancement initial. Le choix des installateurs d'applications se justifie par la ressemblance en termes de comportement avec un code malveillant. En effet, dans ce cas, de nouvelles bibliothèques, des exécutables et même des drivers peuvent être installés dans des répertoires sensibles tels que le répertoire système. Certains de ces fichiers sont enregistrés afin d'être lancés automatiquement au démarrage du système d'exploitation. Parfois, des connexions à Internet sont nécessaires pour des mises à jour ou des enregistrements d'applications. Dans des cas particuliers, une version plus récente du système d'exploitation est utilisée pour pouvoir mener à bien l'installation du logiciel. Ce processus a été conduit sur 200 applications utilisées pour l'évaluation des faux positifs. Tous ces profils d'application légitimes présentent un taux d'auto-similarité supérieur à 96% conformément aux conditions de sélection des codes malveillants.

5.3.3 Résultats expérimentaux

Nous rappelons que l'évaluation de notre approche consiste à répondre à trois interrogations portant sur : le choix d'une mesure de similarités, l'impact nombre d'éléments dans le base de détection et enfin le choix d'un seuil de similarité optimal.

Pour ce faire, nous utilisons une technique de validation croisée en cinq sous ensembles (« *five-fold cross-validation* » [98]). Notre ensemble de profils d'applications malveillantes est alors partitionné aléatoirement en cinq ensembles disjoints de tailles égales. Parmi ces cinq ensembles, un est sélectionné en tant que corpus de test pour l'évaluation des faux négatifs. Les quatre restants sont combinés afin de construire la base de détection associée. Les faux positifs sont évalués au moyen des 200 profils d'applications inoffensives. Nos résultats correspondent alors à la moyenne des cinq évaluations.

Afin de répondre au trois questions posées en début de section 5.3, nous présentons maintenant une analyse de caractéristique de fonctionnement du récepteur (« *Receiver Operating Characteristic* » ou ROC). Cette analyse est menée pour les deux mesures de similarité proposées (NCS et CMID) afin de comparer leurs performances respectives. Elle est aussi conduite pour différentes bases de détection afin d'évaluer l'impact du nombre de profils dans la base de détection sur les résultats obtenus. La première base de détection est celle de référence $\widetilde{\mathcal{B}}$ présentée en section 5.2.2. La seconde base de détection considérée contient tous les profils malveillants comme éléments pour la détection. L'analyse ROC nous permettra ainsi de déterminer les paramètres optimaux de notre détecteur.

5.3.3.1 Résultats pour des codes malveillants connus

Certains codes malveillants peuvent présenter des variations au niveau de leurs profils comportementaux en fonction de leur contexte d'exécution. Par exemple, ils peuvent générer des noms de fichier en fonction de leur environnement ou encore de manière aléatoire au cours de leur installation. Ils peuvent aussi se comporter différemment en fonction du niveau de privilège de l'utilisateur, à savoir si le programme est lancé en tant qu'administrateur de la machine ou non.

Pour cette raison, la figure 5.6 présente des courbes ROC correspondant aux résultats de détection obtenus avec des codes malveillants issus de deux bases de détection différentes ($\mathcal{B}_t$ et $\mathcal{M}$). Dans ce cas, les codes malveillants sont considérés comme connus (compris dans la base de détection) mais issus de contextes d'exécution différents (ici le temps d'exécution). Il est important de remarquer que pour les courbes ROC obtenues avec la base de référence $\mathcal{B}_t$ ne sont ni des fonctions du taux de vrais positifs, ni même fonctions du taux de faux positifs. Ce résultat s'explique par l'approximation faite par l'algorithme glouton. En cas de solutions exactes pour notre base de référence, nous obtiendrions des courbes fonctions de du taux de faux positifs comme c'est le cas avec la base complète.

D'après cette figure, l'impact de la taille de la base de détection sur les résultats obtenus est faible. La base de référence $\mathcal{B}_t$ et la base complète $\mathcal{M}$

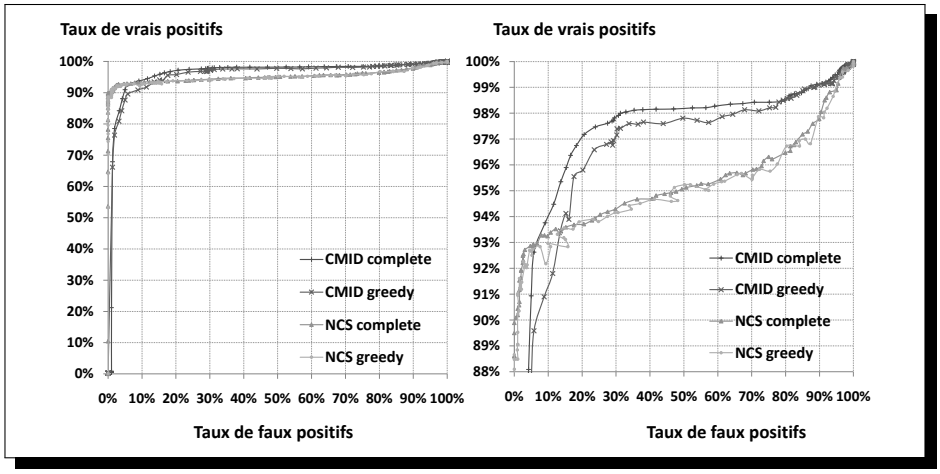


FIGURE 5.6 – Courbes ROC pour la détection de profils malveillants connus avec un temps d'exécution supérieur. À gauche les courbes entières. À droite, une vue détaillée.

présentent des résultats similaires en ce qui concerne NCS. Pour ce qui est de CMID, l'impact de la taille de la base de détection sur le taux de vrais positifs est au maximum inférieur à 3%.

En termes de précision de détection, la figure 5.6 illustre le compromis entre les vrais positifs et les faux positifs offerts par les deux mesures de similarité. Par exemple, si l'on considère la base de détection de référence, pour atteindre 93% de vrais positifs, le taux de faux positifs atteint les 85%. Alors que pour atteindre les 93% de vrais positifs, CMID présente un taux de faux positifs de moins de 15%. Les 97% de vrais positifs sont toutefois atteint pour un taux de 30% de faux positifs. Dans tous les cas, plus de 90% des codes malveillants sont détectés avec moins de 6% de faux positifs.

L'intérêt de ces premiers résultats est de montrer l'efficacité de l'approche capable de détecter des codes malveillants indépendamment des techniques d'obscurcissement de code statique employées. En effet, la collection offerte par VX Heaven contient des variantes de codes malveillants parmi lesquelles 40,1% sont compressées (« *packées* »), 47,2% correspondent à des PE classiques et 12,7% ne peuvent être classées comme compressées ou non par les outils PEiD et ProtectionID [152].

5.3.3.2 Résultats pour des codes malveillants inconnus

Les codes malveillants inconnus correspondent à notre ensemble de test, c'est-à-dire aux profils d'applications malveillantes non contenues dans le base de détection utilisée. La figure 5.7 présente une courbes ROC pour la détection de notre ensemble de codes malveillants ainsi que le pourcentage de profils contenus dans la base de détection de référence. D'après cette figure, le premier constat

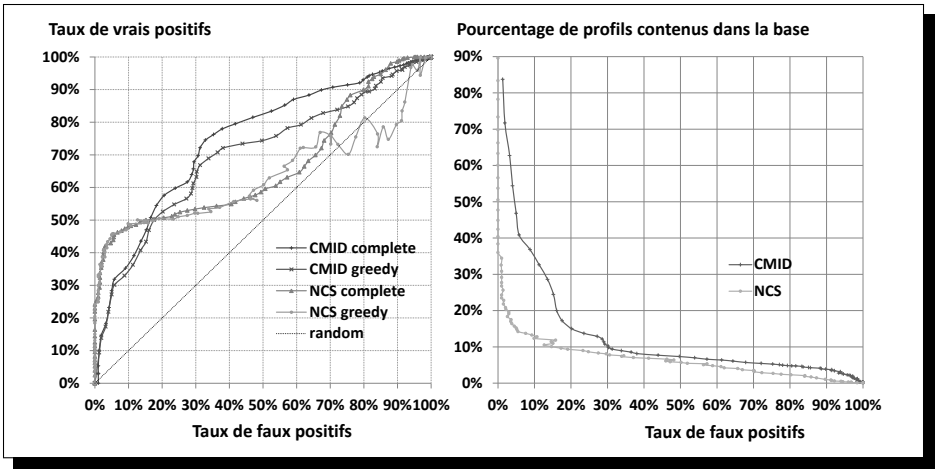


FIGURE 5.7 – Courbe ROC pour la détection de profils malveillants inconnus à gauche et taille de la base de détection de référence à droite.

est la **validation de la limitation de NCS concernant la détection de codes malveillants à partir de profils comportementaux**. En effet, en termes de surface sous la courbe (« *Area Under Curve* » ou AUC), CMID présente de meilleurs résultats que NCS.

Le second constat est l'impact de la taille de la base de détection sur les résultats obtenus. En d'autres termes, nous pouvons alors estimer l'impact de l'approximation obtenue par l'algorithme glouton sur la détection. Concernant CMID, l'impact sur le taux de vrais positifs est au maximum inférieur à 10%. Pour NCS, l'impact est encore moins significatif excepté à partir de 50% où la différence peut atteindre plus de 10%.

Le dernier point est la limitation de notre approche qui présente un taux élevé de faux positifs. En termes de vrais positifs, pour un taux de faux positifs inférieur à 16%, NCS apparait plus intéressante que CMID. Dans le cas contraire, CMID offre de meilleurs résultats. Malheureusement, ce taux de faux positifs de 16% correspond à un taux de vrais positifs de 50%, ce qui apparait comme inacceptable du point de vue de l'utilisateur final.

Nous attribuons ces résultats à deux origines :

1. le manque de précision du générateur de profils comportementaux. Par exemple, notre système de collecte de traces n'est actuellement pas capable de prendre en compte l'activité réseau à partir de la couche session du modèle OSI. De manière similaire, notre programme d'abstraction n'est pas en mesure d'identifier des noms générés de manière aléatoire. Comme mentionné dans [99], la figure 5.7 illustre la difficulté de distinction entre des programmes inoffensifs et malveillants sans une analyse précise de la dépendance des données entre appels systèmes ;
2. les profils comportementaux contenus dans la base de détection ne sont

pas suffisamment discriminants. En effet, pour un profil d'application malveillante, l'intégralité des actions observées ne correspond pas nécessairement au comportement malveillant. Seule une sous partie de ce profil caractérise l'action malveillante à proprement parlé, comme c'est par exemple le cas pour un virus infectant un logiciel légitime. En utilisant CMID sur le sous-profil caractérisant précisément l'action malveillante, nous estimons alors pouvoir diminuer le nombre de faux positifs. L'explication provient de l'inclusion potentielle de la partie non malveillante d'un profil de la base de détection dans une application inoffensive. L'effet observé est alors l'augmentation du degré d'inclusion du profil malveillant dans celui du logiciel analysé. Cette remarque ne concerne pas NCS, puisque le fait de diminuer la taille des profils d'applications malveillantes ne fera qu'augmenter l'éloignement entre un profil malveillant d'origine et son sous-profil caractéristique (voir section 4.3).

5.4 Bilan de la détection

Dans ce chapitre, nous avons présenté une approche de détection dynamique de codes malveillants. Le principe de notre détection repose sur le fait que des variantes d'un code malveillant déjà identifié partagent un fort taux de similarité comportementale avec ce dernier. Ainsi nous avons proposé de détecter des codes malveillants inconnus si leurs comportements apparaissent suffisamment proche de celui d'une application malveillante déjà identifiée. Le point essentiel consiste alors à déterminer le seuil de similarité offrant les résultats de détection optimaux en termes de *fiabilité* et de *pertinence*.

Dans un premier temps, nous avons présenté les composants essentiels de notre architecture :

- le générateur de profils comportementaux, qui à partir du binaire d'un programme est capable d'en extraire les actions exécutées sous la forme d'un profil comportemental. Cet élément permet de s'abstraire des transformations syntaxiques utilisées pour contourner la détection statique ;
- la construction d'une base de détection optimale, qui contient l'ensemble des profils de codes malveillants. Cette base contient idéalement l'ensemble minimal de profils permettant la détection d'un ensemble de codes malveillants connus $\mathcal{M}$, pour un seuil de similarité fixé en fonctions des caractéristiques opérationnelles désirées (*fiabilité* et *pertinence*) ;
- le comparateur de profils en charge du résultat de détection. Ce dernier mesure alors la similarité entre le profil du programme soumis à analyse et ceux de la base de détection pour fournir le résultat final.

Dans un deuxième temps, nous avons décrit les expériences menées dans le but d'évaluer les performances de notre approche. Ces expériences ont révélé les résultats suivants. Ils mettent en évidence les améliorations possibles de notre système :

1. le premier résultat concerne la taille de la base de détection utilisée qui n'a que peu d'impact sur les résultats obtenus. Pour un taux de faux

positifs inférieur à 16%, l'écart entre la base de détection de référence et celle contenant tous les profils à détecter est inférieur à 5% en termes de vrais positifs. Ainsi, la base de détection n'apparaît pas comme un élément d'amélioration prioritaire ;

2. le second résultat concerne l'intérêt de CMID par rapport à NCS qui offre de meilleurs résultats en termes d'AUC. Cette surface traduit la précision de notre détecteur confronté à des codes malveillants inconnus, c'est-à-dire non contenus dans la base de détection. Ce résultat confirme l'hypothèse du chapitre précédent selon laquelle la distance NCD [1] éloigne des éléments de tailles différentes. Cet aspect est aussi conforté par les travaux d'Apel *et al.*[3] qui exposent le manque de précision de NCD par rapport à d'autres métriques plus adaptées au cas des comportements de codes malveillants ;
3. le dernier résultat est la limitation de l'architecture actuelle qui présente des un taux de faux positifs trop important.

Perspective 4 (Amélioration des performances de détection).

Nous envisageons deux axes d'amélioration concernant nos performances de détection :
- *le premier consiste à augmenter la précision de notre générateur de profils comportementaux pour obtenir des rapports plus détaillés ;*
- *le deuxième consiste à extraire de tout profil, contenu dans la base de détection, le sous-profil correspondant à l'activité malveillante discriminante. Cette extraction permettrait alors d'augmenter les performances de détection en utilisant CMID comme mesure de similarité.*

1. Nous rappelons que cette limitation concerne aussi NCS, qui vaut 1-NCD.

Conclusion et perspectives

Dans ce mémoire, nous avons étudié une technique particulière employée par les codes malveillants évolutifs afin d'éviter leur détection : le métamorphisme. Contrairement aux autres techniques préalablement utilisées telles que le chiffrement de code ou encore le polymorphisme, cette technique de mutation permet à un programme métamorphe d'éviter la présence de tout motif statique facilement détectable à tout moment de son exécution. Pour entreprendre cette étude, nous sommes partis de l'étymologie de ce terme afin d'en explorer les différents aspects qui le caractérise, à savoir : les divers formalismes permettant de modéliser le métamorphisme, puis les techniques de mutation de code utilisées pour modifier la forme d'un programme et enfin les différentes approches de détection.

La première partie de cet état de l'art nous a conduit à proposer une définition générique du métamorphisme qui permet d'en unifier les précédentes au moyen d'une hiérarchisation du métamorphisme. Selon cette définition, un code métamorphe d'ordre 0 est un mot du langage $L(G)$ où G désigne une grammaire formelle décrivant les règles de transformations utilisées. Un code métamorphe d'ordre n est alors un mot du langage $L^n(G)$ défini récursivement par $L^n(G) = L(L^{n-1}(G))$. Les deux définitions du métamorphisme proposées jusqu'alors correspondent à l'ordre 0, pour la vision la plus simple, et à l'ordre 1 en ce qui concerne les définitions de Zuo *et al.* ainsi que de Filiol.

Les deux parties suivantes de l'état de l'art ont conduit à la problématique de cette thèse : la construction d'un moteur générique de métamorphisme à résilience prouvée en analyse statique, ainsi qu'une approche de détection dynamique originale s'appuyant sur la complexité de Kolmogorov. Nous exposons ici le bilan de notre étude ainsi que des perspectives pour des travaux à venir.

Bilan

Construction d'un moteur générique de métamorphisme

Dans la première partie de ce mémoire, nous avons présenté la conception d'un moteur générique de métamorphisme d'ordre 0. Notre approche est inspirée du fonctionnement en deux temps de ce type de codes malveillants :

1. dans un premier temps un code métamorphe M se modélise afin d'obtenir

son *archétype* A ;

2. dans un deuxième temps des transformations d'obscurcissement de code sont appliquées à A afin de produire une nouvelle forme mutée M' du programme M.

Notre présentation s'est alors logiquement articulée suivant ces deux étapes. Dans le chapitre 2, nous avons présenté une technique d'obscurcissement de code inspirée de celles utilisées dans le cadre de la protection intellectuelle des logiciels et dont la résilience est prouvée en analyse statique. Ce chapitre correspond à la deuxième étape du cycle d'auto-reproduction d'un programme métamorphe, à savoir, la mutation de son code. Afin de valider l'hypothèse que nous avons faite, selon laquelle les anti-virus actuels utilisent essentiellement des approches statiques, un code métamorphe connu a fait l'objet de modifications au cours de plusieurs expériences. Les résultats obtenus montrent que cette hypothèse est correcte pour 31 des 32 anti-virus testés pour la souche virale considérée.

Dans le chapitre 3, nous avons exposé le fonctionnement global de notre moteur de métamorphisme et plus particulièrement comment l'obtention de l'*archétype* est possible sans pour autant faciliter la détection d'un tel programme. Ce chapitre correspond à la première étape du cycle d'auto-reproduction d'un programme métamorphe : sa modélisation. D'un point de vue implémentation, ce moteur se présente sous la forme d'un ensemble de fichiers sources écrits en langage C. Intégré à une chaîne de compilation particulière, il permet de transformer un programme quelconque (écrit lui aussi en langage C) en un code métamorphe en fournissant une fonction de réplication ainsi qu'une fonction de protection et de dé-protection des données. Ce chapitre apporte ensuite une contribution dans la méthodologie d'évaluation des produits de détection des codes malveillants. En effet, l'application de notre moteur à un ver connu (MyDoom) nous a permis d'observer en « boîte noire » les différentes techniques de détection utilisées par les anti-virus actuels.

Application de la complexité de Kolmogorov à la détection des codes malveillants

Dans la seconde partie de ce mémoire, nous nous sommes intéressés à la détection des codes malveillants métamorphes.

Dans le chapitre 4, nous avons présenté la complexité de Kolmogorov ainsi que les différentes mesures de similarité qui en sont issues. L'avantage de ces mesures est de pouvoir évaluer la similarité entre des objets à partir de l'information qu'ils contiennent. Il s'agit donc de mesures universelles. La première mesure, NCS, est obtenue au moyen d'un compresseur sans perte. Elle a déjà fait ses preuves dans de nombreux domaines et notamment dans le cadre de l'analyse et de la classification de codes malveillants. Dans ce chapitre, nous avons toutefois identifié une limitation propre à NCS. Pour cette raison, une nouvelle mesure de similarité fondée elle aussi sur la compression a été proposée. Cette nouvelle mesure, CMID, correspond à un degré d'inclusion de l'information contenue dans un objet dans un autre objet.

Dans le chapitre 5, nous avons proposé une approche de détection dynamique s'appuyant sur les deux mesures de similarité du chapitre précédent. Les interactions d'un programme en cours d'exécution avec son environnement sont collectées pour constituer un profil comportemental. À partir d'une base de détection composée de plusieurs profils comportementaux de codes malveillants, nous détectons un code malveillant inconnu si son comportement apparait comme suffisamment similaire à l'un des profils contenu dans la base. Les résultats obtenus nous paraissent prometteurs même si des améliorations sont nécessaires afin d'augmenter les performances de détection.

Les principales contributions obtenues dans cette étude du métamorphisme sont illustrées en gris sur la figure 5.8.

Perspectives

Au regard du travail réalisé, plusieurs perspectives ont été identifiées. Nous les présentons dans l'ordre correspondant à l'organisation de ce mémoire :

1. l'étude du métamorphisme d'ordre supérieur à 0 ;
2. l'amélioration du moteur de métamorphisme proposé ;
3. l'application de CMID à d'autres domaines que celui de la détection des codes malveillants ;
4. l'amélioration de l'architecture de détection proposée.

Nous détaillons chacune de ces perspectives ci-après.

Étude du métamorphisme d'ordres strictement positifs

Le métamorphisme reste aujourd'hui encore une technique peu étudiée sur le plan théorique comme en témoigne le peu de définitions formelles existantes. En effet, conformément à la définition que nous proposons, seule la complexité de détection du métamorphisme d'ordre 0 est connue. Cette dernière dépend du type de grammaire formelle employée pour produire les mutations de code conformément à la classification de Chomsky. La détection d'un code métamorphe d'ordre 0 est alors indécidable pour les grammaires de type 0, NP-complète pour les grammaires de type 1 et 2, et enfin polynomiale pour celles de type 3.

Pour la définition proposée par Filiol, à savoir le métamorphisme d'ordre 1, la complexité de la détection associée n'est pas définie. Une première approche pour aborder l'ordre 1 réside dans l'étude des grammaires de Van Wijngaarden [163]. Ces grammaires qui peuvent être vues comme la composition de deux grammaires sans-contextes (type 2 de la classification de Chomsky). La première est utilisée pour générer un ensemble de symboles terminaux qui correspondent aux symboles non-terminaux de la seconde grammaire sans-contexte. Les résultats connus pour ce type de grammaires sont dûs à Sintzoff [155] et à Wijngaarden [162] qui ont démontré qu'elles peuvent simuler une machine de Turing. Elles sont donc aussi expressives que des grammaires de type 0 de la classification de Chomsky. De plus, ces grammaires présentent l'avantage d'être plus

facile à concevoir que des grammaires de type 0 [78]. Elles pourraient donc servir de base à la conception de codes malveillants métamorphes dont la détection serait indécidable.

Amélioration du moteur métamorphisme

Dans ce mémoire, nous avons fait le choix d'un moteur de métamorphisme d'ordre 0. Le modèle théorique qui nous a servi à prouver la résilience des mutations de code employées repose sur la difficulté de la détermination précise des alias dans le cadre de l'analyse statique en présence de prédicats opaques. Plutôt que de devoir générer de tels prédicats, nous avons tiré avantage du τ-obscurcissement de code pour le calcul des transitions entre blocs de code. Ce choix correspond à notre modèle de $k-$obscurcissement de code pour lequel la difficulté de détection statique est prouvée. Nos expériences montrent aussi que ce choix permet de contourner la détection dynamique en différant les actions menées par le programme métamorphe au-delà du temps imparti pour sa détection. La contrepartie de ce type d'obscurcissement est l'augmentation du temps d'exécution pour mener à bien l'action attendue.

Afin d'éviter cette contrainte, d'autres approches d'obscurcissement de code sont envisageables pour la conception d'un moteur de métamorphisme d'ordre 0. Par exemple, l'utilisation d'une table d'interprétation est reconnue comme une techniques efficace en termes de résilience [44]. Le principe consiste à convertir le code d'un programme dans un « *bytecode* » particulier qui constitue une représentation intermédiaire. Ce « *bytecode* » est ensuite interprété par une machine virtuelle associée en charge de l'exécution du code. Selon cette approche, un programme métamorphe doit alors produire une nouvelle machine virtuelle ainsi que son « *bytecode* » associé à chaque réplication. Pour cela, il doit transposer son propre code ainsi que celui du programme sur lequel il est appliqué. La difficulté consiste alors à concevoir les règles de production d'un nouveau « *bytecode* » ainsi que de son CPU logiciel correspondant. De plus, il convient aussi de s'assurer que les interpréteurs générés sont suffisamment différents d'un point de vue syntaxique pour ne pas être détecté. La réalisation d'un tel programme auto-reproducteur reste à faire.

Par ailleurs, le métamorphisme ne se limite pas uniquement à des mutations d'ordre syntaxique. Les travaux de Jacob *et al.* [66, 89] ont ainsi proposé un moteur de polymorphisme fonctionnel. Son fonctionnement s'appuie sur des grammaires attribuées [97] afin de réaliser des mutations au niveau comportemental. Le principe consiste à transposer des règles de réécriture du niveau syntaxique au niveau fonctionnel en remarquant qu'une action de « haut niveau » sur un système d'exploitation peut être réalisée de différentes façons. Pour la réalisation d'un moteur de métamorphisme fonctionnel, des mutations syntaxiques doivent aussi être employées afin de modifier la forme du programme en plus de ses fonctionnalités. La conception d'un moteur de métamorphisme fonctionnel est à ce jour un problème ouvert.

Étude et applications du degré d'inclusion mutuel par compression

Les mesures de similarité présentées dans ce mémoire trouvent leurs fondements théoriques dans la complexité de Kolmogorov. Toutefois, leurs propriétés reposent directement sur l'algorithme de compression utilisé. En effet, NCD et CMID sont respectivement une distance et un degré d'inclusion à condition que le compresseur employé respecte la propriété de *normalité*. Outre le fait d'être normal, les caractéristiques de ces mesures dépendent directement des performances du compresseur utilisé [26]. Ainsi, les travaux d'Apel *et al.* [3] mettent en avant des limitations de NCD pour l'analyse de codes malveillants en termes de sensibilité et de performances. Pour ce qui est de CMID, nous avons illustré son intérêt au moyen de deux exemples simples : le premier dans le cadre de détection d'un plagiat, puis le second, pour la mise en évidence d'une violation de la propriété intellectuelle. L'utilité d'une telle mesure dans des domaines autres que celui de la détection de codes malveillants n'a pas été développée dans ce mémoire. Toutefois, les exemples évoqués précédemment nous confortent dans l'idée que cette mesure peut trouver des applications dans des domaines variés, pour lesquels NCD ne est pas la mesure la plus adéquate.

Amélioration de l'architecture de détection proposée

La dernière perspective que nous présentons concerne l'amélioration des performances de notre approche de détection dynamique fondée sur la similarité comportementale. Pour cela, il nous parait alors nécessaire de scinder l'obtention des profils comportementaux en deux parties, ce que nous ne faisons pas actuellement :

1. dans une première partie, un environnement dédié peut être utilisé afin d'obtenir les profils comportementaux de codes malveillants pour la constitution de la base de détection. L'emploi d'un environnement dédié permet d'obtenir la couverture comportementale la plus complète possible d'un programme. Pour cela, plusieurs outils ont déjà été proposés pour l'aide à l'analyse dynamique de codes malveillants (voir annexe C). Les travaux de Moser *et al.* [129] permettent ainsi d'explorer plusieurs chemins d'exécution et donc de révéler les différents comportements possibles d'un code malveillant indépendamment de son contexte d'exécution. Une telle approche permet par exemple de déceler le comportement d'une bombe logique. À défaut d'une telle architecture d'analyse multi-chemins, l'ajout d'une plateforme de collecte comme Nepenthes [5], Amun [71] ou encore HoneyClients [177] offrant des services fictifs vulnérables permettrait d'augmenter l'activité des codes malveillants observés. Des environnements complets d'analyse peuvent aussi convenir après conversion des rapports obtenus en profils compatibles avec notre approche. Parmi ces outils les principaux sont Anubis [11], CWSandbox [183], Norman Sandbox [134], Joebox [22], etc. Dans tous les cas, les profils obtenus à partir de codes malveillants doivent être les plus discriminants possibles par rap-

port à des application bénignes comme ce qui est fait dans les travaux de Fredrikson *et al.* [68]. En effet, nous avons observé que de nombreux codes malveillants utilisent des applications légitimes comme par exemple une console (cmd.exe), ou encore un navigateur Internet afin de mener leurs actions nuisibles. Or, une partie de l'activité inoffensive de ces applications se retrouve alors dans le profil comportemental généré, comme par exemple l'écriture de certaines clés de registre ou la création de fichiers temporaires. C'est précisément cette partie qui doit être supprimée du profil du code malveillant afin de diminuer sa similarité avec l'application légitime initiale et donc limiter le nombre de faux positifs ;

2. dans une deuxième partie, la génération des profils comportementaux sur un poste utilisateur peut être moins expressive et complète que celle mise en place pour la base de détection. Nous estimons que ce choix permettra d'obtenir un compromis acceptable entre la facilité de déploiement et les performances opérationnelles de détection. L'utilisation d'un détecteur minimaliste permet de compléter les outils de détections actuels, et notamment ceux d'analyse statique, en évitant la mise en place d'architectures complexes dédiées. Toutefois, il nous semble quand même utile d'améliorer la précision de notre générateur de profils comportementaux.

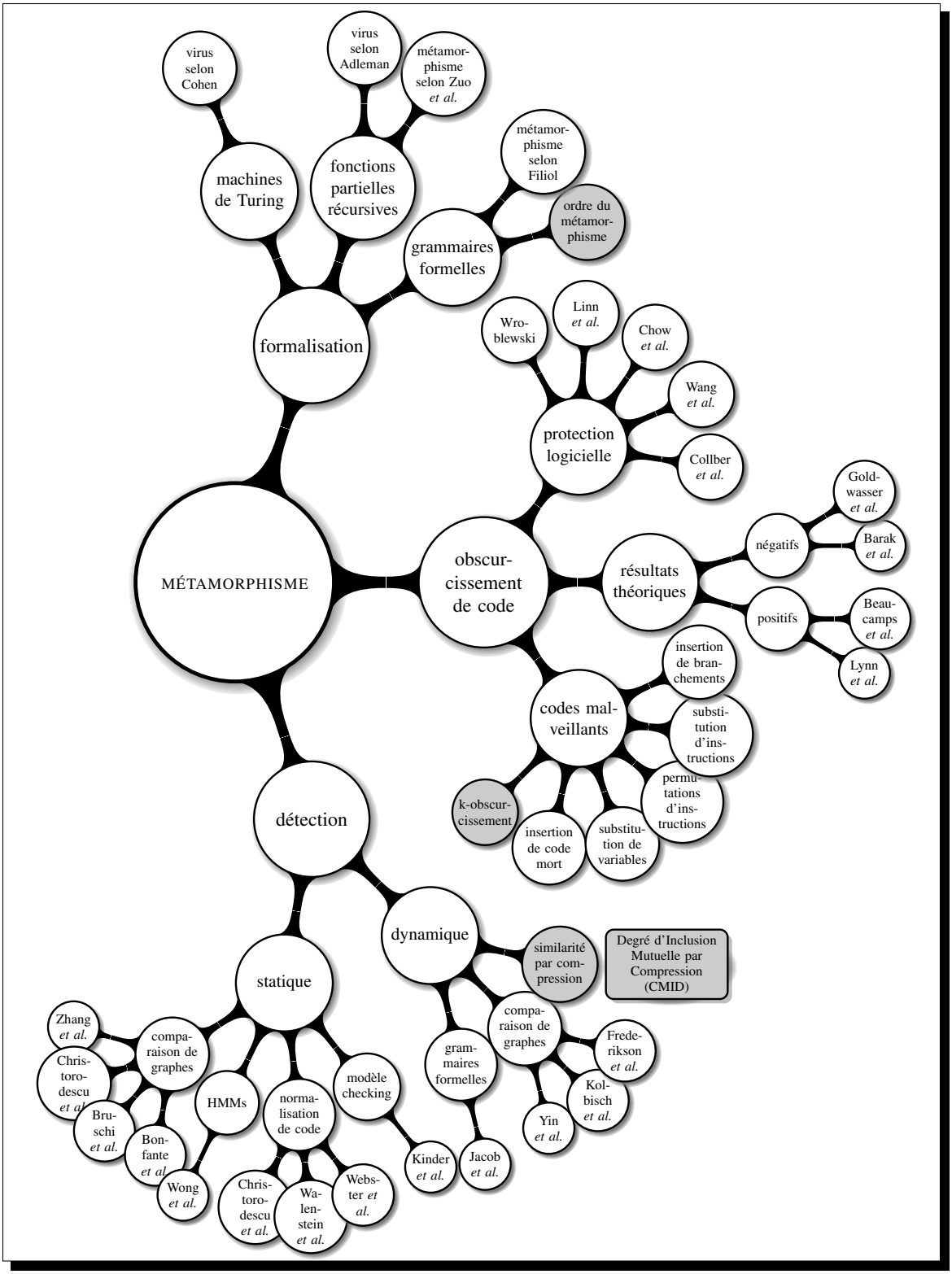


FIGURE 5.8 – Illustration des contributions de ce mémoire représentées en gris.

Troisième partie

Annexes

Annexe A

Existence des virus métamorphes à infinité de formes

Les travaux de Zuo *et al.* [195, 196], présentés en section 1.1.2.3, définissent des virus polymorphes à deux formes comme une pair $\{v, v'\}$ de fonction récursives totales partageant un même noyau (voir définition 6 page 8). Ils étendent ensuite cette définition aux cas de virus polymorphes à infinité de formes dont ils démontrent l'existence au moyen des théorèmes d'itérations et de récursion de Kleene (voir définition 7 page 8). Zuo *et al.* proposent ensuite la définition d'un virus métamorphe comme une pair $\{v, v'\}$ de fonction récursives totales mais avec cette fois si leurs noyaux respectifs (voir définition 8 page 9). Nous introduisons ici la définition de virus métamorphes à infinité de formes, chaque forme disposant de son propre noyau.

Proposition 5. *(Virus métamorphes à infinité de formes). Une fonction récursive totale $v(n, x)$ est un virus métamorphe de noyaux (T_n, I_n, D_n, S_n) à infinité de formes si pour tout (n, x) et pour tout environnement (d, p),*

$$\phi_{v(n,x)}(d,p) = \begin{cases} D_n(d,p), & si\ T_n(d,p) \\ \phi_x(d, p[v(n+1, \underline{S_n(p)})]), & si\ I_n(d,p) \\ \phi_x(d,p), & sinon \end{cases} \tag{A.1}$$

Preuve. *(Virus metamorphes à infinité de formes). On considère une fonction récursive totale $b(m, i, k)$ telle que :*

$$\phi_{b(m,i,k)}(d,p) = \begin{cases} \langle m, d, p\rangle, & si\ \langle m, d, p\rangle \equiv 0\ [3] \\ \phi_i(d, p[\phi_k(m+1, \underline{S(m,p)})]), & si\ \langle m, d, p\rangle \equiv 1\ [3] \\ \phi_i(d,p), & sinon \end{cases}$$

Le théorème s-m-n (dit aussi théorème d'itération de Kleene) appliqué à $b(m, i, k)$ nous donne l'existence d'une fonction f totale récursive telle que $\phi_{f(k)}(m, i) =$

$b(m,i,k)$. Le théorème de récursion de Kleene nous donne l'existence d'un entier n tel que $\phi_{f(n)} = \phi_n$. Soit $v(m,i) = b(m,i,n) = \phi_{f(n)}(m,i) = \phi_n(m,i)$, alors :

$$\begin{aligned}
\phi_{v(m,i)}(d,p) &= \phi_{b(m,i,n)}(d,p) \\
&= \begin{cases} \langle m,d,p\rangle, & si\ \langle m,d,p\rangle \equiv 0\ [3] \\ \phi_i(d,p[\phi_n(m+1,\underline{S(m,p)})]), & si\ \langle m,d,p\rangle \equiv 1\ [3] \\ \phi_i(d,p), & sinon \end{cases} \\
&= \begin{cases} \langle m,d,p\rangle, & si\ \langle m,d,p\rangle \equiv 0\ [3] \\ \phi_i(d,p[v(m+1,\underline{S(m,p)})]), & si\ \langle m,d,p\rangle \equiv 1\ [3] \\ \phi_i(d,p), & sinon \end{cases}
\end{aligned}$$

$\forall m \neq n, \forall i, \forall (d,p)$ tels que $T_m(d,p)$ soit satisfait, comme $\langle m,d,p\rangle \neq \langle n,d,p\rangle$, $\phi_{v(m,i)}(d,p) \neq \phi_{v(n,i)}(d,p)$. Pour deux indexes distincts m et n, les formes associés $v(m,i)$ et $v(n,i)$ sont bien distinctes aussi. Les prédicats récursifs T_m et D_m vérifient bien les conditions initiales, à savoir qu'ils ne sont jamais simultanément vrais et qu'il existe une infinité de couples (d,p) pour lesquels ils sont simultanément faux. □

Annexe B

Règles de réécritures du virus Win32.MetaPHOR

Dans cette annexe, nous exposons les règles de réécritures employées par le virus Win32.MetaPHOR aussi bien pour se modéliser que pour muter son code. Le tableau suivant présente des pseudo-instructions équivalentes.

Pseudo-instruction	**Pseudo-instruction équivalente**
XOR Reg,-1	NOT Reg
XOR Mem,-1	NOT Mem
MOV Reg,Reg	NOP
SUB Reg,Imm	ADD Reg,-Imm
SUB Mem,Imm	ADD Mem,-Imm
XOR Reg,0	MOV Reg,0
XOR Mem,0	MOV Mem,0
ADD Reg,0	NOP
ADD Mem,0	NOP
OR Reg,0	NOP
OR Mem,0	NOP
AND Reg,-1	NOP
AND Mem,-1	NOP
AND Reg,0	MOV Reg,0
AND Mem,0	MOV Mem,0
XOR Reg,Reg	MOV Reg,0
SUB Reg,Reg	MOV Reg,0
OR Reg,Reg	CMP Reg,0
AND Reg,Reg	CMP Reg,0
TEST Reg,Reg	CMP Reg,0
LEA Reg,[Imm]	MOV Reg,Imm
LEA Reg,[Reg+Imm]	ADD Reg,Imm
LEA Reg,[Reg2]	MOV Reg,Reg2
LEA Reg,[Reg+Reg2]	ADD Reg,Reg2
LEA Reg,[Reg2+Reg2+xxx]	LEA Reg,[2*Reg2+xxx]
MOV Reg,Reg	NOP
MOV Mem,Mem	NOP

Le tableau suivant présente des équivalences entre deux pseudo-instructions (colonne de gauche) et une pseudo-instruction (colonne de droite).

Suite de 2 pseudo-instructions	**Pseudo-instruction équivalente**
`PUSH Imm` `POP Reg`	`MOV Reg,Imm`
`PUSH Imm` `POP Mem`	`MOV Mem,Imm`
`PUSH Reg` `POP Reg2`	`MOV Reg2,Reg`
`PUSH Reg` `POP Mem`	`MOV Mem,Reg`
`PUSH Mem` `POP Reg`	`MOV Reg,Mem`
`PUSH Mem` `POP Mem2`	`MOV Mem2,Mem` (codée avec le pseudo-opcode `4F`)
`MOV Mem,Reg` `PUSH Mem`	`PUSH Reg`
`POP Mem` `MOV Reg,Mem`	`POP Reg`
`POP Mem2` `MOV Mem,Mem2`	`POP Mem`
`MOV Mem,Reg` `MOV Reg2,Mem`	`MOV Reg2,Reg`
`MOV Mem,Imm` `PUSH Mem`	`PUSH Imm`
`MOV Mem,Imm` `OP Reg,Mem`	`OP Reg,Imm`
`MOV Reg,Imm` `ADD Reg,Reg2`	`LEA Reg,[Reg2+Imm]`
`MOV Reg,Reg2` `ADD Reg,Imm`	`LEA Reg,[Reg2+Imm]`
`MOV Reg,Reg2` `ADD Reg,Reg3`	`LEA Reg,[Reg2+Reg3]`
`ADD Reg,Imm` `ADD Reg,Reg2`	`LEA Reg,[Reg+Reg2+Imm]`
`ADD Reg,Reg2` `ADD Reg,Imm`	`LEA Reg,[Reg+Reg2+Imm]`
`OP Reg,Imm` `OP Reg,Imm2`	`OP Reg,(Imm OP Imm2)` (l'opération doit être calculée)
`OP Mem,Imm` `OP Mem,Imm2`	`OP Mem,(Imm OP Imm2)` (l'opération doit être calculée)
`LEA Reg,[Reg2+Imm]` `ADD Reg,Reg3`	`LEA Reg,[Reg2+Reg3+Imm]`
`LEA Reg,[(RegX+)Reg2+Imm]` `ADD Reg,Reg2`	`LEA Reg,[(RegX+)2*Reg2+Imm]`
`POP Mem` `PUSH Mem`	`NOP`
`MOV Mem2,Mem` `MOV Mem3,Mem2`	`MOV Mem3,Mem`
`MOV Mem2,Mem` `OP Reg,Mem2`	`OP Reg,Mem`
`MOV Mem2,Mem` `MOV Mem2,xxx`	`MOV Mem2,xxx`
`MOV Mem,Reg` `CALL Mem`	`CALL Reg`
`MOV Mem,Reg` `JMP Mem`	`JMP Reg`
`MOV Mem2,Mem` `CALL Mem2`	`CALL Mem`
`MOV Mem2,Mem` `JMP Mem2`	`JMP Mem`
`MOV Mem,Reg` `MOV Mem2,Mem`	`MOV Mem2,Reg`
`OP Reg,xxx` `MOV Reg,yyy`	`MOV Reg,yyy`
`Jcc @xxx` `!Jcc @xxx`	`JMP @xxx` (ceci s'applique à (Jcc & 0FEh) avec (Jcc | 1)
`NOT Reg`	

NEG Reg	ADD Reg,1
NOT Reg ADD Reg,1	NEG Reg
NOT Mem NEG Mem	ADD Mem,1
NOT Mem ADD Mem,1	NEG Mem
NEG Reg NOT Reg	ADD Reg,-1
NEG Reg ADD Reg,-1	NOT Reg
NEG Mem NOT Mem	ADD Mem,-1
NEG Mem ADD Mem,-1	NOT Mem
NEG Mem != Jcc (CMP **without** Jcc)	NOP
TEST X,Y != Jcc	NOP
POP Mem JMP Mem	RET
PUSH Reg RET	JMP Reg
MOV Reg,Mem CALL Reg	CALL Mem
XOR Reg,Reg MOV Reg8,[Mem]	MOVZX Reg,byte ptr [Mem]
MOV Reg,[Mem] AND Reg,0FFh	MOVZX Reg,byte ptr [Mem]

Le tableau suivant présente des équivalences entre trois pseudo-instructions (colonne de gauche) et une pseudo-instruction (colonne de droite).

Suite de 3 pseudo-instructions	**pseudo-instruction équivalente**

MOV Mem,Reg
MOV Reg,Mem
POP Reg
	OP Reg,Reg2

MOV Mem,Reg
OP Mem,Reg2
MOV Reg,Mem
	OP Reg,Reg2

MOV Mem,Reg
OP Mem,Imm
MOV Reg,Mem
	OP Reg,Imm

MOV Mem,Imm
OP Mem,Reg
MOV Reg,Mem
	OP Reg,Imm (ce ne peut être une instruction SUB)

MOV Mem2,Mem
OP Mem2,Reg
MOV Mem,Mem2
	OP Mem,Reg

MOV Mem2,Mem
OP Mem2,Imm
MOV Mem,Mem2
	OP Mem,Imm

CMP Reg,Reg JO/JB/JNZ/JA/JS/JNP/JL/JG @xxx != Jcc	NOP
CMP Reg,Reg JNO/JAE/JZ/JBE/JNS/JP/JGE/JLE @xxx != Jcc	JMP @xxx
PUSH EAX PUSH ECX PUSH EDX	APICALL_BEGIN
POP EDX POP ECX POP EAX	APICALL_END

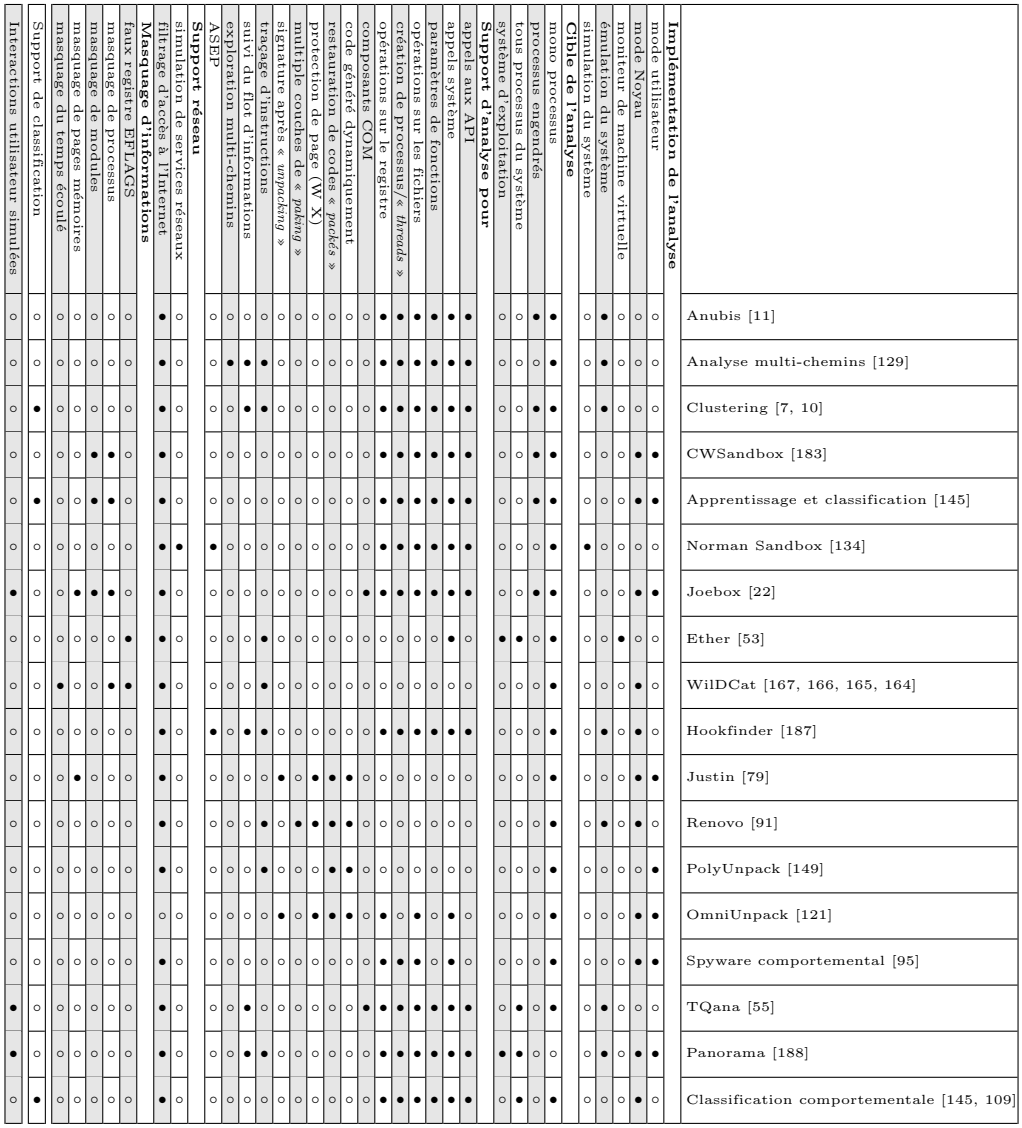
Le dernier tableau présente des équivalences entre trois pseudo-instructions (colonne de gauche) et deux pseudo-instructions (colonne de droite).

Suite de 3 pseudo-instructions	**Suite de 2 pseudo-instructions équivalentes**
MOV Mem,Imm CMP/TEST Reg,Mem Jcc @xxx	CMP/TEST Reg,Imm Jcc @xxx
MOV Mem,Reg SUB/CMP Mem,Reg2 Jcc @xxx	CMP Reg,Reg2 Jcc @xxx
MOV Mem,Reg AND/TEST Mem,Reg2 Jcc @xxx	TEST Reg,Reg2 Jcc @xxx
MOV Mem,Reg SUB/CMP Mem,Imm Jcc @xxx	CMP Reg,Imm Jcc @xxx
MOV Mem,Reg AND/TEST Mem,Imm Jcc @xxx	TEST Reg,Imm Jcc @xxx
MOV Mem2,Reg CMP/TEST Reg,Mem2 Jcc @xxx	CMP/TEST Reg,Mem Jcc @xxx
MOV Mem2,Mem AND/TEST Mem2,Reg Jcc @xxx	TEST Mem,Reg Jcc @xxx
MOV Mem2,Mem SUB/CMP Mem2,Reg Jcc @xxx	CMP Mem,Reg Jcc @xxx
MOV Mem2,Mem AND/TEST Mem2,Imm Jcc @xxx	TEST Mem,Imm Jcc @xxx
MOV Mem2,Mem SUB/CMP Mem2,Imm Jcc @xxx	CMP Mem,Imm Jcc @xxx

Annexe C

Revue des outils d'analyse dynamique de codes malveillants

Sur la page suivante, nous présentons un tableau, issu de [56], qui propose un comparatif entre les principaux outils d'analyse dynamique de codes malveillants.



	Anubis [11]	Analyse multi-chemins [129]	Clustering [7, 10]	CWSandbox [183]	Apprentissage et classification [145]	Norman Sandbox [134]	Joebox [22]	Ether [53]	WilDCat [167, 166, 165, 164]	Hookfinder [187]	Justin [79]	Renovo [91]	PolyUnpack [149]	OmniUnpack [121]	Spyware comportemental [95]	TQana [55]	Panorama [188]	Classification comportementale [145, 109]
Implémentation de l'analyse																		
mode utilisateur	○	○	○	●	●	○	●	○	○	○	●	○	●	●	●	○	●	○
mode Noyau	○	○	○	●	●	○	●	○	●	●	●	●	○	●	●	○	●	○
moniteur de machine virtuelle	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○
émulation du système	○	●	●	○	○	○	○	○	○	●	○	●	○	○	○	●	●	○
simulation du système	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
Cible de l'analyse																		
mono processus	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●
processus engendrés	○	○	●	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○
tous processus du système	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○
système d'exploitation	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○
Support d'analyse pour																		
appels aux API	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
appels système	●	●	●	●	●	●	●	●	○	●	○	○	○	○	○	●	●	●
paramètres de fonctions	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
opérations sur les fichiers	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
création de processus/« *threads* »	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
opérations sur le registre	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
composants COM	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○
code généré dynamiquement	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○
restauration de codes « *packés* »	○	○	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○
protection de page (W X)	○	○	○	○	○	○	○	○	○	○	●	●	○	●	○	○	○	○
multiple couches de « *puking* »	○	○	○	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○
signature après « *unpacking* »	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
traçage d'instructions	○	●	●	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○
suivi du flot d'informations	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
exploration multi-chemins	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	○
ASEP	○	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	○
Support réseau																		
simulation de services réseaux	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
filtrage d'accès à l'Internet	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
Masquage d'informations																		
faux registre EFLAGS	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○
masquage de processus	○	○	○	●	●	○	●	○	●	○	○	○	○	○	○	○	○	○
masquage de modules	○	○	○	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○
masquage de pages mémoires	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○
masquage du temps écoulé	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Support de classification	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●
Interactions utilisateur simulées	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	●	●	○

TABLE C.1 – Tableau extrait de [56] récapitulant les différentes techniques employées par les outils actuels d'analyse dynamique de codes malveillants.

Annexe D

Exemple de différentiation entre NCS et CMID : plagiat d'une fable de la Fontaine

Va-t-en, chétif Insecte, excrément de la terre.
C'est en ces mots que le Lion
Parlait un jour au Moucheron.
L'autre lui déclara la guerre.
Penses-tu, lui dit-il, que ton titre de Roi
Me fasse peur ni me soucie ?
Un Bœuf est plus puissant que toi,
Je le mène à ma fantaisie.
À peine il achevait ces mots
Que lui-même il sonna la charge,
Fut le Trompette et le Héros.
Dans l'abord il se met au large,
Puis prend son temps, fond sur le cou
Du Lion, qu'il rend presque fou.
Le Quadrupède écume, et son œil étincelle ;
Il rugit, on se cache, on tremble à l'environ ;
Et cette alarme universelle
Est l'ouvrage d'un Moucheron.
Un avorton de Mouche en cent lieux le harcelle,
Tantôt pique l'échine, et tantôt le museau,
Tantôt entre au fond du naseau.
La rage alors se trouve à son faîte montée.
L'invisible ennemi triomphe, et rit de voir
Qu'il n'est griffe ni dent en la bête irritée

Qui de la mettre en sang ne fasse son devoir.
Le malheureux Lion se déchire lui-même,
Fait résonner sa queue à l'entour de ses flancs,
Bat l'air qui n'en peut mais, et sa fureur extrême
Le fatigue, l'abat ; le voilà sur les dents.
L'Insecte du combat se retire avec gloire :
Comme il sonna la charge, il sonne la victoire,
Va partout l'annoncer, et rencontre en chemin
L'embuscade d'une Araignée :
Il y rencontre aussi sa fin.
Quelle chose par là nous peut être enseignée ?
J'en vois deux, dont l'une est qu'entre nos ennemis
Les plus à craindre sont souvent les plus petits ;
L'autre, qu'aux grands périls tel a pu se soustraire,
Qui périt pour la moindre affaire.

Bibliographie

[1] L. Adleman. An Abstract Theory of Computer Viruses. In *Advances in Cryptology—CRYPTO'88*, pages 354–374. Springer, 1990.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] M. Apel, C. Bockermann, and M. Meier. Measuring Similarity of Malware Behavior. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, pages 891–898. IEEE, 2009.

[4] G. Arboit. A Method for Watermarking Java Programs via Opaque Predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[5] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *Recent Advances in Intrusion Detection*, pages 165–184. Springer, 2006.

[6] D.H. Bailey. The computation of π to 29,360,000 decimal digits using Borweins' quartically convergent algorithm. *Mathematics of Computation*, 50(181):283–296, 1988.

[7] M. Bailey, J. Oberheide, J. Andersen, Z.M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, pages 178–197. Springer-Verlag, 2007.

[8] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In *Advances in Cryptology—Crypto 2001*, pages 1–18. Springer, 2001.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, page 177. ACM, 2003.

[10] U. Bayer, P.M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[11] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[12] P. Beaucamps. Advanced Polymorphic Techniques. *International Journal of Computer Science*, 2(3):194–205, 2007.

[13] P. Beaucamps and E. Filiol. On the possibility of practically obfuscating programs. *Journal in Computer Virology*, 3(1):3–21, 2007.

[14] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[15] C.H. Bennett, P. Gács, M. Li, MB Vitanyi, and W.H. Zurek. Information Distance. *IEEE Transactions on Information Theory*, 44(4):1407–1423, 1998.

[16] C. Bloom. Solving the Problems of Context Modeling, 1996.

[17] G. Bonfante, M. Kaczmarek, and J.Y. Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 5(3):263–270, 2009.

[18] J.M. Borello and L. Mé. Code Obfuscation Techniques for Metamorphic Viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[19] L. Bridges. The changing face of malware. *Network Security*, 2008(1):17–20, 2008.

[20] D. Bruschi, L. Martignoni, and M. Monga. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment*, 2006.

[21] D. Bruschi, L. Martignoni, and M. Monga. Code Normalization for Self-Mutating Malware. *IEEE Security & Privacy*, 5(2):46–54, 2007.

[22] S. Buhlmann. Extending joebox-a scriptable malware analysis system. Master's thesis, University of Applied Sciences Northwestern Switzerland, 2008.

[23] A.W. Burks. *Essays on Cellular Automata*. University of Illinois Press, 1970.

[24] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, System Research Center of Digital Equipement Corporation, 1994.

[25] R. Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *Advances in Cryptology-CRYPTO'97: 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 1997. Proceedings*, page 455. Springer, 1997.

[26] M. Cebrián, M. Alfonseca, and A. Ortega. Common pitfalls using the normalized compression distance: What to watch out for in a compressor. *Communications in Information and Systems*, 5(4):367–384, 2005.

[27] T.M. Chen and J.M. Robert. Worm Epidemics in High-Speed Networks. *Computer*, 37:48–53, 2004.

[28] X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker. Shared Information and Program Plagiarism Detection. *Information Theory, IEEE Transactions on*, 50(7):1545–1551, 2004.

[29] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions ond*, 2(3):113–124, 1956.

[30] N. Chomsky. On Certain Formal Properties of Grammars. *Information and control*, 2(2):137–167, 1959.

[31] M.R. Chouchane and A. Lakhotia. Using engine signature to detect metamorphic malware. In *Proceedings of the 4th ACM workshop on Recurring malcode*, page 78. ACM, 2006.

[32] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. *Information Security*, 2200:144–155, 2001.

[33] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*. USENIX Association, 2003.

[34] M. Christodorescu and S. Jha. Testing Malware Detectors. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 2004.

[35] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[36] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware Normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.

[37] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[38] C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[39] R. Cilibrasi, AL Cruz, S. de Rooij, and M. Keijzer. The CompLearn toolkit, 2003. Disponible à l'URL suivante : `http://www.complearn.org` (dernier accès en décembre 2010).

[40] R. Cilibrasi, P. Vitanyi, and R. Wolf. Algorithmic clustering of music based on string compression. *Computer Music Journal*, 28(4):49–67, 2004.

[41] R. Cilibrasi and P.M.B. Vitányi. Clustering by compression. *IEEE Transactions on Information theory*, 51(4):1523–1545, 2005.

[42] E. Clarke, O. Grumberger, and D. Long. *Model Checking*. MIT Press, 1999.

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

[44] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[45] C. Collberg, C. Thomborson, and D. Low. Breaking Abstractions and Unstructuring Data Structures. In *Proceedings of the IEEE International Conference on Computer Languages*. IEEE, 1998.

[46] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1998)*, pages 184–196. ACM, 1998.

[47] C.S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. *IEEE Transactions on software engineering*, 28:735–746, 2002.

[48] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2007. Disponible à l'URL suivante : `http://www.grappa.univ-lille3.fr/tata` (dernier accès en décembre 2010).

[49] T.M. Cover, J.A. Thomas, and J. Wiley. *Elements of of Information Theory*, volume 1. Wiley Online Library, 1991.

[50] J.R. Crandall, G. Wassermann, D.A.S. de Oliveira, Z. Su, S.F. Wu, and F.T. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. *ACM SIGARCH Computer Architecture News*, 34(5):25–36, 2006.

[51] A. Desnos, É. Filiol, and I. Lefou. Detecting (and creating!) a HVM rootkit (aka BluePill-like). *Journal in Computer Virology*, 7(1):1–27, 2009.

[52] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[53] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[54] M. Driller. Metamorphism in practice. *29A Magazine*, 1(6):–, 2002.

[55] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song. Dynamic Spyware Analysis. In *USENIX Annual Technical Conference*, pages 233–246, 2007.

[56] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Computing Surveys Journal*, X:X, 2011.

[57] T. Fawcett. ROC Graphs: Notes and Practical Considerations for Researchers. *Machine Learning*, 31:1–38, 2004.

[58] P. Ferrie. Un combate con el Kernado. *Virus bulletin*, -:8–9, 2002.

[59] P. Ferrie. Attacks on More Virtual Machine Emulators, 2007.

[60] P. Ferrie and T. Lee. W32.mydoom.a@mm, 2004. Disponible à l'URL suivante : `http://www.symantec.com/security_response/writeup.jsp?docid=2004-012612-5422-99` (dernier accès en décembre 2010).

[61] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO*, pages 186–194, 1986.

[62] E. Filiol. Strong cryptography armoured computer viruses forbidding code analysis: The Bradley virus. In *Proceedings of the 14th EICAR Conference*, pages 201 – 217, 2005.

[63] E. Filiol. Whale: le virus se rebiffe. *Journal de la sécurité informatique MISC*, 19:–, 2005.

[64] É Filiol. Metamorphism Formal Grammars and Undecidable Code Mutation. *International Journal of Computer Science*, 2(1):70–75, 2007.

[65] É Filiol. *Techniques virales avancées*. Springer, 2007.

[66] E. Filiol, G. Jacob, and L.M. Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3(1):23–37, 2007.

[67] P. Foggia. The VFLIB graph matching library, version 2.0, 2001.

[68] M. Fredrikson, M. Christodorescu S. Jha, R. Sailer, and X. Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *IEEE Symposium on Security and Privacy*, pages 45–60, 2010.

[69] T. Garnett. Dynamic Optimization of IA-32 Applications Under DynamoRIO.

[70] A. Gazet. Comparative analysis of various ransomware virii. *Journal in computer virology*, 6(1):77–90, 2010.

[71] J. Göbel. Amun: A Python Honeypot. Technical report, University of Mannheim, 2009.

[72] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik*, 38(1):173–198, 1931.

[73] J. Goguen and G. Malcolm. *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer Academic Publishers, 2000.

[74] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–35, 1974.

[75] S. Goldwasser and Y.T. Kalai. On the impossibility of obfuscation with auxiliary input. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 553–562. IEEE, 2005.

[76] S. Goldwasser and G.N. Rothblum. On Best-Possible Obfuscation. *Theory of Cryptography*, 4392:194–213, 2007.

[77] J.B. Grizzard, V. Sharma, C. Nunnery, B.B.H. Kang, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*. USENIX Association, 2007.

[78] G. Gueguen. Van Wijngaarden grammars, metamorphism and K-ary malwares, 2010. disponible à l'adresse : `http://arxiv.org/abs/1009.4012` (dernier accès en décembre 2010).

[79] F. Guo, P. Ferrie, and T. Chiueh. A Study of the Packer Problem and Its Solutions. In *Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008.

[80] R.K. Guy. *Unsolved problems in number theory*. Springer Verlag, 2004.

[81] M. H. Halstead. *Elements of Software Science*. Elsevier North Holland, 1977.

[82] D. Harley, U.E. Gattiker, and R. Slade. *Virus: définitions, mécanismes et antidotes (Systèmes et réseaux Coll. Référence)*. Campus Press, 2002.

[83] M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc. New York, NY, USA, 1977.

[84] S. Horwitz. Precise Flow-insensitive May-alias Analysis is NP-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6, 1997.

[85] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[86] ITSEC. Evaluation Criteria of the Information System Security. Technical report, Office des publications officielles des Communautés Européennes, 1991.

[87] G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology*, 4(3):251–266, 2008.

[88] G. Jacob, H. Debar, and E. Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *Recent Advances in Intrusion Detection*, pages 81–100. Springer, 2009.

[89] G. Jacob, E. Filiol, and H. Debar. Functional polymorphic engines: formalisation, implementation and use cases. *Journal in Computer Virology*, 5(3):247–261, 2009.

[90] N.D. Jones. *Computability and complexity: from a programming perspective*. The MIT Press, 1997.

[91] M.G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53. ACM, 2007.

[92] Richard M. Karp. Reducibility Among Combinatorial Problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, 2010.

[93] E. Kaspersky. DarkParanoid-Who Me? *Virus bulletin*, -:8–9, 1998.

[94] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting Malicious Code by Model Checking. *Intrusion and Malware Detection and Vulnerability Assessment*, 3548:174–187, 2005.

[95] E. Kirda, C. Kruegel, G. Banks, G. anf Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[96] S.C. Kleene. On Notation for Ordinal Numbers. *Journal of Symbolic Logic*, 3(4):150–155, 1938.

[97] D.E. Knuth. Semantics of context-free grammars. *Theory of Computing Systems*, 2:127–145, 1968.

[98] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International joint Conference on artificial intelligence*, volume 14, pages 1137–1145. Citeseer, 1995.

[99] C. Kolbitsch, P.M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, X.F. Wang, and UC Santa Barbara. Effective and Efficient Malware Detection at the End Host. In *18th Usenix Security Symposium*, 2009.

[100] A.N. Kolmogorov. Three approaches to the definition of the concept "quantity of information". *Problemy Peredachi Informatsii*, 1(1):3–11, 1965.

[101] KA Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1):77–108, 1996.

[102] J. Kraus. Selbstreproduktion bei Programmen. Master's thesis, Universit "at Dortmund Fachschaft Informatik, 1980.

[103] J. Kraus. On self-reproducing computer programs. *Journal in Computer Virology*, 5:9–87, 2009.

[104] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.

[105] H.W. Kuhn. The Hungrian Method for the Assignment Problem. *Naval Research Logistics Quaterly*, 2:83–97, 1955.

[106] A. Lakhotia, A. Kapoor, and EU Kumar. Are metamorphic viruses really invincible ? In *Virus Bulletin*, pages 5–7, 2004.

[107] W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):337, 1992.

[108] K.P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es):7, 1996.

[109] T. Lee and J. Mody. Behavioral classification. In *European Institute for Computer Antivirus Research Conference (EICAR)*, 2006.

[110] M. Lesk. The New Front Line: Estonia under Cyberassault. *Security & Privacy, IEEE*, 5(4):76–79, 2007.

[111] M. Li, J.H. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17(2):149, 2001.

[112] M. Li, X. Chen, X. Li, B. Ma, and P.M.B. Vitányi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 2004.

[113] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag New York Inc, 2008.

[114] D. Lin. Hunting for Undetectable Metamorphic Viruses. *Journal in Computer Virology*, X:X, 2011.

[115] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[116] E. Littré. *Dictionnaire de la langue française: Supplément*. Hachette, 1886.

[117] D. Low. Java Control Flow Obfuscation. Master's thesis, Master of Science Thesis, Department of Computer Science, The University of Auckland, 1998.

[118] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM, 2005.

[119] B. Lynn, M. Prabhakaran, and A. Sahai. Positive Results and Techniques for Obfuscationarboit. In *Advances in Cryptology-EUROCRYPT 2004*, pages 20–39. Springer, 2004.

[120] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials Held in conjunction with Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2002.

[121] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441. IEEE, 2007.

[122] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 2(4):308–320, 1976.

[123] Microsoft. Microsoft Security Intelligence Report volume 5, January through June 2008.

[124] Microsoft. Microsoft Security Intelligence Report volume 7, January through June 2009.

[125] Microsoft. Microsoft Security Intelligence Report volume 6, July through December 2008.

[126] Microsoft. Microsoft Security Intelligence Report volume 8, July through December 2009.

[127] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security & Privacy*, 1(4):33–39, 2003.

[128] T. Moore, R. Clayton, and R. Anderson. The Economics of Online Crime. *The Journal of Economic Perspectives*, 23(3):3–20, 2009.

[129] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy, 2007. SP'07*, pages 231–245, 2007.

[130] A. Moshchuk, T. Bragin, S.D. Gribble, and H.M. Levy. A Crawler-based Study of Spyware on the Web. In *Proceedings of the 2006 Network and Distributed System Security Symposium*, pages 17–33, 2006.

[131] M. Munson Taghi and C. John. Measurement of data structure complexity. *Journal of Systems and Software*, 20(3):217–225, 1993.

[132] N. Nethercote and J. Seward. Valgrind:: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003.

[133] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[134] Norman. Norman SandBox Whitepaper, 2003. Disponible à l'URL suivante : `http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf` (dernier accès en décembre 2010).

[135] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software Obfuscation on a Theoretical Basis and Its Implementation. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 86(1):176–186, 2003.

[136] E. I. Oviedo. Control flow, data flow, and program complexity. In *IEEE COMPSAC*, pages 146–152, 1980.

[137] C.H. Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[138] E.L. Post. Recursive Unsolvability of a Problem of Thue. *Journal of Symbolic Logic*, 12(1):1–11, 1947.

[139] M. Preda, R Giacobazzi, S. Debray, K. Coogan, and G.M. Townsend. Modelling metamorphism by abstract interpretation. In *Proceedings of the 17th international conference on Static analysis*, pages 218–235, 2010.

[140] M.D. Preda, M. Christodorescu, S. Jha, and S. Debray. A Semantics-Based Approach to Malware Detection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):25, 2008.

[141] Qozah. Polymorphism and grammars. In *29A E-zine*, volume 4, 1999. Disponible à l'URL suivante : `http://www.29a.net/29a-4/29a-4.207` (dernier accès en décembre 2010).

[142] L.R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[143] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[144] H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[145] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 5137:108–125, 2008.

[146] J. Riordan and B. Schneier. Environmental key generation towards clueless agents. *Mobile Agents and Security*, X:15–24, 1998.

[147] R.L. Rivest, L. Adleman, and L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of secure computation (Workshop Georgia Institute of Technologie)*, pages 169–179, 1978.

[148] H. Rogers Jr. *Theory of recursive functions and effective computability*. MIT Press Cambridge, MA, USA, 1987.

[149] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 289–300. IEEE, 2006.

[150] M. Russinovich and B. Cogswell. Windows Sysinternals, 2008. Disponible à l'URL suivante : `http://download.sysinternals.com/Files/SysinternalsSuite.zip` (dernier accès en décembre 2010).

[151] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, pages 45–54, 2002.

[152] M. Shafiq, S. Tabish, F. Mirza, and M. Farooq. PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2009.

[153] C.E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):55, 2001.

[154] P. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *4th IEEE Information Assurance Workshop*, 2003.

[155] M. Sintzoff. Existence of a Van Wijngaarden syntax for every recursively enumerable set. *Annales de la Société Scientifique de Bruxelles*, 81(2):115–118, 1967.

[156] E. H. Spafford. The Internet worm incident. In *2nd European Software Engineering Conference*, pages 446–468, 1989.

[157] D. Spinellis. Reliable Identification of Bounded-length Viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, 2003.

[158] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[159] P. Szor and P. Ferrie. Hunting for metamorphic. In *Virus Bulletin*, volume 123, 2001.

[160] J. Toger. *Specification-Driven Dynamic Binary Translation*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2004.

[161] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230, 1937.

[162] A. van Wijngaarden. The generative power of two-level grammars. In *Proceedings of the 2nd Colloquim on Automata Languages and Programming*, pages 9–16, 1974.

[163] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, CH. Lindsey, L. Meertens, and RG. Fisker. *Revised report on the algorithmic language ALGOL 68*. Springer-Verlag Berlin, 1976.

[164] A. Vasudevan and R. Yerraballi. Sakthi: A retargetable dynamic framework for binary instrumentation. In *Hawaii International Conference in Computer Sciences*, 2004.

[165] A. Vasudevan and R. Yerraballi. Stealth Breakpoints. In *Computer Security Applications Conference, 21st Annual*, page 10, 2005.

[166] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In *IEEE Symposium on Security and Privacy*, 2006.

[167] A. Vasudevan and R. Yerraballi. Spike: engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the 29th Australian Computer Science Conference*, pages 311–320, 2006.

[168] GS Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications, J. American Inst. *Transactions of the American Institute of Electrical Engineers*, 55:109–115, 1926.

[169] J. Von Neumann. The general and logical theory of automata. In *Cerebral Mechanisms in Behavior: The Hixon Symposium*, pages 1–41, 1951.

[170] J. Von Neumann, A.W. Burks, et al. *Theory of self-reproducing automata*. Univ. of Illinois Press Urbana, IL, 1966.

[171] VX Heavens repository. Disponible a l'URL suivante : `http://vx.netlux.org/` (dernier accès en décembre 2010).

[172] S. Wagon. Is π normal ? *The Math Intelligencer*, 7:65–67, 1985.

[173] A. Walenstein, R. Mathur, M. Chouchane, and A. Lakhotia. The Design Space of Metamorphic Malware. In *ICIW 2007 2nd International Conference on i-Warfare and Security*, page 241, 2007.

[174] A. Walenstein, R. Mathur, M.R. Chouchane, and A. Lakhotia. Normalizing Metamorphic Malware Using Term Rewriting. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 75–84. IEEE Computer Society, 2006.

[175] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of Software-based Survivability Mechanisms. In *IEEE Computer Society*, 2003.

[176] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, Tech. Rep. CS-2000-12, 2000.

[177] K. Wang. Using HoneyClients to Detect New Attacks. In *RECON conference*, 2005.

[178] J. Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

[179] M. Webster and G. Malcolm. Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, 2(3):149–161, 2006.

[180] M. Webster and G. Malcolm. Detection of Metamorphic and Virtualization-based Malware using Algebraic Specification. *Journal in Computer Virology*, 5(3):221–245, 2009.

[181] S. Wehner. Analyzing Worms and Network Traffic using Compression. *Journal of Computer Security*, 15(3):303–320, 2007.

[182] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[183] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. In *IEEE Symposium on Security and Privacy, 2007. SP'07*, 2007.

[184] W. Wong and M. Stamp. Hunting for Metamorphic Engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[185] G. Wroblewski. General Method of Program Code Obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, pages 153–159, 2002.

[186] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[187] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*. Citeseer, 2008.

[188] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, page 127. ACM, 2007.

[189] P.V. Zbitskiy. Code mutation techniques by means of formal grammars and automatons. *Journal in Computer Virology*, 5(3):199–207, 2009.

[190] Q. Zhang and D.S. Reeves. MetaAware: Identifying metamorphic malware. In *23rd Annual Computer Security Applications Conference (ACSAC)*, pages 411–420. IEEE, 2007.

[191] WX Zhang and Y. Leung. Theory of including degrees and its applications to uncertainty inferences. In *Fuzzy Systems Symposium, 1996.'Soft Computing in Intelligent Systems and Information Processing'., Proceedings of the 1996 Asian*, pages 496–501. IEEE, 2002.

[192] W. Zhu, C. Thomborson, and F.Y. Wang. Obfuscate arrays by homomorphic functions. In *2006 IEEE International Conference on Granular Computing*, pages 770–773, 2006.

[193] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[194] C.C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 138–147. ACM, 2002.

[195] Z. Zuo and M. Zhou. Some Further Theoretical Results about Computer Viruses. *The Computer Journal*, 47(6):627–633, 2004.

[196] Z. Zuo, Q. Zhu, and M. Zhou. On the Time Complexity of Computer Viruses. *IEEE Transactions on information theory*, 51(8):2962–2966, 2005.