N° d'ordre : 4351 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
Ingénieur de recherche à l'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) / examinateur

Frédéric Valette
Ingénieur au centre DGA Maîtrise de l'Information / 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

Je tiens avant tout à exprimer mes remerciements aux membres du jury qui ont accepté d'évaluer mon travail de thèse : Thomas Jensen pour avoir bien voulu présider le jury de cette thèse. Guillaume Bonfante et Jean-Marc Steyaert pour avoir accepté d'être les rapporteurs de ce manuscrit ainsi que pour leurs commentaires. Merci également à Marc Dacier, à Loïc Duflot ainsi qu'à Frédéric Valette pour avoir accepté d'examiner mon mémoire et de faire parti de mon jury de thèse.

Un merci tout particulier à mes deux encadrants ,Ludovic Mé et Éric Filiol pour avoir non seulement accepté de diriger cette thèse mais surtout pour leurs conseils avisés qui m'ont servis de phare tout au long de ces années de recherche. C'est grâce à vous si j'ai tant progressé au niveau de la rigueur de mon travail académique ainsi que sur le plan pédagogique.

Bien entendu, ce travail n'aurait jamais pu voir le jour sans l'appui de la hiérarchie du centre DGA-MI qui m'a autorisé à mener à bien ces travaux de recherche en même temps que mon activité professionnelle, ce en quoi je les remercie.

Je tiens également à remercier toutes les personnes qui m'ont permis, à un moment ou à un autre, d'avancer sur le plan scientifique : Didier Eymery pour avoir fait de moi un *Hacker*. Éric Bornette pour m'avoir soutenue dès l'origine de mes travaux et pour sa relecture attentive de mon mémoire. Arleene McFane pour son aide dans la rédaction des articles en anglais. Pascal Navarre pour m'avoir considérablement aidé dans la mise en place de plateformes d'expérimentation. Philippe Bion pour nos discussions sur les analyses de codes malveillants autour d'un café. Denis et Yvon pour leur amitié et les soirées en refuges passées à m'écouter parler virus autour d'une bière. Aux personnes de l'ex-département *Analyse de la Menace Informatique* (AMI) pour m'avoir supporté pendant toutes ces années ainsi qu'aux membres de l'équipe SSIR de Supélec.

Finalement, un grand merci à mes parents qui m'ont permis d'être ce que je suis aujourd'hui ainsi que pour l'organisation du pot d'après thèse. Ma dernière pensée s'adresse naturellement à mon épouse Fabienne, où quelques mots ne suffiraient pas à décrire son soutient et son aide au quotidien...

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'information 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 déni de service distribué (« *Distributed Denial of Service* » ou DDoS) jamais décrite en Europe a eu lieu le 27 avril 2007 [79]. 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) [63], 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 modifiées par des utilisateurs non autorisés ou par suite d'erreurs ;
- la disponibilité, les informations doivent être accessibles à tout utilisateur autorisé, en toutes 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 informatiques. 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)) [89, 90, 91, 92]. Ce panorama récent 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 [123, 124] et de Burks [15], 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 [69], 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 [30], 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 [72] en 1980 et ceux de Cohen [30] en 1986. Parmi les premiers virus découverts, les plus célèbres sont Elk Cloner tournant sous AppleDOS 3.3, apparue en 1983, ainsi que le virus d'amorce de disque Brain, découvert en 1986 [60].

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 son

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 [83] à savoir, une inclinaison à malfaire. Pour lever toute ambiguïté, nous utiliserons uniquement comme traduction le terme « malveillant ».

origine semble plus accidentelle que volontaire, ce programme constitue le premier ver connu [114]. 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 [116] 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 [17] et SLAMMER en 2003 [93]. 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 [143]. 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* ») [96], des réseaux de zombies (« *botnets* ») [57], des logiciels visant à extorquer de l'argent (« *ransomware* ») [11, 51].

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 [94] .

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 [91]. Six mois plus tard, ce nombre s'élevait alors à 116 millions [90] pour finalement atteindre 126 millions à la fin de l'année 2009 [92]. 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 [118]. 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

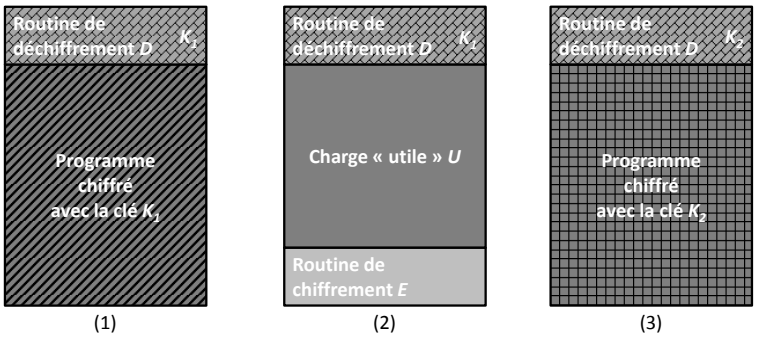

FIGURE 1 – Fonctionnement d'un code auto-reproducteur chiffré.

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 s'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 » [119].

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 *pertinent*e. 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 **??** et **??**, qui présentent la conception, l'implémentation et l'utilisation de notre moteur de métamorphisme.

Le chapitre **??** 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 **??** 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 **??** et **??**, qui exposent notre approche de détection dynamique de codes malveillants fondée sur la complexité de Kolmogorov.

Le chapitre **??** 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 similarité 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 [28]. 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 [141].

Le chapitre **??** 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 suite à 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 [72, 73] 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 [30]. Dans ces travaux, Cohen s'attachent à déterminer la complexité de la détection virale en s'appuyant sur le formalisme des machines de Turing [103]. 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 fini déterministe (« *Deterministic Finite Automaton* » 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 [30], 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 [30]). Un ensemble viral $\mathcal{V}_c$ est défini de la façon suivante :*

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

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 [30], les deux principaux problèmes abordés sont celui de 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 [30]). 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 [30]). 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 malveillants

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.* [144, 145], 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 [69] correspondent à un autre formalisme permettant de décrire la notion d'algorithme. Bien que cette approche semble plus abstraite que celle par machine de Turing, ces deux formalismes sont toutefois équivalents. En effet, la classe des fonctions partielles récursives correspond exactement aux fonctions calculées par les machines de Turing [112]. 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] [52]. 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 sein 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 :

1. La numérotation de Gödel s'appuie sur la factorisation en nombres premiers. Un entier est associé 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.

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 :*

- ***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.

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.*

La détection d'un virus au sens d'Adleman apparait de fait hors de portée pratique. Intéressons-nous maintenant au cas des programmes évolutifs.

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

Ce théorème exprime la difficulté de détermination de l'ensemble des programmes infectés par v.

1.1.2.3 Le modèle viral de Zuo *et al.*

Zuo *et al.* [144, 145] ont repris et complété le formalisme d'Aldeman afin de 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 [144]). 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 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 [144]). 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 [144]). 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 [144]). 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 [144]). 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 particulière sur la taille des programmes. Dans le cas où les programmes sont de tailles bornées, Spinellis a démontré qu'il est possible de construire des codes viraux polymorphes pour lesquels la détection est prouvée NP-complète [117]. Ce résultat est obtenu par réduction du problème SAT [67] 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 [145]). 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 **??**.

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 [108]. 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 [103]). 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 des langages, celui de la décision d'un langage.

Définition 10. *(Décision d'un langage [66]). 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.

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 ta-

$G = (\{A, B\}, \{a, b, c, x, y\}, S, R)$ 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 [108].

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

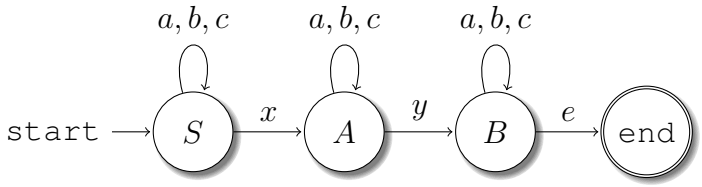

FIGURE 1.3 – Automate permettant de détecter la routine polymorphe générée par la grammaire G de la figure 1.2

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 [18, 19] 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*** (ou encore *sans-contexte*), 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 :

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

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 [19]). Le problème de décision d'un langage est :*
- *indécidable pour les grammaires de type 0 [80] ;*
- *PSPACE-complet pour les grammaires de type 1 [49] ;*
- *polynomial pour les grammaires de type 2 [40] ($\mathcal{O}(|G|^2 n^3)$ avec $|G|$ la taille de la grammaire G et n la longueur de la chaîne d'entrée) ;*
- *linéaire pour les grammaires de type 3.*

1.1.3.2 Le métamorphisme selon Filiol

S'appuyant sur les résultats de Chomsky, Filiol [44] propose de définir le métamorphisme au moyen de grammaires formelles. Si dans la définition de 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 [44]). 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 [105], au même titre que le problème de l'arrêt de Turing [121]. 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 [118, chapitre 7]. Cet aspect est conforté par l'exemple du code malveillant Win32.MetaPHOR [2] connu pour être le virus métamorphe le plus abouti [44, 118] dont les règles de réécriture sont fournies en annexe **??**. Ces mutations de code correspondent à des transformations syntaxiques généralement simples comme nous le verrons en section 1.2. Les travaux de Zbitskiy [139] 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. La figure 1.4 illustre le lien hiérarchique existant entre les diverses classes de complexité employées.

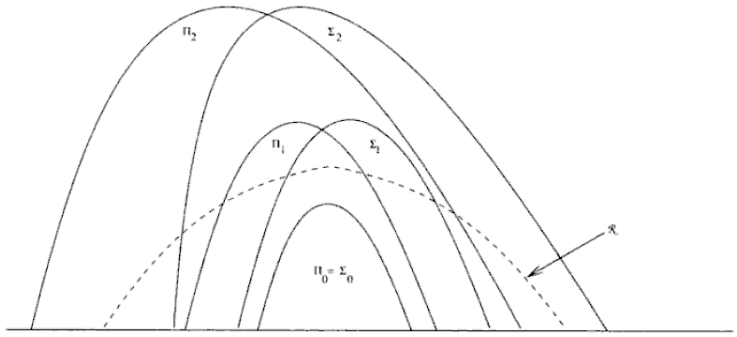

FIGURE 1.4 – Illustration (extraite de [46]) de la hiérarchie des classes de complexité. Les pointillés représentent la limite des problèmes que l'on sait résoudre dans la pratique.

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

2. Ce virus est présenté en section 1.2.7.2

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 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 [118], 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 [20, 39, 76, 106, 119, 126, 127, 131, 132, 135, 140]. 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.* [145] ainsi que Filiol [44]. Dans ce cas, ce sont les règles de mutation elles-mêmes qui changent 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 [118] 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 [139] ainsi que les règles de réécritures employées par le virus METAPHOR (voir annexe **??**). 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 [3], 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

3. 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 ».

ceux compatibles avec le « *framework* » .NET [4] 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. C'est dans le but de protéger ces informations contenues dans les fichiers intermédiaires que s'est développée la technique de 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 [16, 8, 86, 4, 55]. 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 [31, 32, 33]. Finalement, des approches sont proposées pour le tatouage (« *watermarking* ») [34] ainsi que la protection des logiciels [82, 84, 101, 136]. 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 [31, 33]). 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 $V_{i+1} = \mathcal{T}(V_i)\}$ [5]. 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

4. 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.

5. 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.

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) [103]. 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 ;
- 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 [111] ainsi que celui de la transformation d'un algorithme de chiffrement symétrique en chiffrement asymétrique [37]. Pour plus de détails sur les applications possibles de l'obscurcissement de code voir les travaux de Barak *et al.* [4]. 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.* [4].

Définition 14. *(TM obscurcisseur [4]). 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 étapes de calcul 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 $M \in \mathcal{M}$,*

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

La notation $\approx$ signifie que les distributions de probabilité sont équivalentes à une fonction négligeable près en fonction de la taille du programme M considéré.

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 [4] 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 » [4]). 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 [56] 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 [43]. 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 » [56]). 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.* [86] 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. [86].) Pour un oracle aléatoire $\mathcal{R} : \{0,1\}^* \to \{0,1\}^{2k}$, 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 [8]). 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 [4] est qu'elle conduit au résultat positif suivant :

Théorème 10. *(Existence de τ-obscurcisseur de code [8]). 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.* [145] 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 [31, 33] 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 [59], la complexité cyclomatique [88], la complexité du flot de données [102] ou encore celle de la structure des données [97]. La définition de la puissance d'une transformation proposée par Collberg *et al.* est la suivante :

Définition 16. *(puissance [31]). 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 [31]). 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 1.5.

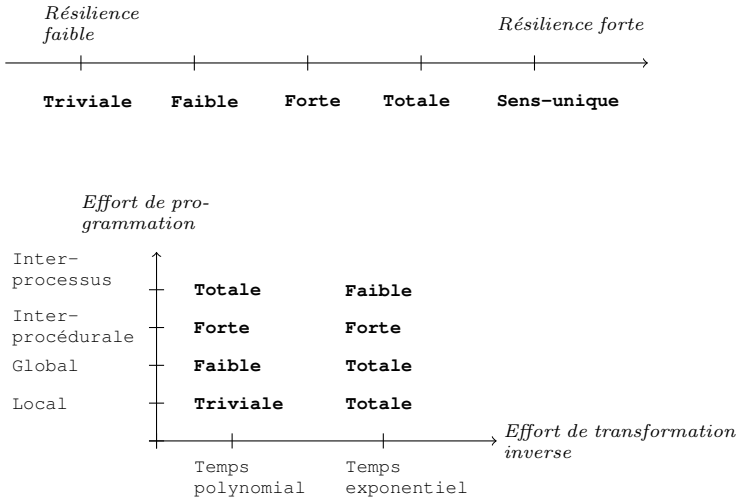

FIGURE 1.5 – Matrice d'effort R, extraite de [31], permettant l'évaluation de la résilience. L'échelle de résilience est représentée en haut de la figure.

La dernière mesure communément utilisée dans l'évaluation de l'obscurcissement de code est le coût, qui représente l'augmentation de ressources nécessaires à l'exécution du programme obscurci $\mathcal{O}(\mathcal{P})$ par rapport au programme d'origine P.

Définition 18. *(coût [31]). 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.* [31, 33] 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 [33]). 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 [31, 33, 82, 101, 128, 129]. Divers techniques ont

été proposées pour construire ce type de prédicats [33]. 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 1.6 présente des exemples de prédicats opaques extraits de [3]. 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

$$\forall (x, y) \in \mathbb{Z}^2, 7y^2 - 1 \neq x^2 \tag{1.7}$$

$$\forall x \in \mathbb{Z}, 3 | (x^3 - x) \tag{1.8}$$

$$\forall x \in \mathbb{Z}, 2 | x \vee 8 | (x^2 - 1) \tag{1.9}$$

$$\forall x \in \mathbb{Z}, \sum_{i=1, i \neq 0[2]}^{2x-1} i = x^2 \tag{1.10}$$

$$\forall x \in \mathbb{N}, 2 \left| \left\lfloor \frac{x^2}{2} \right\rfloor \right. \tag{1.11}$$

FIGURE 1.6 – Exemples de prédicats opaques arithmétiques toujours vrais.

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 paire.

Les travaux d'Arboit [3] 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 $\left(\frac{(4a)^{x+1}}{2}\right)^2 \equiv a[p]$.

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

L'analyse statique est définie par Landi [77] 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 [62] et même indécidables dans le cas des langages autorisant des branchements conditionnels, des boucles, des allocations dynamiques de mémoire ainsi que des structures récursives [77, 110]. 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 [58], 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 [31].

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.* [31, 33].

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 [118, 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.

```
1  .method private static hidebysig specialname void .cctor()
2  {
3    ldstr "[.NET.Snail - sample CLR virus (c) whale 2004 ]"
4    stsfld class System.String Ylojnc.lgxmAxA::WaclNvK
5    nop
6    ldc.i4.6
7    ldc.i4.s 0xF
8    call int32 [mscorlib]System.Environment::get_TickCount()
9    nop
10   newobj void nljvKpqb::.ctor(int32 len1, int32 len2, int32 seed)
11   stsfld class nljvKpqb Ylojnc.lgxmAxA::XxnArefPizsour
12   call int32 [mscorlib]System.Environment::get_TickCount()
13   nop
14   newobj void [mscorlib]System.Random::.ctor(int32)
15   stsfld class [mscorlib]System.Random Ylojnc.lgxmAxA::aajqebjtoBxjf
16   ret
17 }
```

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 découper 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 [142].

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.* [31, 33] reprend l'ensemble des transformations dé-

crites. 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.* [128, 129] 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.* [101]. 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.* [21]. 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 [82] 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 [6]. 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 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 [136, 137] 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.* [31, 33] et celle de Wang *et al.* [128, 129] 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 [7]. De même, le site VX Heavens [125] offre une collection de quelques 271 092 programmes à caractère malveillant dont 95% sont des binaires et seulement 5% des codes en langages interprétés.

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

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

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 [118, 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.

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 esi,dword ptr [ebp+0Ch]`
`mov edi,dword ptr [ebp+08h]`
`repnz movsb`
	`mov edi,dword ptr [ebp+08h]`
	`mov ecx,104h`
	`mov esi,dword ptr [ebp+0Ch]`
	`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 [39]. 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 [118, 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.* [76] 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 1.7 :

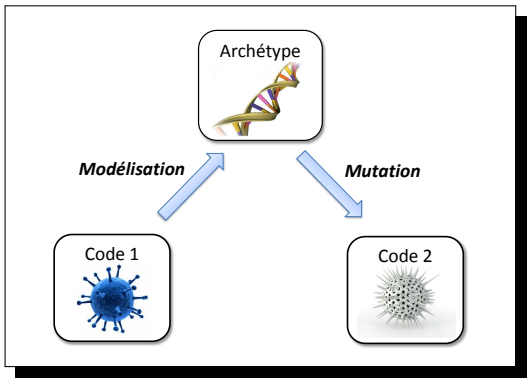

FIGURE 1.7 – Schéma simplifié du processus d'auto-reproduction d'un code métamorphe.

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 représentation bien que d'autres appellations telles que *forme normale* ou encore *forme canonique* soient 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 1.7. 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.* [126] 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 [8] 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 : 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 [9] qui constitue à ce jour le code

8. Les fichiers sources sont téléchargeables à l'URL suivante : `http://vx.netlux.org/dl/mag/29a-6.zip` (dernier accès en décembre 2010).
9. Le source assembleur est téléchargeable à l'URL suivante : `http://vx.netlux.org/src_view.php?file=metaphor1d.zip` (dernier accès en décembre 2010).

métamorphe le plus abouti [45]. Une description complète est exposée dans [7] et [45] 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% correspondent 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) [10].

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. La figure 1.8 présente la réplication du virus pour une portion de l'amorce virale.

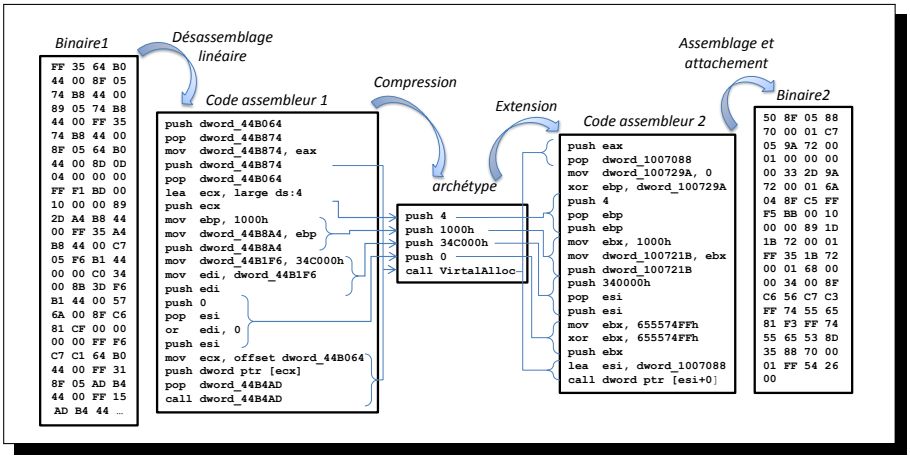

FIGURE 1.8 – Illustration du fonctionnement de l'amorce du virus METAPHOR.

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

10. Certains travaux [139] 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 1.8 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 1.8. 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 1.8 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 1.8 ;
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 1.8.

Comme en témoigne la figure 1.8, les formes mutées de ce virus métamorphe 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 entrée/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 (voir 1.2.2.2), permettant notamment l'authentification par mot de passe, démontrent 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 [13, 76] 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 **??** 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 [41], appelée aussi tableau de contingence, dont une illustration est donnée en figure 1.9.

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.

		Détecteur Idéal	
		Positif	*Négatif*
Détecteur Réel	*Positif*	Vrai Positif (VP)	Faux Positif (FP)
	Négatif	Faux Négatif (FN)	Vrai Négatif (VN)
		P	*N*

FIGURE 1.9 – 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 [41].

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 1.10 :

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 1.10 et détaillé en section 1.3.2.1. Il comporte deux étapes qui reprennent celles de la compilation mais en sens inverse :

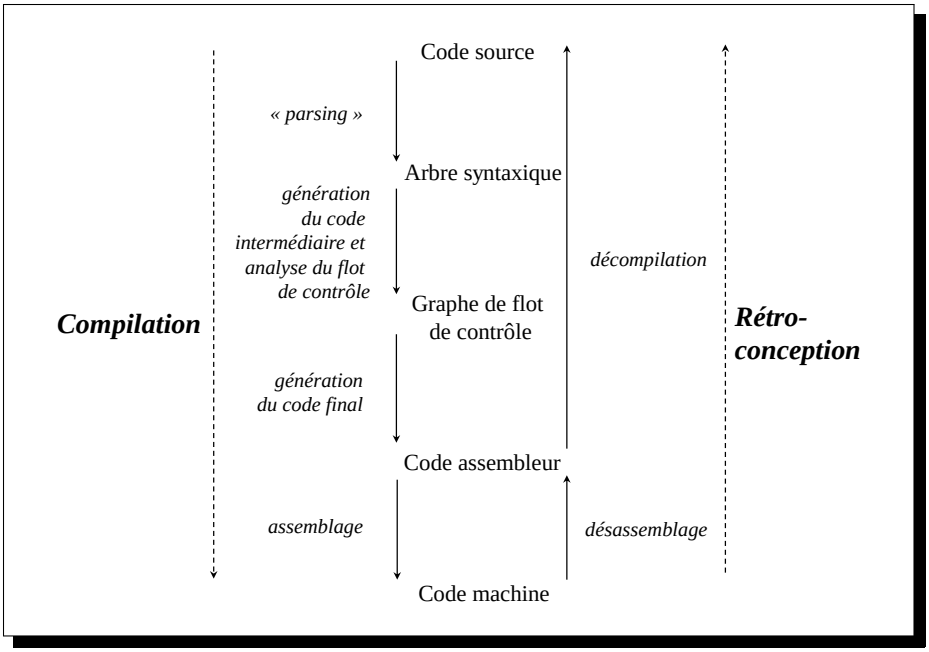

FIGURE 1.10 – Schéma du lien existant entre une chaîne de compilation et le processus de rétro-conception.

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 [26, 27], est illustrée à droite de la figure 1.10. 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 [23].

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 1.9, 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 [121] se réduit directement à celui du désassemblage [11].

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

11. 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.

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 [82].

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 [113] 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 1.11 montre une implémentation itérative correcte, en langage C, de la fonction factorielle pour une architecture 32 bits, ainsi que son CFG associé.

```
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 }
```

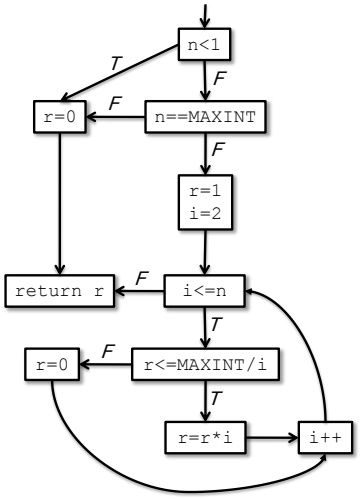

FIGURE 1.11 – Exemple de programme avec son CFG associé.

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 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, 61]. 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.* [26, 27]. 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 jamais 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) [109] 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.* [135] 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 diverses 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 [81]. 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 [29]. 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.* [115] décrivent un système de détection vérifiant des propriétés du CFG d'un programme suspect par rapport à une formule de logique temporelle linaire (« *Linear Temporal 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.* [68] en introduisant une nouvelle logique temporelle dénommée « *Computation Tree Predicate Logic* » (CTPL). Cette logique est aussi expressive que « *Computation Tree Logic* » (CTL) mais permet 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 [68], est donné en figure 1.12 avec sa formule CTPL correspondante. Brièvement, cette formule CTPL décrit le com-

```
mov  edi,[ebp+arg0]
xor  ebx,ebx
push edi
...
lea  eax,[ebp+ExFileName]
mov  [esp+65Ch+var65C],104
push eax
push ebx
call ds:GetModuleFileNameA
lea  eax,[ebp+NewFileName]
push ebx
push eax
lea  eax,[ebp+ExFileName]
push eax
call ds:CopyFileA
```

$$
\begin{aligned}
&\exists L_m \exists L_c \exists v_{File}(\\
&\ \exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0 \\
&\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\ \mathbf{E}(\neg \exists t(\mathtt{mov}(r_0,t) \vee \mathtt{lea}(r_0,t))\ \mathbf{U} \#\mathtt{loc}(\\
&\qquad L_0)) \wedge \\
&\quad \mathbf{EF}(\mathtt{mov}(r_1,0) \wedge \mathbf{EX}\ \mathbf{E}(\neg \exists t(\mathtt{mov}(r_1,t) \vee \mathtt{lea}(r_1,t))) \mathbf{U} \#\mathtt{loc}(L_1)) \\
&\qquad \wedge \\
&\quad \mathbf{EF}(\mathtt{push}(c_0) \wedge \mathbf{EX}\ \mathbf{E}(\neg \exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
&\qquad \mathbf{U}(\mathtt{push}(r_0) \wedge \#\mathtt{loc}(L_0) \wedge \mathbf{EX}\ \mathbf{E}(\neg \exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
&\qquad\quad \mathbf{U}(\mathtt{push}(r_1) \wedge \#\mathtt{loc}(L_1) \wedge \mathbf{EX}\ \mathbf{E}(\neg \exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
&\qquad\qquad \mathbf{U}(\mathtt{call}(\mathtt{GetModuleFileNameA}) \wedge \#\mathtt{loc}(L_m)))) \\
&\qquad\qquad) \\
&\ \wedge (\exists r_0 \exists L_0 (\\
&\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\ \mathbf{E}(\neg \exists t(\mathtt{mov}(r_0,t) \vee \mathtt{lea}(r_0,t))\ \mathbf{U} \#\mathtt{loc}(L_0 \\
&\qquad)) \wedge \\
&\quad \mathbf{EF}(\mathtt{push}(r_0) \wedge \#\mathtt{loc}(L_0) \wedge \mathbf{EX}\ \mathbf{E}(\neg \exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
&\qquad \mathbf{U}(\mathtt{call}(\mathtt{CopyFileA}) \wedge \#\mathtt{loc}(L_c))) \\
&\quad)) \\
&\ \wedge \mathbf{EF}(\#\mathtt{loc}(L_m) \wedge \mathbf{EF} \#\mathtt{loc}(L_c)) \\
&)
\end{aligned}
$$

FIGURE 1.12 – 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 1.13 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`.

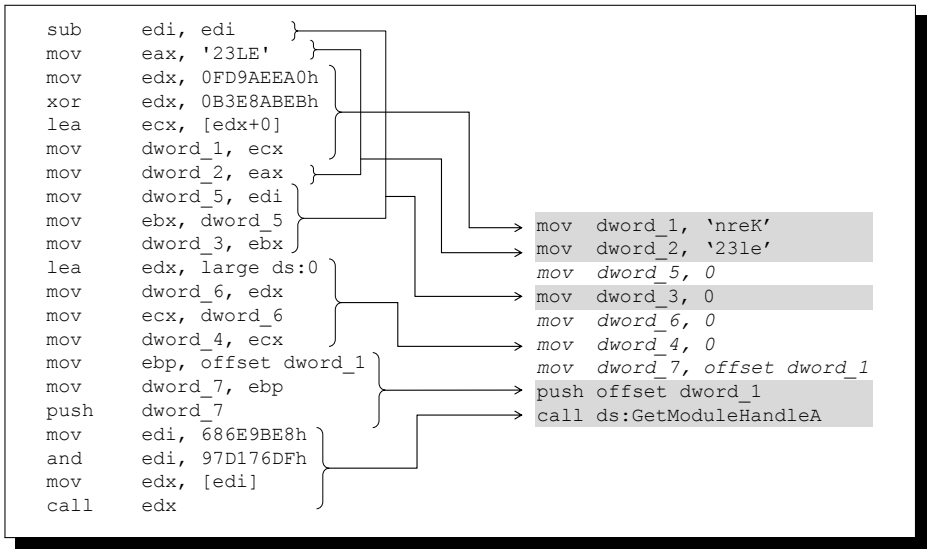

FIGURE 1.13 – Illustration d'une réduction de code sur le virus METAPHOR.

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.* [25] 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.* [127] 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.* [131, 132] 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 [53]. 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 [67]. 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.* [22] 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 [24], 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.* [107] 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.* [24] 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.* [12, 13] 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 [12] inspiré des travaux de Kruegel *et al.* [74], 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 [47].

Dans [13], la similarité entre les graphes est inspirée d'autres travaux [71] 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.* [140] 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 [75], 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.* [10] 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 [35] 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.* [64] : « *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 ob-*

servables 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 *pertinent*e. 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'informations 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* [120] 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 [85], Cobra [122], DynamoRIO [50], Valgrind [98], Diota [87], 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 peut alors tourner avec des privilèges restreints et se voir 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 [100] 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 [12].

1.3.3.1.c Les machines virtuelles

D'après Goldberg [54], 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 [78] 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 [13], VirtualBox [130], VirtualPC [14] et QEMU [9]. Parmi les outils d'analyse s'appuyant sur des machines virtuelles, les plus utilisés sont sans doute Anubis [6] s'appuyant sur QEMU, et CWSandbox [134] 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 [42] 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

12. 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.
13. Disponible à l'URL : `http://www.vmware.com/` (dernier accès en décembre 2010)
14. Disponible à l'URL : `http://www.microsoft.com/windows/virtual-pc/` (dernier accès en décembre 2010).

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, 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 [38] s'appuyant sur l'hyperviseur Xen [5], 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.* [36].

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 [14], 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.* [65] 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 1.14 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 1.14 – Exemple de grammaire attribuée décrivant le comportement de duplication tiré de [65].

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 1.15 illustre une partie du comportement du vers NETSKY décrit sous forme d'un graphe représentant les appels systèmes observés ainsi que leurs dépendances.

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.

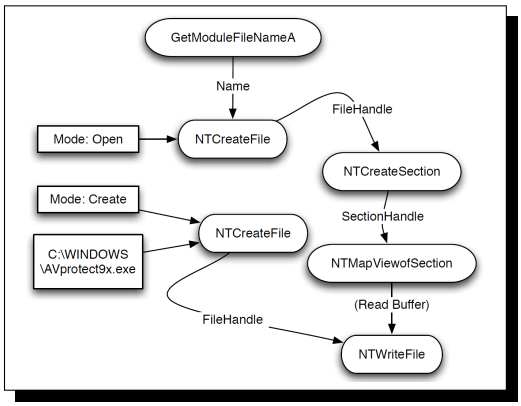

FIGURE 1.15 – Illustration de d'un graphe de dépendances entre les appels systèmes tiré de [70] représentant le vers NETSKY.

1.3.3.3.a Approche de Yin *et al.*

Yin *et al* [138] proposent une architecture de détection dynamique par émulation de code reposant sur QEMU [9]. Leur solution s'appuie sur la technique dite de « *data tainting* » [99], 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.

Parler d'Argod [104].

1.3.3.3.b Approche de Kolbisch *et al.*

Kolbisch *et al.* [70] 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* » [133] 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.* [48] 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 d'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 du système d'exploitation sur lequel il s'exécute, de ses droits et privilèges, ainsi que des applicatifs présents, etc. Ce problème est abordé par Moser *et al.* [95]) 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 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 1.16 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.

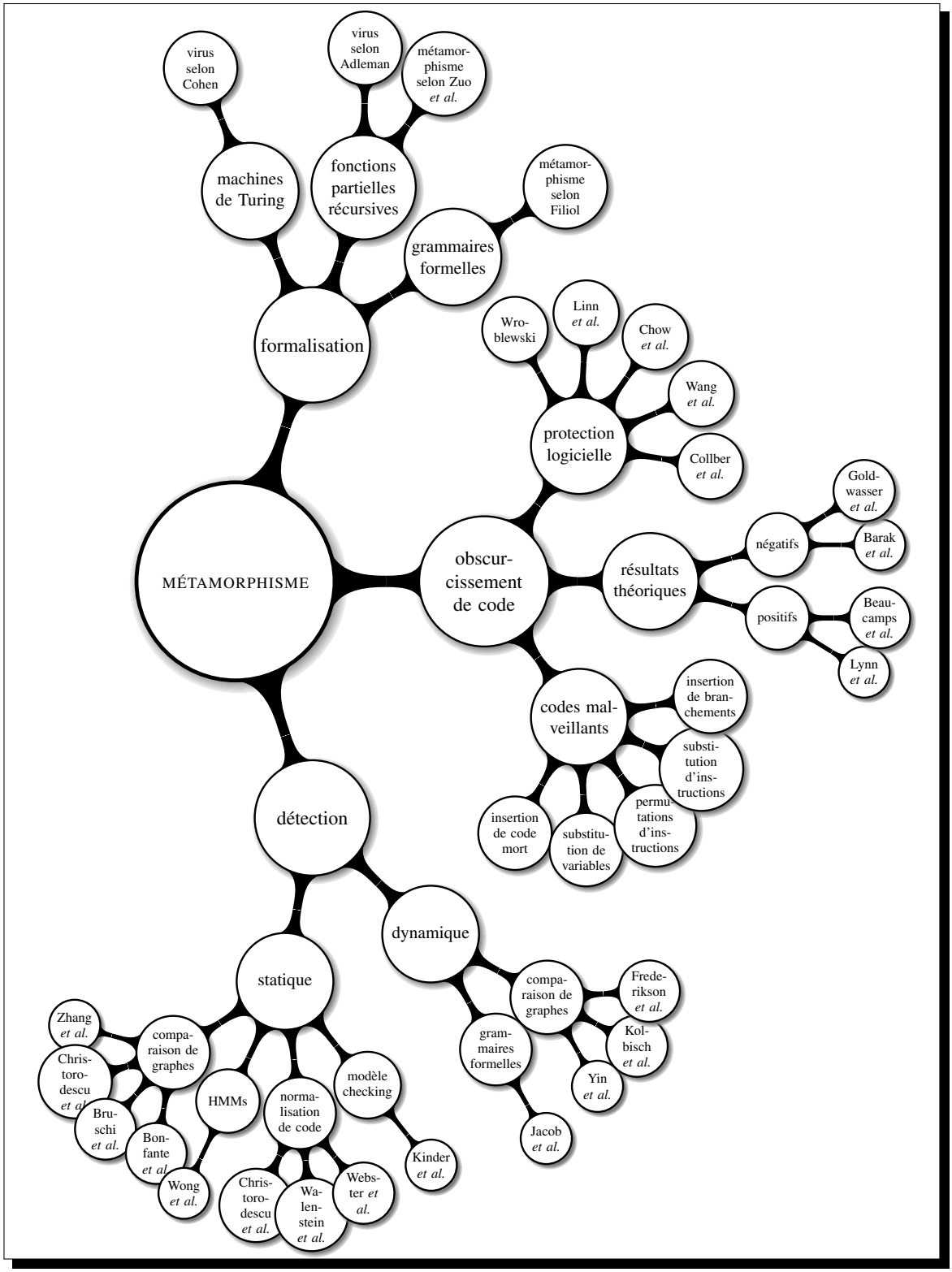

FIGURE 1.16 – Schéma récapitulatif de l'état de l'art sur le métamorphisme

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] G. Arboit. A Method for Watermarking Java Programs via Opaque Predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[4] 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.

[5] 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.

[6] 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.

[7] P. Beaucamps. Advanced Polymorphic Techniques. *International Journal of Computer Science*, 2(3) :194–205, 2007.

[8] P. Beaucamps and E. Filiol. On the possibility of practically obfuscating programs. *Journal in Computer Virology*, 3(1) :3–21, 2007.

[9] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[10] G. Bonfante, M. Kaczmarek, and J.Y. Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 5(3) :263–270, 2009.

[11] L. Bridges. The changing face of malware. *Network Security*, 2008(1) :17–20, 2008.

[12] 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.

[13] D. Bruschi, L. Martignoni, and M. Monga. Code Normalization for Self-Mutating Malware. *IEEE Security & Privacy*, 5(2) :46–54, 2007.

[14] S. Buhlmann. Extending joebox-a scriptable malware analysis system. Master's thesis, University of Applied Sciences Northwestern Switzerland, 2008.

[15] A.W. Burks. *Essays on Cellular Automata*. University of Illinois Press, 1970.

[16] 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.

[17] T.M. Chen and J.M. Robert. Worm Epidemics in High-Speed Networks. *Computer*, 37 :48–53, 2004.

[18] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions ond*, 2(3) :113–124, 1956.

[19] N. Chomsky. On Certain Formal Properties of Grammars. *Information and control*, 2(2) :137–167, 1959.

[20] 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.

[21] 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.

[22] 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.

[23] M. Christodorescu and S. Jha. Testing Malware Detectors. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 2004.

[24] 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.

[25] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware Normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.

[26] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[27] C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software : Practice and Experience*, 25(7) :811–829, 1995.

[28] R. Cilibrasi and P.M.B. Vitányi. Clustering by compression. *IEEE Transactions on Information theory*, 51(4) :1523–1545, 2005.

[29] E. Clarke, O. Grumberger, and D. Long. *Model Checking*. MIT Press, 1999.

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

[31] 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.

[32] 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.

[33] 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.

[34] C.S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. *IEEE Transactions on software engineering*, 28 :735–746, 2002.

[35] 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).

[36] 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.

[37] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6) :644–654, 1976.

[38] 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.

[39] M. Driller. Metamorphism in practice. *29A Magazine*, 1(6) :–, 2002.

[40] J. Earley. An Efficient Context-Free Parsing Algorithm. *ACM Communication*, 13(2) :94–102, 1970.

[41] T. Fawcett. ROC Graphs : Notes and Practical Considerations for Researchers. *Machine Learning*, 31 :1–38, 2004.

[42] P. Ferrie. Attacks on More Virtual Machine Emulators, 2007.

[43] A. Fiat and A. Shamir. How to Prove Yourself : Practical Solutions to Identification and Signature Problems. In *CRYPTO*, pages 186–194, 1986.

[44] É Filiol. Metamorphism Formal Grammars and Undecidable Code Mutation. *International Journal of Computer Science*, 2(1) :70–75, 2007.

[45] É Filiol. *Techniques virales avancées*. Springer, 2007.

[46] É. Filiol. *Les virus informatiques : théorie, pratique et applications*. Seconde édition, Springer, 2009.

[47] P. Foggia. The VFLIB graph matching library, version 2.0, 2001.

[48] 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.

[49] M.R. Garey and D.S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness. A Series of Books in the Mathematical Sciences*. WH Freeman and Company, San Francisco, Calif, 1979.

[50] T. Garnett. Dynamic Optimization of IA-32 Applications Under DynamoRIO.

[51] A. Gazet. Comparative analysis of various ransomware virii. *Journal in computer virology*, 6(1) :77–90, 2010.

[52] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik*, 38(1) :173–198, 1931.

[53] J. Goguen and G. Malcolm. *Software Engineering with OBJ : Algebraic Specification in Action*. Kluwer Academic Publishers, 2000.

[54] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6) :34–35, 1974.

[55] 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.

[56] S. Goldwasser and G.N. Rothblum. On Best-Possible Obfuscation. *Theory of Cryptography*, 4392 :194–213, 2007.

[57] 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.

[58] R.K. Guy. *Unsolved problems in number theory*. Springer Verlag, 2004.

[59] M. H. Halstead. *Elements of Software Science*. Elsevier North Holland, 1977.

[60] D. Harley, U.E. Gattiker, and R. Slade. *Virus : définitions, mécanismes et antidotes (Systèmes et réseaux Coll. Référence)*. Campus Press, 2002.

[61] M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc. New York, NY, USA, 1977.

[62] S. Horwitz. Precise Flow-insensitive May-alias Analysis is NP-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1) :1–6, 1997.

[63] ITSEC. Evaluation Criteria of the Information System Security. Technical report, Office des publications officielles des Communautés Européennes, 1991.

[64] 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.

[65] 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.

[66] N.D. Jones. *Computability and complexity : from a programming perspective*. The MIT Press, 1997.

[67] 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.

[68] 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.

[69] S.C. Kleene. On Notation for Ordinal Numbers. *Journal of Symbolic Logic*, 3(4) :150–155, 1938.

[70] 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.

[71] 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.

[72] J. Kraus. Selbstreproduktion bei Programmen. Master's thesis, Universit "at Dortmund Fachschaft Informatik, 1980.

[73] J. Kraus. On self-reproducing computer programs. *Journal in Computer Virology*, 5 :9–87, 2009.

[74] 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.

[75] H.W. Kuhn. The Hungrian Method for the Assignment Problem. *Naval Research Logistics Quaterly*, 2 :83–97, 1955.

[76] A. Lakhotia, A. Kapoor, and EU Kumar. Are metamorphic viruses really invincible ? In *Virus Bulletin*, pages 5–7, 2004.

[77] W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4) :337, 1992.

[78] K.P. Lawton. Bochs : A portable pc emulator for unix/x. *Linux Journal*, 1996(29es) :7, 1996.

[79] M. Lesk. The New Front Line : Estonia under Cyberassault. *Security & Privacy, IEEE*, 5(4) :76–79, 2007.

[80] H.R. Lewis and C.H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[81] D. Lin. Hunting for Undetectable Metamorphic Viruses. *Journal in Computer Virology*, X :X, 2011.

[82] 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.

[83] E. Littré. *Dictionnaire de la langue française : Supplément*. Hachette, 1886.

[84] D. Low. Java Control Flow Obfuscation. Master's thesis, Master of Science Thesis, Department of Computer Science, The University of Auckland, 1998.

[85] 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.

[86] B. Lynn, M. Prabhakaran, and A. Sahai. Positive Results and Techniques for Obfuscationarboit. In *Advances in Cryptology-EUROCRYPT 2004*, pages 20–39. Springer, 2004.

[87] 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.

[88] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 2(4) :308–320, 1976.

[89] Microsoft. Microsoft Security Intelligence Report volume 5, January through June 2008.

[90] Microsoft. Microsoft Security Intelligence Report volume 7, January through June 2009.

[91] Microsoft. Microsoft Security Intelligence Report volume 6, July through December 2008.

[92] Microsoft. Microsoft Security Intelligence Report volume 8, July through December 2009.

[93] 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.

[94] T. Moore, R. Clayton, and R. Anderson. The Economics of Online Crime. *The Journal of Economic Perspectives*, 23(3) :3–20, 2009.

[95] 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.

[96] 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.

[97] M. Munson Taghi and C. John. Measurement of data structure complexity. *Journal of Systems and Software*, 20(3) :217–225, 1993.

[98] N. Nethercote and J. Seward. Valgrind : : A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2) :44–66, 2003.

[99] 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.

[100] Norman. Norman SandBox Whitepaper, 2003. Disponible à l'URL suivante : http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf (dernier accès en décembre 2010).

[101] 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.

[102] E. I. Oviedo. Control flow, data flow, and program complexity. In *IEEE COMPSAC*, pages 146–152, 1980.

[103] C.H. Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[104] G. Portokalidis, A. Slowinska, and H. Bos. Argos : an Emulator for Fingerprinting Zero-Day Attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4) :15–27, 2006.

[105] E.L. Post. Recursive Unsolvability of a Problem of Thue. *Journal of Symbolic Logic*, 12(1) :1–11, 1947.

[106] 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.

[107] 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.

[108] 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).

[109] L.R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2) :257–286, 1989.

[110] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1467–1471, 1994.

[111] 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.

[112] H. Rogers Jr. *Theory of recursive functions and effective computability*. MIT Press Cambridge, MA, USA, 1987.

[113] 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.

[114] J.F. Shoch and J.A. Hupp. The "Worm" Program–Early Experience With a Distributed Computation. *Communications of the ACM*, 25(3) :172–180, 1982.

[115] 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.

[116] E. H. Spafford. The Internet worm incident. In *2nd European Software Engineering Conference*, pages 446–468, 1989.

[117] D. Spinellis. Reliable Identification of Bounded-length Viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1) :280–284, 2003.

[118] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[119] P. Szor and P. Ferrie. Hunting for metamorphic. In *Virus Bulletin*, volume 123, 2001.

[120] J. Toger. *Specification-Driven Dynamic Binary Translation*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2004.

[121] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1) :230, 1937.

[122] A. Vasudevan and R. Yerraballi. Cobra : Fine-grained Malware Analysis using Stealth Localized-executions. In *IEEE Symposium on Security and Privacy*, 2006.

[123] J. Von Neumann. The general and logical theory of automata. In *Cerebral Mechanisms in Behavior : The Hixon Symposium*, pages 1–41, 1951.

[124] J. Von Neumann, A.W. Burks, et al. *Theory of self-reproducing automata*. Univ. of Illinois Press Urbana, IL, 1966.

[125] VX Heavens repository. Disponible a l'URL suivante : http://vx.netlux.org/ (dernier accès en décembre 2010).

[126] 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.

[127] 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.

[128] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of Software-based Survivability Mechanisms. In *IEEE Computer Society*, 2003.

[129] 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.

[130] J. Watson. Virtualbox : bits and bytes masquerading as machines. *Linux Journal*, 2008(166) :1, 2008.

[131] M. Webster and G. Malcolm. Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, 2(3) :149–161, 2006.

[132] M. Webster and G. Malcolm. Detection of Metamorphic and Virtualization-based Malware using Algebraic Specification. *Journal in Computer Virology*, 5(3) :221–245, 2009.

[133] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[134] 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.

[135] W. Wong and M. Stamp. Hunting for Metamorphic Engines. *Journal in Computer Virology*, 2(3) :211–229, 2006.

[136] 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.

[137] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[138] 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.

[139] P.V. Zbitskiy. Code mutation techniques by means of formal grammars and automatons. *Journal in Computer Virology*, 5(3) :199–207, 2009.

[140] Q. Zhang and D.S. Reeves. MetaAware : Identifying metamorphic malware. In *23rd Annual Computer Security Applications Conference (ACSAC)*, pages 411–420. IEEE, 2007.

[141] 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.

[142] 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.

[143] 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.

[144] Z. Zuo and M. Zhou. Some Further Theoretical Results about Computer Viruses. *The Computer Journal*, 47(6) :627–633, 2004.

[145] Z. Zuo, Q. Zhu, and M. Zhou. On the Time Complexity of Computer Viruses. *IEEE Transactions on information theory*, 51(8) :2962–2966, 2005.

VU :
Le Directeur de Thèse

VU :
Le Responsable de L'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,