Éric Filiol

Collection IRIS
dirigée par Nicolas Puech

Techniques virales avancées

Springer

ESAT

INRIA

Techniques virales avancées

Springer
Paris
Berlin
Heidelberg
New York
Hong Kong
Londres
Milan
Tokyo

Éric Filiol

Techniques virales avancées

Springer

Éric Filiol
Chef du laboratoire de virologie et cryptologie
École Supérieure et d'Application des Transmissions
B.P. 18
35998 Rennes Armées

et INRIA-Projet Codes

ISBN 13: 978-2-287-33887-8 Springer Paris Berlin Heidelberg New York

Printed in France
Springer-Verlag France est membre du groupe Springer Science + Business Media

SPIN: 11748564

Maquette de couverture : Jean-François MONTMARCHÉ

À ma femme Laurence,
à mon fils Pierre,
à mes parents.

Avant-propos

« Quand le sage montre la lune, l'idiot regarde le doigt. »

Proverbe chinois

« Tout individu a droit à la liberté d'opinion et d'expression, ce qui implique le droit de ne pas être inquiété pour ses opinions et celui de chercher, de recevoir et de répandre, sans considérations de frontières, les informations et les idées par quelque moyen d'expression que ce soit. »

Article 19 - Déclaration universelle des droits de l'Homme

« Les dictatures fomentent l'oppression, la servilité et la cruauté ; mais le plus abominable est qu'elles fomentent l'idiotie. »

J.-L. Borges

« Le véritable maître d'une chose n'est pas celui qui la possède mais celui qui peut la détruire. »

Frank Herbert

Le présent ouvrage traite des techniques virales dites « évoluées » et fait suite à l'ouvrage *Les virus informatiques : théorie, pratique et applications.* L'idée de départ était de rédiger ce second volume dans la continuité du premier, tant sur le fond que sur la forme : présenter des techniques virales plus complexes et les illustrer par des codes connus ou des codes conçus spécialement par l'auteur. Le succès du premier tome indiquait clairement cette voie. Présenter l'algorithmique virale et les concepts qui en découlent, en s'appuyant sur le code source d'exemples concrets, détaillés et complets, est la seule approche scientifiquement et intellectuellement viable. Elle suppose certes de faire confiance d'emblée au lecteur – lequel utilisera les connaissances et techniques présentées dans un but respectable – car il n'est pas possible de bâtir un monde meilleur sans la confiance et le respect qui l'accompagnent.

Expliquer une science ou une technique en dissimulant ou en dérobant l'essentiel au lecteur sous prétexte de limiter un prétendu risque – somme toute marginal – lié à la prolifération de connaissances dangereuses, relève du fallacieux et ne peut être inspiré que par l'incompétence et/ou la défense d'intérêts corporatistes et partisans. Cela pour un certain nombre de raisons qu'ignorent ou que méprisent ceux qui estiment détenir le monopole de la pensée :

- en terme de science, seule compte réellement la capacité à reproduire les résultats, à les vérifier. Cacher une partie des informations et demander de croire sur parole, revient à manipuler l'esprit et à imposer une vision potentiellement biaisée ou erronée de la réalité. Cela concourt à une asepsie de la pensée. Or l'activité scientifique procède précisément du contraire : dialogue, partage, contradiction voire controverse, bouillonnement des idées, autrement dit échange et communication et donc vie ;
- restreindre les connaissances sous prétexte de limiter le risque qui est lié à leur divulgation est une absurdité totale. Toutes les dictatures ont tenté de gouverner en misant sur l'ignorance. Toutes ont échoué. D'autre part, c'est ignorer les leçons enseignées par l'épistémologie. Les inventions et les découvertes ne sont généralement pas le fait d'un individu unique. La double hélice de l'ADN a été « découverte » par deux équipes simultanément. C'était dans « l'air du temps », voilà tout. D'autres exemples pourraient être cités, qui prouveraient que l'humilité est une règle essentielle en matière de découverte scientifique. À quoi bon cacher ce que d'autres finiront inévitablement par découvrir ?
- c'est enfin nier que l'esprit humain, la plupart du temps, est raisonnable. Une connaissance n'est pas dangereuse en soi. C'est seulement un fait. Seule l'utilisation de ce fait, dans un sens ou dans un autre, soumise au libre arbitre, peut faire l'objet d'un jugement.

Mais toutes ces considérations doivent être ramenées à l'évolution des législations récentes, tant en France [86][1] que dans la plupart des autres pays [134]. Celle-ci rend l'exercice de plus en plus périlleux et commande à plus de prudence et de responsabilité de la part du scientifique et de l'auteur, au moment

[1] L'article 323 du Code pénal prévoit un emprisonnement de trois ans et 45 000 euros d'amende.

de publier. C'est la raison pour laquelle le parti pris initial a été abandonné, à regret. Mais que le lecteur se rassure, la pertinence du propos n'a pas été amoindrie. Sa portée n'a pas été non plus rognée, bien au contraire. Seule la forme a dû s'adapter aux contraintes récentes imposées par l'évolution de la législation dans ce domaine.

L'autre raison qui a incité à modifier son approche tient à l'évolution récente de la littérature consacrée à la virologie. Le livre de Peter Szor [130], en particulier, paru en 2005, présente un certain nombre de techniques que ce second tome se proposait de traiter, certes sous un angle différent et plus technique que celui de Szor, analyste chez l'éditeur d'antivirus Symantec. Il était dès lors inutile d'étudier, même sous un angle plus technique, des sujets déjà traités par d'autres ou du moins d'une manière plus ou moins équivalente. De plus, ce livre et d'autres parus en 2005 – qui n'ont malheureusement pas la qualité et la pertinence de celui de Szor – traitent une fois de plus de techniques virales et antivirales qui relèvent désormais du passé, même s'il s'agit d'un passé immédiat.

Il est apparu plus utile et pertinent de redéfinir les objectifs initialement fixés de ce second tome, et de se projeter dans le futur de la virologie informatique et/ou d'en considérer les aspects généralement insoupçonnés, même par un grand nombre de professionnels du domaine. L'auteur a choisi de présenter des aspects techniques inconnus, ou de revisiter des techniques connues sous un jour totalement différent tout en les généralisant.

Il était enfin temps de sortir du cycle quelque peu stérile « détection, analyse et mise à jour » dans lequel les éditeurs d'antivirus se sont laissés enfermer. La virologie informatique offre par ailleurs des perspectives bien plus intéressantes et constructives que l'activité habituelle des programmeurs de codes malveillants : diffuser un énième ver utilisant une énième vulnérabilité. En plus d'offrir un champ immense de réflexion scientifique, cette science a la capacité de donner naissance à un nombre incalculable d'applications extrêmement utiles. De ce point de vue, il serait plus adéquat de substituer la notion de code ou d'agent mobile[2] à celle de virus ou de ver.

Les fondements de l'algorithmique virale ayant été présentés dans le premier tome, les retours des lecteurs ainsi que l'expérience ont démontré qu'il était plus intéressant de conserver une approche plus mathématique et algorithmique. Somme toute, s'il est pratique de présenter un code pour une plate-forme donnée, pour un langage particulier, cela suppose également qu'un problème général soit restreint à une ou plusieurs de ses instances particulières. En revanche, présenter le problème sous l'angle général, en identifier les ressorts algorithmiques, notamment au regard de la théorie de la complexité, est infiniment plus puissant et universel. C'est précisément cette incapacité à briser la dépendance vis-à-vis d'une vision liée à une plate-forme donnée ou à un langage donné, qui explique le relatif échec des éditeurs d'antivirus [129] en matière de lutte antivirale. Leur incapacité à penser en terme de modèle et

[2] L'auteur reprends ici l'idée suggérée par Jean-Marie Borello, étudiant en thèse au laboratoire de virologie et de cryptologie.

d'algorithmique les enferme dans un monde techniquement très pauvre[3].

L'évolution des attaques par codes malveillants, depuis 2003-2004 montre que l'ère des grandes épidémies va probablement céder progressivement la place à des menaces plus sophistiquées et cachées. Les motivations des attaquants évoluant vers l'appât du gain sous toutes ses formes – argent, informations monnayables... –, les créateurs des codes malveillants cherchent à être les plus discrets possible pour ne pas être détectés par les logiciels de sécurité (antivirus, pare-feux, systèmes de détection d'intrusion). Cela leur permet d'agir dans la durée. Le but n'est plus de détruire des données ou de rendre indisponibles des services, mais de se livrer à l'espionnage, au vol et à l'extorsion. Nous sommes entrés dans l'ère des attaques ciblées et des temps sombres s'annoncent pour les éditeurs de logiciels de sécurité.

Cet ouvrage présente donc des techniques virales évoluées, voire novatrices, sous l'angle des modèles et de la théorie de la complexité. Le fil conducteur de cet ouvrage est de montrer que **la technique sera toujours en faveur de l'attaquant**, que ce soit pour pénétrer un système, se maintenir dans ce système, y agir à sa guise et en faire sortir les données qu'il souhaite. Le défenseur aura toujours contre lui les lois mathématiques, lesquelles peuvent être favorables à l'attaquant : c'est là tout l'objet du présent ouvrage.

Cela signifie que le salut, pour le gardien d'un système n'est pas dans la technique mais dans la politique de sécurité qu'il définira, les règles qui en découleront, leur application, leur contrôle et l'adhésion des décideurs. En d'autres termes, seul l'humain et l'organisationnel peuvent espérer vaincre la technique et la prendre à son propre jeu. C'est somme toute un constat rassurant. Mais le prix à payer est élevé : sacrifice de l'ergonomie sur l'autel de la sécurité, contraintes organisationnelles, gestion par la prééminence de l'humain sur le matériel et la course au rendement... Dans la guerre moderne qui fait rage, la ressource critique est l'information. De sa sécurité dépendent des emplois, des parts de marché, quelquefois des vies humaines..., autrement dit, le patrimoine d'une entreprise ou d'un État.

L'officier de sécurité doit prendre le pas sur le contrôleur de gestion. Le danger, dans ce contexte, ne réside plus dans l'attaque grand public par un énième ver ou code malveillant, utilisant la dernière vulnérabilité détectée – même si ces attaques peuvent encore causer des dégâts –, mais dans les attaques ciblées, dirigées contre des personnes, des sociétés ou des administrations dans le but de leur porter préjudice, quelquefois gravement.

L'expérience du terrain – notamment à travers des expertises judiciaires – et les expérimentations en laboratoire montrent qu'aucun logiciel de sécurité (antivirus, pare-feux...) n'est ni ne peut être capable de détecter de telles attaques. Or la menace est réelle : mafias, États-voyous (ou non), organismes de renseignement privés ou étatiques utilisent de plus en plus ces techniques.

[3] Le meilleur exemple est celui d'un « expert » en virologie informatique qui a écrit en décembre 2005 qu'un code source était inutile pour mettre à jour les antivirus... Ces derniers utilisent en effet essentiellement des techniques encore naïves d'analyse de forme, qui imposent de disposer d'une instance compilée du code *(sic)*.

L'affirmer, bien sûr, ne suffit plus, il faut le démontrer tant le scepticisme est encore grand, notamment chez les décideurs victimes de certains « consultants » et autres marchands du temple. Et plus que tout, il serait irresponsable de laisser les victimes potentielles dans l'ignorance.

Cet ouvrage s'articule autour de deux parties. La première partie présente les fondements théoriques essentiels nécessaires pour comprendre comment une attaque ciblée, indétectable en pratique, est préparée et construite. Le chapitre 2 traite de l'analyse de la protection antivirale et de la façon dont telle ou telle technique est réellement mise en œuvre par un produit antivirus donné. Aucune attaque sérieuse ne peut, de nos jours, être lancée sans une connaissance préalable et précise des protections que l'attaquant va rencontrer, selon le principe célèbre : « Si tu veux combattre ton ennemi, connais-le. »

Le chapitre 3 s'attache ensuite à modéliser les techniques antivirales actuelles et à montrer les limites, incontournables, de ce modèle. Une attaque sophistiquée sera toujours construite de sorte à ce que ces limites jouent en sa faveur. Le chapitre 4 traite enfin d'une nouvelle variété de codes malveillants qui offre aux attaquants un potentiel inouï : les virus k-aires ou virus combinés. Ces codes avaient été présentés dans le premier volume mais il a paru nécessaire d'étudier plus avant ce qui risque d'être la menace de demain.

La seconde partie s'attache à illustrer les différents aspects et phases d'une attaque ciblée possédant un haut degré de sophistication en considérant quelques problèmes connus, parmi de nombreux possibles, dont la complexité de résolution est telle qu'aucun analyste ou programme antiviral ne peut les résoudre en pratique.

L'organisation générale de cette partie correspond à peu près aux différentes phases d'une telle attaque. Les deux chapitres suivants traitent de la résistance à la détection soit par polymorphisme/métamorphisme (chapitre 6), soit par des techniques de furtivité (chapitre 7). Ce dernier chapitre considère également la pénétration du système ou les moyens permettant d'en sortir des données. Enfin, le chapitre 8 présente les techniques de blindage de code qui permettent de résister à l'analyse de code voire à l'interdire (désassemblage, débogage...). Ces techniques rendent donc très difficiles voire impossibles non seulement la mise à jour des antivirus et autres logiciels de sécurité, mais également l'identification des actions menées par le code malveillants (vol de données par exemple).

Toutes les techniques présentées dans cette partie ont été testées avec succès en laboratoire à l'aide de codes « preuves de concept ». Ces expériences illustrent que **la sécurité ne peut ni ne doit reposer uniquement sur des produits techniques à l'efficacité limitée par essence, mais sur une politique de sécurité qui doit imposer aux réseaux les plus sensibles d'être isolés et qui, en conséquence, doit gérer au mieux les contraintes résultant de cet isolement.**

Ce livre s'adresse essentiellement aux étudiants de troisième cycle, aux chercheurs dans les domaines de l'informatique théorique et de la sécurité de l'information. Il s'adresse également aux ingénieurs travaillant dans ce dernier

domaine, aux administrateurs système et/ou réseau ainsi qu'aux officiers de sécurité.

Certains aspects traités dans ce volume sont présentés dans le cours supérieur de virologie donné par l'auteur à l'École Supérieure d'Électricité de Bretagne, l'École Nationale Supérieure des Techniques Avancées, l'École Nationale Supérieure des Télécommunications de Bretagne et l'École Supérieure et d'Application des Transmissions. Les prérequis sont une bonne connaissance de la théorie de la complexité, des probabilités et des statistiques. Une bonne maîtrise de l'algorithmique et de la programmation est préférable, notamment pour les projets et exercices proposés en fin de chaque chapitre.

Conscient qu'un certain nombre d'imperfections peuvent subsister dans ce présent ouvrage, malgré le soin apporté à sa rédaction et à sa relecture, l'auteur prie les lecteurs de bien vouloir l'en excuser et les invite d'ores et déjà à lui signaler les éventuelles (mais inévitables) coquilles trouvées, afin d'améliorer progressivement cet ouvrage. Elles seront corrigées au fur et à mesure sur la page web suivante : `www-rocq.inria.fr/codes/\'Eric.Filiol/index.html`.

Ce livre est avant tout dédié aux lecteurs du premier volume qui ont largement contribué à son succès et ont rendu possible la présente suite. Les contacts et retours ont été nombreux, constructifs et toujours stimulants. Merci à vous tous. L'auteur forme le vœu que ce second opus vous comble tout comme le premier.

Ce livre est ensuite dédié au général Max Mayneris. Il a contribué, d'une certaine manière et sans le savoir, à ce que ce livre et le précédent voient le jour. Je garde de lui le souvenir d'un homme cordial et passionné mais discret. Aussi, ce témoignage de gratitude le sera-t-il tout autant.

Enfin, je tiens à remercier, pour des raisons diverses, les personnes qui ont rendu ce livre possible : avant tout Nathalie Huilleret, Nicolas Puech et Guido Zozimo Landolfo des éditions Springer, qui ont sans l'ombre d'une hésitation été séduits par le projet de suite au livre précédent sur les virus ; les sous-lieutenants Fall (Sénégal) et Doromon Ndiekor (Tchad), les lieutenants de vaisseau Hansma et de Drezigue, les chefs de bataillon Nissé et de Lizoreux (qui ont participé au développement de certaines versions des virus ou à l'implémentation de certaines techniques présentées dans ce livre, lors de leur stage d'ingénieur, au sein du laboratoire de virologie et de cryptologie de l'École Supérieure et d'Application des Transmissions), mais également le général de division Damien Bagaria, le colonel Annick Reto, le lieutenant-colonel Bruno Troussard, le cours SSIC de l'ESAT, qui, tous, à leur façon, m'ont soutenu dans cette entreprise et ont compris la nécessité de développer, chez les futurs professionnels du ministère de la Défense, une véritable culture technique éclairée, en matière de virologie informatique.

Je souhaite également remercier tout ceux qui m'ont d'une manière ou d'une autre soutenu : Guillaume Arcas, Erwan Attié, Philippe Beaucamps, Guillaume Bonfante, Méline Berthelot, Christophe Bidan, Vlasti Broucek, Nicolas Brulez, Jean-Luc Casey, Stéphane Clodic, Rainer Fahs, Jean-Paul Fizaine, Grégoire Jacob, Sébastien Josse, Brigitte Jülg, Claude Kirchner, Alexis La Goutte, Mo-

hammed Lambarki, Mickaël Le Liard, Pierre Loidreau, Maryline Maknavicius-Laurent, Thierry Martineau, Jean-Yves Marion, Ludovic Mé, Arnaud Metzler, Claude Petit, Frédéric Raynal, Jean-Marc Steyaert, Cédric Tavernier, Paul Turner, Frédéric Weiss, et tout ceux que j'ai oublié, pour m'avoir permis de faire partager à d'autres cette passion, tous mes élèves et étudiants sans lequels les cursus créés n'auraient pas vu le jour. Enfin, ma femme Laurence qui a relu avec minutie et compréhension le manuscrit et m'a soutenu avec patience dans cette entreprise.

Guer, Octobre 2006

Éric Filiol
Eric.Filiol@inria.fr
efiliol@esat.terre.defense.gouv.fr

Table des matières

Table des illustrations

Liste des tableaux

Fondements théoriques

Chapitre 1

Introduction

Le terme de sécurité décrit le duel incessant opposant l'attaquant au défenseur. Il en découle logiquement que défendre sans connaître les modes d'action et de pensée de l'attaquant est tout aussi illusoire que de prétendre attaquer tout en ignorant l'état de la défense qui est opposée. Les deux mondes s'enrichissent mutuellement. Cependant, la perception selon laquelle ce « duel » serait incessant – ni armes absolues, ni protection absolue – doit être quelque peu revue et amendée. En réalité, l'attaque dispose de plusieurs avantages sur le défenseur. Lesquels sont-ils ?

- Le premier est l'élément de surprise. L'attaquant a non seulement toujours l'initiative mais il innove constamment. Son efficacité dépend fortement de sa capacité à dérouter celui qui défend par une approche novatrice et inattendue. Il s'efforcera de passer par là où on l'attend le moins.
- L'attaquant est l'ennemi des certitudes, surtout celles du défenseur. Il souscrit éventuellement à celle selon laquelle il est toujours possible, avec suffisamment d'acharnement et d'ingéniosité, de percer les protections en place – que ces dernières soient uniquement techniques ou non. Aussitôt qu'une chose est réputée impossible[1], l'attaquant saura que c'est par là qu'il faut passer. Un bon exemple est celui des virus de démarrage : le passage à des systèmes évolués, dialoguant directement avec les pilotes de périphériques, devait les rendre impossible. Si ces virus ont effectivement pratiquement disparu – du moins pour ce que nos antivirus peuvent en dire –, les auteurs de codes preuve-de-concept ont montré avec des virus comme *Joshi* ou *March 6*, puis en 2005 avec le projet *BootRootkit* que le risque était loin d'avoir disparu.
- L'attaquant a en quelque sorte une obligation de résultat et, en outre, il est en mesure de prouver ce qu'il prétend. Réussir une attaque est une preuve en soi. Il suffit de la mener à son terme. Cela est impossible pour celui qui défend : affirmer que son système est protégé ou qu'il est sûr ne peut être prouvé. C'est tout au plus un espoir. L'attaquant cherchera en

[1] Faisons confiance à certains consultants et commerciaux pour entretenir ce mythe !

général à cacher le plus longtemps possible ce qu'il est parvenu à faire. La notion de vulnérabilité *0-Day* [77] est probablement le meilleur des exemples.
- L'attaquant possède toujours un temps d'avance. C'est en grande partie une conséquence des points précédents mais pas uniquement. La complexité des attaques augmentant, le défenseur a de plus en plus de difficulté à en démonter les mécanismes, comprendre ce qui s'est réellement passé, tirer les conclusions et modifier sa défense en profondeur. Au fond, il est en général très difficile, pour celui qui protège, de comprendre ce que l'attaquant avait réellement en tête. Tout bon joueur d'échec sait cela. Un plan, une manœuvre ou une action peuvent en dissimuler d'autres.

Tout ces éléments font que le défenseur d'un système, au sens large, est passif et aveugle alors que l'attaquant est actif. Ce qui nous sauve – du moins en ce qui concerne les attaques détectées – c'est que ce dernier est le plus souvent aveugle lui-même. Les deux acteurs s'agitent le plus souvent et n'ont pour seul guide que l'empirisme, même si ce dernier peut, pour l'un et l'autre des acteurs, être quelquefois inspiré voire génial. Au final, on se rend compte que la chance est bien souvent le facteur déterminant.

Cet état de relative voire totale cécité que partagent souvent attaquant et défenseur tient au fait que tous deux n'ont qu'une vision « technicienne » des choses. L'attaquant implémente une idée, quelquefois brillante, mais ignore le contexte global dans lequel il va l'exprimer. Il se polarise le plus souvent sur des aspects purement techniques d'implémentation. Le défenseur quant à lui s'agite, se démène... tout cela pour trouver une protection technique qu'il croit ou espère plus puissante, alors que la solution réelle, si tant est qu'elle existe, n'est déjà plus du domaine technique.

Cela évoque une anecdote (réelle) survenue dans les années quatre-vingt-dix à un sous-officier en opération extérieure au Tchad. Lassé de voir l'un de ses collègues autochtones arborer un gri-gri censé le protéger des balles, ce sous-officier prit le gri-gri, le passa au cou d'un poulet et le tua d'une balle, prouvant toute l'inefficacité de la protection. Deux jours plus tard, le même autochtone revint avec un autre gri-gri. Devant l'exaspération du sous-officier qui tentait de lui expliquer les conséquences logiques de l'expérience de l'avant-veille, l'autochtone répliqua avec une belle assurance que le marabout local lui avait assuré que ce nouveau gri-gri était bien plus puissant ! Ne songeons pas un instant à nous gausser de cet Africain : nous avons nous aussi, dans un autre contexte, nos propres « gri-gris ». Ce sont quelquefois nos outils de sécurité... et nous ne préciserons pas qui sont nos marabouts.

La principale difficulté de la sécurité réside dans la vision trop restreinte que nous avons des problèmes. Nous nous limitons à une gestion de l'anecdotique – les attaques que nous subissons les unes après les autres – alors que nous devrions avoir une vue globale, anticipatrice et la plus universelle possible. Cela n'est possible que par la modélisation et l'approche théorique. Dans le tome précédent [38, chapitre 3], cette approche avait permis de démontrer que le problème de la détection antivirale est un problème généralement indécidable.

La principale et essentielle conséquence est qu'aucune protection totale n'existe ni n'est possible. Mais si ce résultat théorique est souvent éloigné de la réalité, pour un grand nombre d'instances du problème, cela ne signifie pas pour autant que l'approche théorique est sans intérêt. Bien au contraire !

En modélisant un système et les actions sur ce système, ainsi que les objets qui y évoluent, il est possible de dépasser la vision limitée du technicien :

- l'attaquant sera alors en mesure de mettre en évidence les objets, structures et méthodes qui jouent en sa faveur et sont de nature à faire échouer, en pratique, la défense. Ainsi, au lieu de procéder par intuition et tâtonnements, fondant sa réussite soit sur la chance soit sur le fait que le défenseur est moins compétent que lui, il saura, par l'analyse du modèle général, identifier les instances adéquates, de nature à rendre sa détection impossible en pratique ;
- le défenseur quant à lui, par l'analyse de modèles, pourra immédiatement identifier les faiblesses intrinsèques de son système et décider des meilleures protections possibles : techniques, organisationnelles ou les deux. En particulier, dans le cas d'instances du problème de détection complexes, pour lesquelles il n'existe aucune solution technique satisfaisante, il sera en mesure de favoriser les mesures organisationnelles.

Cette partie est donc consacrée à certains aspects fondamentaux de la virologie informatique : modélisation de la défense (les techniques antivirales) et identification de nouvelles techniques virales (codes k-aires). Ces résultats, inédits pour la quasi-totalité d'entre eux, identifient les aspects qui seront considérés par tout attaquant pour monter une attaque indétectable en pratique.

Chapitre 2

L'analyse de la défense

2.1 Introduction

L'essentiel de la protection antivirale repose sur le déploiement d'un ou plusieurs logiciels antivirus. Ces logiciels, de fait, sont la ligne de défense que devra affronter l'attaquant. Il est par conséquent naturel pour ce dernier d'analyser les différents produits qu'il devra contourner. D'une manière duale, tout professionnel devant sécuriser ses systèmes ou réseaux face au risque viral doit également être en mesure d'évaluer l'offre disponible. L'analyse des produits antiviraux est donc une étape importante dans l'évaluation de sécurité au regard du risque infectieux, et ce, quel que soit le côté, attaque ou défense, où l'on se place.

Mais analyser un produit antiviral n'est pas chose facile et les seules mesures que l'on peut avoir sont non seulement peu pertinentes mais également biaisées. Elles sont toujours très difficilement reproductibles, du fait d'une certaine opacité des protocoles de tests, des échantillons testés et des mécanismes cibles de l'évaluation [70]. Èvaluer un produit à l'aide de codes malveillants déjà connus, ou utilisant des techniques connues, est d'une pertinence discutable. Afficher des pourcentages de détection proches de 100 % sans plus de données exploitables, ni permettre la reproductibilité des résultats est scientifiquement contestable.

Se pose alors le problème de savoir comment l'attaquant, qui lui travaille à produire des codes non détectés par les antivirus, peut analyser des produits d'une manière efficace. En somme, pourquoi et comment l'attaquant est-il plus efficace que le défenseur ? Pourquoi parvient-il à déceler les faiblesses permettant le contournement de l'antivirus, là où le défenseur est incapable, du moins en apparence, de les identifier ? Ces deux questions posent le problème de l'analyse et de l'évaluation des logiciels antiviraux. Est-il possible de définir un modèle de test autorisant une analyse rigoureuse, reproductible et universellement acceptée par tous comme une mesure efficace ?

Nous allons résoudre ce problème pour les techniques systématiquement

utilisées, à des degrés divers, par les antivirus : la recherche de signature et plus généralement les techniques d'analyse de forme. La généralisation aux autres techniques de détection (qui relèvent de la détection comportementale) sera ensuite évoquée. Ce chapitre va présenter un modèle générique décrivant toutes les techniques de détection par analyse de forme. L'étude de ce modèle montrera comment identifier les faiblesses exploitables, comment les exploiter mais proposera également pour ce modèle un schéma antiviral sécurisé capable de fortement contrarier l'activité des créateurs de codes malveillants.

2.2 L'analyse de forme

La détection des codes malveillants repose essentiellement sur l'analyse de forme, communément appelée recherche de signatures. Cette dernière appellation est en réalité impropre car elle ne considère qu'une approche très restreinte de l'analyse de forme, laquelle regroupe de nombreuses techniques [130]. L'analyse de forme consiste à analyser un code vu comme une séquence de bits ou d'octets, hors de tout contexte d'exécution. Cette analyse recherche, relativement à une ou plusieurs bases, des chaînes de bits ou d'octets, à la structure plus ou moins complexe, répertoriées dans ces bases comme caractéristiques d'un code malveillant donné.

Plus qu'ils ne l'avouent, quasiment tous les éditeurs d'antivirus du commerce utilisent de manière importante les techniques d'analyse de forme. Leurs dénégations jouent sur le fait que le grand public ne considère qu'une technique particulière et triviale : la recherche de signatures alors que les éditeurs mettent en avant d'autres techniques plus élaborées (voir le chapitre 3) mais conceptuellement identiques à la recherche de signatures. Pour s'en convaincre, il faut réfléchir au fait que dès lors qu'un antivirus est capable de nommer précisément un virus, cela n'est possible que parce que ce nom est référencé dans un fichier (la ou les bases), et qu'au regard de ce nom correspondent un ou plusieurs éléments touchant à sa forme. Sinon, les antivirus ne nommeraient les codes malveillants qu'en termes génériques (généralement liés à une ou plusieurs fonctions) et imprécis.

En réalité, l'analyse de forme est systématiquement utilisée[1] par les antivirus du commerce. L'analyse des produits antiviraux – qui va être présentée en détail dans ce chapitre – le démontre sans équivoque. Cette analyse de forme intervient, comme résumé sur la figure 2.1 (voir également chapitre 3) :

- soit directement à l'aide de techniques dites de première génération (recherche de signatures simples, techniques de caractères génériques [*wildcards*], techniques dites de *mismatch*...). Ces techniques sont essentiellement mises en œuvre par le module de scan statique « à la demande »

[1] Un seul antivirus prétend se passer de toute base liée à l'analyse de forme – et par conséquent prétend ne pas devoir être mis à jour. En réalité, cet antivirus travaille par analyse de comportements et considère des bases de comportements, lesquelles doivent également mises à jour (voir [38, chapitre 5]).

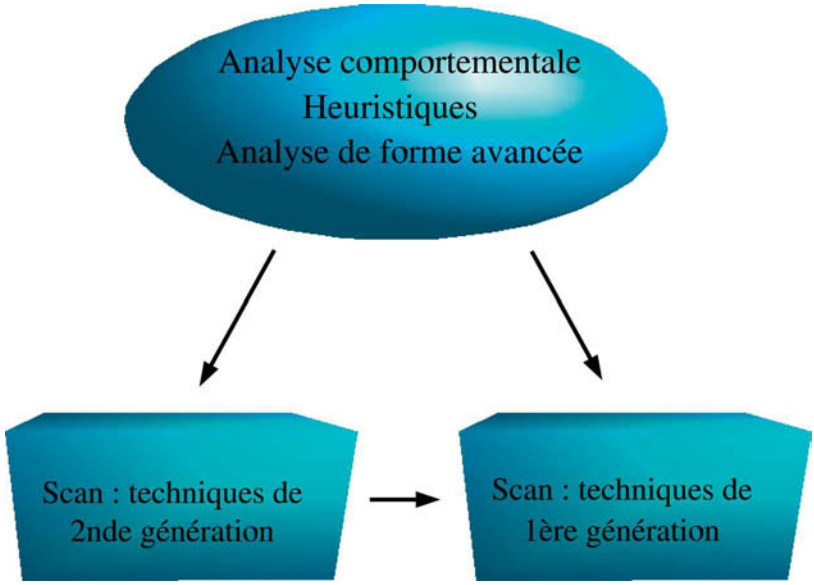


Figure 2.1 – Organisation générale de la stratégie antivirale

(scan manuel) ;

- soit indirectement comme étape finale décisive, ou bien comme phase de validation d'autres techniques de détection appliquées en amont. Ces dernières étant plus incertaines et ayant une approche plus générique (techniques de scan de seconde génération, techniques spécifiques, méthode de décrypteur statique, émulation de code, analyse heuristique [130]...), elles doivent être validées par une analyse de forme qui identifiera spécifiquement et précisément un code donné, grâce à une ou plusieurs bases. Cela concerne à la fois les opérations de scan manuel et d'analyse dynamique (à l'exécution ou lors d'un simple accès).

Cette dépendance très forte – et inévitable en l'état actuel des techniques de détection – vis-à-vis des techniques d'analyse de forme fait que tout programmeur de virus sera capable de contourner assez facilement tout antivirus les mettant en œuvre. Le plus souvent, et l'expérience le prouve malheureusement (voir les résultats fournis en annexe 10), la seule recherche de signatures – une banale chaîne d'octets – est la seule technique réellement utilisée, pour la quasi totalité des produits. Le travail de l'attaquant consiste alors à déterminer quelle est cette signature. Une simple analyse en boîte noire, comme nous allons le voir, suffit. Cette connaissance acquise, il saura alors comment contourner l'antivirus. L'attaquant doit donc résoudre le problème général dit

problème d'extraction de signature. Si extraire une signature constituée d'une simple chaîne d'octets est relativement facile, extraire des signatures plus élaborées est un problème autrement plus complexe. Le cas des signatures simples a déjà été partiellement traité [24]. Nous allons présenter une solution générale à ce problème et caractériser sa complexité calculatoire.

Le problème d'extraction de signature, et sa solution, permettent de mieux comprendre l'état actuel de la menace infectieuse informatique. Un grand nombre de codes malveillants actuels ne sont en fait que des variantes d'une souche initiale. Des familles de vers comme la famille *W32/Bagle* ou *W32/Netsky* comportent plus d'une trentaine de variantes. Le schéma est toujours le même : l'apparition d'une souche originale donne naissance à des variations produites et disséminées par ce qu'il faut bien qualifier de « plagiaires », c'est-à-dire de simples pirates, sans imagination, qui se contentent d'analyser une souche connue, détectée par les antivirus et de la disséminer après l'avoir suffisamment modifiée pour qu'elle ne soit plus détectable. À titre d'exemple, considérons le ver *W32/Blaster* :

- la souche originale *W32.Blaster.A* a fait son apparition le 11 août 2003 ;
- les variantes *W32.Blaster.B* et *W32.Blaster.C* sont apparues le 13 août 2003 ;
- la variante *W32.Blaster.D* est apparue le 19 août 2003 ;
- la variante *W32.Blaster.E* est apparue le 29 août 2003 ;
- la variante *W32.Blaster.F* est apparue le 1 septembre 2003.

En une quinzaine de jours, cinq variantes ont ainsi été produites avec une facilité surprenante. Plus généralement, les statistiques établies mensuellement ou de manière hebdomadaire, par certains professionnels[2], confirment cette tendance. Cette prolifération de codes malveillants, non seulement complique la vie des utilisateurs et des professionnels de la sécurité informatique, mais elle remet éhalement chaque fois en cause la sécurité générale des systèmes et des réseaux. Cela porte également un coup sérieux à la confiance générale dans la sécurité du réseau Internet.

Cette prolifération accroît, enfin, de manière considérable, le travail d'analyse des éditeurs d'antivirus, lesquels finissent par être submergés[3]. Il n'est pas rare, lors de périodes de prolifération intense, de constater une mise à jour des bases de signatures trois ou quatre fois par jour. Cela explique (voir annexe 10) pourquoi les éditeurs mutualisent leurs efforts, partagent les données techniques et, par conséquent, pourquoi les données techniques de détection sont très proches voire identiques d'un éditeur à l'autre. Au final, la sécurité générale des systèmes et des résaux est loin d'être suffisante et acceptable. En 2004, la communauté antivirale a été mise au pilori par le ministère anglais de l'Industrie et du Commerce [49, 129], dont il a dénoncé l'incapacité à lutter efficacement contre la prolifération virale.

[2] Par exemple, le site de *Fortinet*, `http://www.fortinet.com/FortiGuardCenter/global_threat_stats.html`.

[3] Certains ont cherché à automatiser la production de signatures mais en sacrifiant à la qualité générale, notamment concernant la résistance à l'analyse en boîte noire.

Ce qui apparaît comme plus surprenant, c'est l'absence de volonté de la part des éditeurs d'antivirus de réellement contrarier – ou au minimum de tenter de le faire – cette activité de production de variantes à partir de souches ou de variantes détectées. D'une certaine manière, alors que des solutions efficaces existent (voir section 2.6), cette attitude encourage l'activité des plagiaires de codes malveillants et la responsabilité des éditeurs peut être partiellement retenue. Il est tout aussi surprenant que les recommandations techniques de référence définissant les profils de protection [135], ne considèrent pas la résistance à l'analyse en boîte noire comme un critère important dans l'évaluation des produits antivirus [70].

Dans ce chapitre, nous allons étudier le problème de l'extraction des éléments d'analyse de forme, que nous nommerons *schéma de détection* ou encore « *signature* » pour plus de simplicité[4]. Mais nous montrerons que ce terme recouvre une réalité bien plus riche que celle limitée à un ensemble d'octets. Un modèle mathématique général pour la notion de signature, utilisant la notion de *fonction de détection* représentée par des fonctions booléennes, a été élaboré. Il permet d'une part de définir et d'étudier les propriétés que toute « bonne » signature devrait avoir. La résistance vis-à-vis de l'analyse en boîte noire est l'une d'entre elles. D'autre part, ce modèle permet de résoudre le problème de l'extraction de signature, dans sa forme générale (c'est-à-dire pour toute technique d'analyse de forme). La résolution de ce problème se ramène pour l'essentiel à un problème d'apprentissage de fonctions booléennes sous leur forme normale disjonctive (DNF). Si ce dernier problème est encore, dans sa forme générale, un problème ouvert, les résultats ont montré que les antivirus n'en représentent que des instances au mieux faciles sinon triviales. Un attaquant peut toujours très facilement extraire les signatures utilisées par un antivirus et donc facilement contourner l'analyse de forme de ces derniers.

Nous verrons ensuite comment faire en sorte d'interdire toute analyse en boîte noire, et donc en bout de chaîne, la production de variantes non détectées, à moins de supposer qu'un nombre N important d'attaquants ne s'unissent et qu'ils ne disposent d'une puissance de calcul cumulée importante. Différentes constructions sont possibles et ce, sans amoindrir les performances d'exécution ni utiliser des ressources mémoires conséquentes. Ce schéma sécurisé permet en outre, dans une certaine mesure, d'incriminer les auteurs de variantes lors d'enquêtes.

2.3 Modèle mathématique de l'analyse de forme

Le but est ici de définir précisément ce qu'est un schéma de détection ou « signature ». La modélisation aura notamment pour objectif d'appréhender les deux aspects fondamentaux du problème : la défense (une détection efficace)

[4] La notion de schéma de détection – qui comprend entre autres, la chaîne d'octets identifiant spécifiquement un code – est plus puissante. Le terme de signature est inapproprié dans la mesure où il est utilisé dans un sens très précis en cryptologie.

et l'attaque (les auteurs de codes malveillants) ainsi que leur interaction.

2.3.1 Définition d'un schéma de détection

Considérons un fichier quelconque $\mathcal{F}$ de taille n. Il représentera potentiellement un fichier infecté ou directement le code d'un programme malveillant. $\mathcal{F}$ est une séquence d'octets, autrement dit une séquence de n symboles pris dans l'alphabet $\Sigma = \mathbb{N}_{255} = \{0, 1, \ldots, 255\}$. $\mathcal{F}$ est donc un élément de Σ^n.

Nous noterons $\mathcal{S}_{\mathcal{M}}$ un motif de détection de taille s relativement à un code malveillant $\mathcal{M}$. C'est un élément de Σ^s. Ainsi

$$\mathcal{S}_{\mathcal{M}} = \{b_1, b_2, \ldots, b_s\}.$$

Le i-ième octet de $\mathcal{F}$ (respectivement de $\mathcal{S}_{\mathcal{M}}$) sera noté $\mathcal{F}(i)$ (respectivement $\mathcal{S}_{\mathcal{M}}(i)$).

On dit que $\mathcal{F}$ est infecté par $\mathcal{M}$ s'il existe s indices dans $\mathcal{F}$, notés $\{i_1, \ldots, i_s\}$ tels que

$$\mathcal{F}(i_j) = b_{\sigma(j)} \qquad 1 \leq j \leq s,$$

où σ désigne une permutation des octets de $\mathcal{S}_{\mathcal{M}}$. Cette permutation permet de décrire toutes les modifications ou transformations éventuelles opérées par un auteur de codes malveillants sur $\mathcal{M}$. En effet, des techniques d'obfuscation de code (voir le chapitre 8), de polymorphisme [38, 130] peuvent être utilisées pour modifier la structure de $\mathcal{S}_{\mathcal{M}}$ par réarrangement de ses octets. Plusieurs cas sont à considérer :

- la modification de l'indexation des octets de $\mathcal{S}_{\mathcal{M}}$ (par exemple, par insertion de code inutile ou mort [*dummy code*]). Alors nous avons $\sigma = Id_{\mathbb{N}_s^*}$;
- la modification de l'ordre des octets de $\mathcal{S}_{\mathcal{M}}$ (par l'usage d'obfuscation ou de chiffrement par transposition[5]). Alors $\sigma \neq Id_{\mathbb{N}_s^*}$. Toutefois, le flux d'exécution réordonne les octets de $\mathcal{S}_{\mathcal{M}}$ pendant l'exécution de $\mathcal{F}$. Les techniques de scan de seconde génération, les métaheuristiques visent à retrouver ou contourner la permutation σ, tandis que les techniques de scan plus basiques (dite de première génération) assurent l'identification et la détection proprement dite.

Le lecteur remarquera que dans notre approche, nous considérons des indices dans $\mathcal{F}$ où sont localisés effectivement les octets de $\mathcal{S}_{\mathcal{M}}$, à une permutation près de ces octets, plutôt que les octets de $\mathcal{S}_{\mathcal{M}}$ eux-mêmes. Cette approche permet de mieux appréhender celle de tout plagiaire qui souhaiterait produire une variante non détectée à partir d'un code détecté, et ce sans trop d'effort. Il s'intéresse essentiellement aux positions i qu'il lui faudra modifier plutôt qu'à la valeur $\mathcal{F}(i)$ correspondante. Dans un contexte d'analyse en boîte noire, nous noterons $\mathcal{S}_{\mathcal{F},\mathcal{M}}$ l'ensemble des indices $\{i_1, i_2, \ldots, i_s\}$.

[5] Il existe deux techniques de chiffrement : les procédés de substitution dans lesquels les caractères sont remplacés par d'autres selon une convention secrète, et les procédés de transposition, dans lesquels l'ordre des caractères est modifié, également selon une convention secrète. Les deux procédés peuvent être combinés dans le cadre du surchiffrement.

Décrivons maintenant formellement l'action d'un détecteur par analyse de forme $\mathcal{D}$. Pour cela, définissons les s variables binaires X_j $(1 \leq j \leq s)$ comme suit :

$$X_j = \begin{cases} 1 & \text{si } \mathcal{F}(i_j) = b_{\sigma(j)} \\ 0 & \text{autrement.} \end{cases}$$

Cette notation permet de décrire avec précision les modification éventuelles des octets de $\mathcal{S}_\mathcal{M}$ par tout plagiaire tentant de contourner un détecteur donné $\mathcal{D}$. Ainsi, $X_j = 0$ signifie que le plagiaire a modifié $\mathcal{S}_\mathcal{M}(j)$. L'association de tout octet de $\mathcal{S}_\mathcal{M}$ à une variable booléenne permet de considérer l'ensemble booléen $\mathbb{F}_2^{|\mathcal{S}_\mathcal{M}|}$.

Considérons à présent une fonction booléenne $f_\mathcal{M} : \mathbb{F}_2^s \to \mathbb{F}_2$ où $\mathbb{F}_2$ est le corps de Galois à deux éléments[6]. Nous dirons que $\mathcal{D}$ décide que $\mathcal{F}$ est infecté par le code malveillant $\mathcal{M}$, relativement à la *fonction de détection* $f_\mathcal{M}$ et au motif $\mathcal{S}_\mathcal{M}$ si et seulement si $f_\mathcal{M}(X_1, X_2, \ldots, X_s) = 1$. En d'autres termes,

$$f_\mathcal{M}(X_1, X_2, \ldots, X_s) = \begin{cases} 1 & \text{si } \mathcal{F} \text{ est infecté par } \mathcal{M} \\ 0 & \text{si } \mathcal{F} \text{ n'est pas infecté par } \mathcal{M}. \end{cases}$$

Les fonctions de détection seront considérées sous leur forme normale disjonctive (DNF), c'est-à-dire sous la forme d'une union logique (notée $\vee$) de M mintermes constitués de l'intersection logique (notée $\wedge$ ou simplement omis lorsqu'il n'y a pas de risque de confusion) de variables booléennes X_i :

$$f(X_1, X_2, \ldots, X_s) = \bigvee_{j=0}^{M} \Big(\bigwedge_{l \in S_j \subseteq \{1,2,\ldots,s\}} X_{j_l} \Big).$$

Par construction, les variables booléennes n'apparaissent jamais sous la forme $\overline{X_i}$ (négation de X_i). Les fonctions de détection sont par conséquent représentées par des DNF monotones. Nous verrons dans la section 2.4.3, que toute détection par analyse de forme peut être modélisée par une fonction de détection $f_\mathcal{M}$ et un ensemble $\mathcal{S}_\mathcal{M}$. Cela nous conduit à définir la notion générale de *schéma de détection.*

Définition 2.1 *(Schéma de détection)*
Soit un code $\mathcal{M}$. On appelle schéma de détection *de $\mathcal{M}$, la donnée d'une paire $\mathcal{SD} = \{\mathcal{S}_\mathcal{M}, f_\mathcal{M}\}$. La fonction $f_\mathcal{M}$ est appelée* fonction de détection *relativement à $\mathcal{M}$. Dans la détection par analyse de forme, $\mathcal{S}_\mathcal{M}$ est un ensemble d'octets. Dans l'analyse fonctionnelle (analyse comportementale), $\mathcal{S}_\mathcal{M}$ sera un ensemble de fonctions de programmes.*

Remarque. Une base de signature peut être définie comme un ensemble de schémas de détection. En particulier, les motifs de détection, et par là, la taille des fonctions de détection, peuvent varier d'un schéma de la base à un autre.

[6] Autrement dit, il s'agit de l'ensemble $\{0, 1\}$ qui, muni des opérations d'addition et de multiplication, réalise une structure de corps.

Les auteurs de variantes non détectées à partir de codes détectés vont analyser leur schéma de détection pour un ou plusieurs détecteurs donnés. Afin de disposer d'un modèle plus adapté pour la description de l'approche adoptée par le plagiaire, nous considérerons la notion de *schéma de non-détection* ou *schéma de contournement.*

Définition 2.2 *(Schéma de contournement)*
Soit un code $\mathcal{M}$. On appelle schéma de contournement de $\mathcal{M}$, la donnée d'une paire $\mathcal{SC} = \{\mathcal{S}_\mathcal{M}, \overline{f_\mathcal{M}}\}$. La fonction $\overline{f_\mathcal{M}}$ est appelée fonction de non détection *ou* fonction de contournement *relativement à $\mathcal{M}$. Dans la détection par analyse de forme, $\mathcal{S}_\mathcal{M}$ est un ensemble d'octets. Dans l'analyse fonctionnelle (analyse comportementale), $\mathcal{S}_\mathcal{M}$ sera un ensemble de fonctions de programmes.*

La fonction de contournement $\overline{f_\mathcal{M}}$ est en fait la négation de la fonction $f_\mathcal{M}$ soit $1 \oplus \overline{f_\mathcal{M}}$. Cette fonction décrit les différentes possibilités de modifications pouvant être effectuées dans les octets composant $\mathcal{S}_\mathcal{M}$ pour contourner un schéma de détection donné. Ces possibilités correspondent aux s-uplets $(x_1, x_2, \ldots, x_s) \in \mathbb{F}_2^s$ pour lesquels la fonction vaut 1. Pour un s-uplet donné, la modification à faire alors est la suivante :

$$\begin{cases} \text{si } x_i = 0 & \text{l'octet } i \text{ de } \mathcal{S}_\mathcal{M} \text{ doit être modifié} \\ \text{si } x_i = 1 & \text{l'octet } i \text{ de } \mathcal{S}_\mathcal{M} \text{ peut être laissé non modifié.} \end{cases}$$

Remarque. Le code malveillant $\mathcal{M}$ est caractérisé à la fois par le motif de détection $\mathcal{S}_\mathcal{M}$ mais également par la fonction de détection, qui, en quelque sorte, représente le mode de recherche et de validation de $\mathcal{S}_\mathcal{M}$. Ainsi, il est imaginable qu'un fichier donné contienne le motif $\mathcal{S}_\mathcal{M}$ mais que malgré tout, il ne soit pas détecté comme malveillant. C'est là tout l'intérêt de la notion de schéma de détection dans la mesure où la fonction de détection associée, lorsqu'elle n'est pas triviale (voir section 2.4), va limiter les possibilités de fausses alarmes.

2.3.2 Propriétés des schéma de détection

La question naturelle qui se pose est celle concernant les propriétés que doit avoir tout schéma de détection efficace. La qualité finale d'un antivirus donné en dépend. Ces propriétés permettent également d'évaluer l'offre logicielle dans ce domaine, sur une base que l'on souhaite rigoureuse et reproductible. Actuellement, cette évaluation est un processus subjectif, non reproductible par un tiers et souvent les résultats peuvent grandement varier d'une évaluation à une autre, pour un même ensemble de produits. Nous allons présenter les propriétés essentielles qu'un schéma de détection de qualité devrait posséder. Dans la section 2.4, nous verrons comment ces propriétés sont *effectivement* réalisées dans les antivirus du commerce.

Entropie et transinformation

Cette propriété décrit l'incertitude qu'un analyste doit affronter lors d'une analyse en boîte noire relativement à un détecteur $\mathcal{D}$ donné, pour identifier (extraire) le schéma de détection $\{\mathcal{S}_\mathcal{M}, f_\mathcal{M}\}$.

Dans ce but, nous utiliserons la fonction *entropie*, définie par C. E. Shannon [123], en théorie de l'information. Soit une variable X pouvant prendre un ensemble fini de valeurs x_i, chacune avec une probabilité p_i. L'incertitude liée à X, encore appelée entropie de X est définie par :

$$H(X) = -\sum_k p_k \log_2(p_k).$$

Plus l'entropie est faible, moins l'incertitude est importante. Ainsi, à l'extrême, quand nous avons $p_i = 1$ pour un i donné (les autres p_j valent 0, pour $j \neq i$) alors $H(X) = 0$. Il n'y a aucune incertitude car on a toujours $X = x_i$. Dans le contexte du problème de l'extraction d'un schéma de détection, la variable X représente un schéma[7] $\{\mathcal{S}_\mathcal{M}, f_\mathcal{M}\}$ et tous ses paramètres. En effet, l'analyste n'a *a priori* aucune information sur un quelconque des paramètres du schéma (taille s, les octets de $\mathcal{S}_\mathcal{M}$, le nombre de termes et les termes de $f_\mathcal{M}$).

La difficulté à extraire le schéma de détection complet, dans un processus d'analyse en boîte noire, augmente avec $H(\mathcal{S}_{\mathcal{F},\mathcal{M}})$; et si $H(\mathcal{S}_{\mathcal{F},\mathcal{M}}) = 0$, l'analyste n'a à affronter aucune incertitude.

Le fait est que dans notre cas, le calcul de $H(X)$ est très complexe. C'est la raison pour laquelle nous utiliserons plutôt le concept d'*information mutuelle*, encore dénommée *transinformation*. Si nous considérons que $\mathcal{S}_\mathcal{M}$ et $f_\mathcal{M}$ peuvent être décrits par un ensemble de variables (pour l'analyste), alors nous définissons

$$I(\mathcal{S}_{\mathcal{F},\mathcal{M}}; f_\mathcal{M}, \mathcal{S}_\mathcal{M}) = H(\mathcal{S}_{\mathcal{F},\mathcal{M}}) - H(\mathcal{S}_{\mathcal{F},\mathcal{M}} | f_\mathcal{M}, \mathcal{S}_\mathcal{M}),$$

comme la quantité d'information que $f_\mathcal{M}$ et $\mathcal{S}_\mathcal{M}$ révèlent ensembles au sujet de $\mathcal{S}_{\mathcal{F},\mathcal{M}}$. En d'autres termes, la transinformation décrit et quantifie ce qu'apporte à l'analyste le fait de disposer de $\mathcal{D}$. La question est alors de déterminer si $\mathcal{D}$ fournit une information à l'analyste et si oui, quelle est la quantité d'information fournie.

Par construction, il est évident que l'on peut de manière symétrique adopter le même formalisme pour un schéma de contournement. Ainsi :

$$I(\mathcal{S}_{\mathcal{F},\mathcal{M}}; f_\mathcal{M}, \mathcal{S}_\mathcal{M}) = I(\mathcal{S}_{\mathcal{F},\mathcal{M}}; \overline{f_\mathcal{M}}, \mathcal{S}_\mathcal{M}).$$

[7] Il s'agit là d'un abus de notation, destiné à simplifier l'approche. En toute rigueur, la variable X représente un ensemble de variables décrivant s, $\mathcal{S}_\mathcal{M}$ et $f_\mathcal{M}$. Autrement dit, on peut écrire $X = (X_1, X_2, \ldots, X_n)$. Alors, soit par un codage approprié, on réunit toutes ces données en un ensemble de valeurs prises par la variable X (type codage de Gödel ; [38, chapitre 2]), soit on considère la fonction entropie conjointe $H(X_1, X_2, \ldots, X_n) = -\sum_{(x_1,x_2,\ldots,x_n)} p[X_1 = x_1, \ldots, X_n = x_n] \log_2(p[X_1 = x_1, \ldots, X_n = x_n])$.

Rigidité

Une fois que l'analyste est parvenu à extraire l'ensemble $\mathcal{S}_\mathcal{M}$ d'un schéma $\mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M})$ pour produire une variante non détectée, relativement à ce schéma, il devra le contourner. Nous définirons la *rigidité* d'un schéma, que nous noterons $\mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M})$, comme la difficulté plus ou moins grande qu'aura un plagiaire, à le contourner. Cette donnée dépend de manière évidente non seulement de la taille de $\mathcal{S}_\mathcal{M}$ mais également du poids de la fonction de détection $f_\mathcal{M}$:

- plus la taille de $\mathcal{S}_\mathcal{M}$ est importante, plus le nombre de possibilités de modifications offertes au plagiaire augmente ;
- moins la fonction de détection possède d'entrées pour laquelle elle vaut 1 (le poids de la fonction), plus le mode de recherche de $\mathcal{S}_\mathcal{M}$ est facile à leurrer.

Ces constatations amènent donc à définir la rigidité par

$$\mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M}) = \frac{\{X = (X_1, X_2, \ldots, X_s) | f_\mathcal{M}(X) = 0\}}{s2^s} \tag{2.1}$$

$$= \frac{\{X = (X_1, X_2, \ldots, X_s) | \overline{f_\mathcal{M}}(X) = 1\}}{s2^s}. \tag{2.2}$$

Ainsi, nous avons

$$0 \leq \mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M}) \leq 1.$$

Plus $\mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M})$ est faible, plus le schéma de détection est facile à contourner. À l'extrême, si $\mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M}) = 0$, le code n'est pas référencé dans la base.

Il est intéressant de remarquer que $\mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M})$ est lié uniquement au nombre de configurations d'octets qui peuvent être modifiés (comme $X_i \in \{0, 1\}$). Un plagiaire peut changer chacun des octets présents dans ces configurations et le remplacer par une des quelconques 255 autres valeurs que celles effectivement présentes dans le code. Par conséquent, le nombre total de modifications possibles devient extrêmement important. Il croît inversement avec $\mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M})$. Plus précisément, si on note $X = (X_1, X_2, \ldots, X_s) \in \mathbb{F}_2^s$ et si $\mathrm{wt}(X)$ désigne le poids de X (nombre de bits égaux à 1 dans l'écriture binaire de X), alors le nombre total Δ de modifications possibles, que le plagiaire peut effectuer est donné par

$$\Delta = \sum_{X \in \mathbb{F}_2^s} \overline{f_\mathcal{M}}(X).(256^{s - \mathrm{wt}(X)} - 1). \tag{2.3}$$

Nous examinerons certains cas dans la section 2.5.2.

Efficacité

L'efficacité d'un schéma de détection concerne sa capacité à détecter et à identifier de manière univoque un code malveillant donné connu. Il doit également le faire sans erreur, c'est-à-dire ne pas incriminer à tort un programme sain. Cette dernière contrainte est en fait liée à la probabilité de fausse alarme pour le test statistique associé (voir chapitre 3). C'est également lié à des aspects phylogénétiques des codes malveillants tels que définis dans [60, 71].

Des études [74] ont montré qu'en règle générale, déterminer la valeur de cette probabilité de fausse alarme est plus difficile qu'il n'y paraît. Typiquement, le paramètre $s = |\mathcal{S}_{\mathcal{M}}|$ a une valeur comprise entre 12 et 36 [74]. La probabilité pour une séquence de s octets d'apparaître au hasard est de $\frac{1}{256^s}$. Donc, dès que s est suffisamment grand, cette probabilité théorique tend vers 0 et, en théorie toujours, aucun fichier ne peut être détecté par erreur comme infecté. Malheureusement, la réalité ne colle pas à la théorie. Comme souligné dans [74], la probabilité de trouver une séquence aléatoire de $s = 24$ octets dans un corpus de 500 Mo est de 0,34 alors que la théorie donne une probabilité de $0,85 \times 10^{-49}$.

Cet écart vient du fait que les octets dans un fichier ne sont pas des variables aléatoires indépendantes, identiquement distribuées (selon une probabilité $p = \frac{1}{256}$). À titre d'exemple, nous avons $P[b_0 =' M'] = 1$ et $P[b_1 =' Z' | b_0 =' M'] = 1$ dans un exécutable Win32. Il existe de fortes dépendances entre les octets dans un fichier. Un meilleur modèle consisterait à utiliser des processus markoviens[8], pour décrire les dépendances fortes des octets entre eux.

Ainsi qu'il est mentionné dans [74], « *l'expérience, qu'elle soit humaine ou algorithmique, est un ingrédient essentiel dans le choix d'une bonne signature virale.* » La plupart des antivirus du commerce utilisent des paramètres de taille s relativement efficaces.

Remarque. L'efficacité d'un schéma de détection dépend principalement de la taille s. C'est la raison pour laquelle, certains antivirus sont plus sujets que d'autres à provoquer des fausses alarmes, du fait qu'ils utilisent des valeurs de s trop petites. Mais la fonction de détection peut avoir, dans le cas de fonctions de détection non triviales, un effet positif sur le taux de fausses alarmes. L'exploration et l'étude des fonctions de détection relativement à cet aspect est un problème ouvert.

Propriétés des fonctions de détection

La fonction de détection joue un rôle important dans un schéma de détection. Elle constitue le mode de recherche du motif proprement dit. Une propriété comme la rigidité montre que le poids de cette fonction (le nombre de ses entrées pour lesquelles la fonction vaut 1) est un paramètre essentiel. Alors qu'un poids d'une unité indique une seule possibilité de réalisation, un poids plus important augmente d'autant les possibilités (ou les configurations) de détection. Ce poids détermine également celui de la fonction de non détection, en vertu de l'égalité bien connue, pour une fonction $f : \mathbb{F}_2^n \to \mathbb{F}_2$:

$$\mathrm{wt}(f) = 2^n - \mathrm{wt}(\overline{f}).$$

En d'autres termes, si la fonction de détection possède un poids faible (nombre de possibilités de détection limité), la fonction de non détection aura un poids

[8] Un *processus markovien* est un processus dans lequel, à chaque instant, la probabilité d'un état quelconque du système dans le futur dépend seulement de l'état du système à l'instant actuel (t_0) sans dépendre de la manière dont le système a été amené dans cet état.

élevé (nombre de possibilités de contournement élevé). Cela permet de donner la définition suivante.

Définition 2.3 *Soient un schéma de détection* $\mathcal{SD} = \{\mathcal{S}_\mathcal{M}, f_\mathcal{M}\}$ *et le schéma de contournement* $\mathcal{SC} = \{\mathcal{S}_\mathcal{M}, \overline{f_\mathcal{M}}\}$ *et soit* $s = |\mathcal{S}_\mathcal{M}|$. *Le schéma* $\mathcal{SD}$ *est dit* plus fort *que le schéma* $\mathcal{SC}$ *si et seulement si*

$$2^{s-1} \leq wt(f_\mathcal{M}) \leq 2^n - 1$$

Cette définition permet détablir un premier critère pour une « bonne » fonction de détection.

Une autre propriété intéressante concerne l'importance relative des variables X_i en entrée de la fonction de détection. Il est important qu'elles aient toutes la même importance sur la valeur de la fonction $f_\mathcal{M}$ (ou de manière équivalente, sur celles de $\overline{f_\mathcal{M}}$). Dans le cas contraire, si une variable était prépondérante (respectivement moins prépondérante) sur les autres, le plagiaire ne manquerait pas de tirer parti de cette propriété pour optimiser ses chances de contournement de la détection : il modifierait prioritairement (respectivement en dernier) cette variable. Cela se généralise à un ensemble quelconque de t variables. Nous adopterons la définition suivante.

Définition 2.4 *Une fonction de détection sera dite* faiblement contournable à l'ordre t *si et seulement si la sortie de la fonction de détection ne dépend statistiquement d'aucun sous-ensemble de taille au plus* t *de variables d'entrées. Une fonction de détection sera dite* fortement contournable à l'ordre t *si et seulement si la fonction de détection dépend statistiquement et identiquement de tout sous-ensemble de taille au plus* t *de variables d'entrées.*

En d'autres termes, aucun sous-ensemble d'au plus t variables d'entrées ne sera plus intéressant à considérer qu'un autre pour modifier le code, en vue de produire une variante non détectée. La différence entre « faiblement » et « fortement » tient au fait que, dans le premier cas, il n'existe aucune dépendance entre un quelconque sous-ensemble de variables de taille au plus t, alors que dans le second cas, il existe une dépendance, mais elle est la même pour tous ces sous-ensembles. Il est assez intuitif de supposer que le premier cas est plus difficilement réalisable que le second. Nous le démontrerons dans la section 2.6.

Notons au passage que si $f_\mathcal{M}$ est faiblement contournable à l'ordre t (respectivement fortement contournable à l'ordre t), il en est de même de $\overline{f_\mathcal{M}}$.

Cette propriété amène tout naturellement à considérer une classe particulière de fonctions booléennes, très importantes en cryptologie à clef secrète : les fonctions immunes aux corrélations à l'ordre t. Afin d'utiliser cette propriété, nous allons d'abord définir l'outil mathématique de base pour l'étude des fonctions booléennes. Le lecteur pourra consulter [11, chapitre 2], [92, pp. 207] et [35] pour plus de détails sur cet outil.

Définition 2.5 *Soit une fonction booléenne sur* $\mathbb{F}_2^n$. *La* transformée de Walsh-Hadamard *de f est la transformée de Fourier de la fonction signe correspondante, $x \mapsto (-1)^{f(x)}$:*

$$\forall u \in \mathbb{F}_2^n,\ \widehat{\chi_f}(u) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x)} (-1)^{<u,x>}$$

où $< ., . >$ désigne le produit scalaire usuel.

La transformée de Walsh permet de caractériser les dépendances statistiques existant entre des sous-ensembles de variables en entrée et la sortie de la fonction. Précisons cela.

Définition 2.6 *Une fonction booléenne à n variables est dite* sans corrélation à l'ordre t, *ou immune aux corrélations à l'ordre t, si sa distribution de valeurs ne change pas lorsque l'on fixe au plus t entrées.*

Autrement dit, la sortie de la fonction est statistiquement indépendante de tout vecteur $(X_{i_1}, X_{i_2}, \ldots, X_{i_t})$. Par exemple, la fonction $f(X_1, X_2, \ldots, X_n) = X_1 \oplus X_2 \oplus \ldots X_n$ est sans corrélation à l'ordre $(n-1)$.

En 1988 [145], un résultat important a permis de caractériser cette propriété à l'aide de la transformée de Walsh.

Proposition 2.1 *[145] La fonction booléenne f à n variables est sans corrélation à l'ordre t si et seulement si elle vérifie*

$$\widehat{\chi_f(u)} = 0 \quad \forall u \in \mathbb{F}_2^n,\ 1 \leq wt^9(u) \leq t.$$

Nous pouvons maintenant établir la proposition suivante.

Proposition 2.2 *Une fonction $f_{\mathcal{M}}$ est faiblement contournable à l'ordre t si et seulement si elle est sans corrélation à l'ordre t. Une fonction $f_{\mathcal{M}}$ est fortement contournable à l'ordre t si et seulement si*

$$\forall u \in \mathbb{F}_2^n \text{ tel que } 1 \leq wt(u) \leq t \qquad \widehat{\chi_f(u)} \text{ est une constante.}$$

Preuve.
Évidente par définition de l'immunité aux corrélations. Notons que cette propriété est également valable pour la fonction de non détection $\overline{f_{\mathcal{M}}}$. En effet, nous avons

$$\widehat{\chi_f(u)} = -\widehat{\chi_{\overline{f}}(u)}.$$

■

Nous verrons dans la section 2.6.1 un exemple de fonction pour chacune de ces deux propriétés.

[9] $wt(u)$ désigne le poids de u, c'est-à-dire le nombre de bits valant 1 dans son écriture binaire.

2.4 Le problème de l'extraction

2.4.1 L'extraction de schémas de détection

Le problème de l'extraction de motifs de détection a été étudié par Christodorescu et Jha [24]. L'algorithme qu'ils ont proposé ne considère qu'un cas trivial de fonctions de détection et, par conséquent, le problème général de l'extraction de schéma n'est pas résolu. Nous allons étudier deux algorithmes d'extraction, le second permettant de résoudre de manière générale ce problème, et ce quelle que soit la fonction de détection utilisée.

Soit un détecteur $\mathcal{D}$ donné, l'objectif est de retrouver le schéma de détection complet, pour un code malveillant $\mathcal{M}$, relativement à $\mathcal{D}$. Cette reconstruction est réalisée par une analyse en boîte noire. Aucune technique de rétro-ingénierie n'est utilisée. Avec les notations précédentes, il s'agit donc de retrouver la fonction de non détection $\overline{f_{\mathcal{M}}}$ et les indices des octets impliqués dans le motif lui-même. Comme le fichier examiné $\mathcal{F}$ est le code malveillant lui-même, nous utiliserons la notation $\mathcal{S}_{\mathcal{M},\mathcal{M}}$ au lieu de $\mathcal{S}_{\mathcal{F},\mathcal{M}}$.

Nous considérerons la fonction de non détection plutôt que la fonction de détection pour les raisons suivantes :

- le point de vue de l'attaquant est plus intéressant pour évaluer la résistance d'un antivirus à l'analyse en boîte noire ;
- extraire la fonction de détection aurait nécessité de calculer ensuite sa négation pour obtenir la fonction de non détection. Malheureusement, le calcul de la négation d'une forme disjonctive normale a une complexité, en pire cas, exponentielle[10]. En conséquence, l'algorithme E-2 extraira directement $\overline{f_{\mathcal{M}}}$.

2.4.2 Approche naïve : algorithme E-1

Un premier algorithme a été conçu pour traiter l'instance la plus fréquente du problème de l'extraction. Si on le compare à l'algorithme proposé par Christodorescu et Jha [24], il peut sembler, à première vue, moins efficace et plutôt naïf. Toutefois, l'expérience pratique montre que ce n'est pas le cas. Cet algorithme « naïf », bien au contraire, est parvenu systématiquement à résoudre le problème de l'extraction, pour l'instance concernée, là où celui de Christodorescu et Jha avait échoué pour certains cas dans lesquels le motif de détection est très dispersé dans le code.

Considérons le code $\mathcal{M}$ de taille n. Le pseudo-code de ce premier algorithme est donné en figure 2.1. L'algorithme modifie successivement chaque octet de $\mathcal{M}$ (remplacé par un octet nul) et soumet le code ainsi modifié au détecteur $\mathcal{D}$. Si ce dernier détecte toujours $\mathcal{M}$ (c'est-à-dire $\mathcal{D}(\mathcal{M}) = \sigma$), cela signifie que l'octet modifié n'est pas impliqué dans le schéma de détection. Dans le cas contraire, l'indice de l'octet appartient à $\mathcal{S}_{\mathcal{M},\mathcal{M}}$.

[10] Ce calcul revient à transformer une forme disjonctive normale en sa forme conjonctive normale. Cette opération a une complexité exponentielle [99, Théorème 4.1].

Entrée : un code malveillant $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$, un détecteur $\mathcal{D}$ et une identification σ pour $\mathcal{M}$.
Sortie : le motif $\mathcal{S}_{\mathcal{M},\mathcal{M}}$ (indices).

$\mathcal{S}_{\mathcal{M},\mathcal{M}} \leftarrow \{\}$
Pour i = 1 à n faire
    modifier seulement l'octet m_i in $\mathcal{M}$
    Si $\mathcal{D}(\mathcal{M}) \neq \sigma$ alors
        $\mathcal{S}_{\mathcal{M},\mathcal{M}} \leftarrow \mathcal{S}_{\mathcal{M},\mathcal{M}} \cup \{i\}$
    Fin si
Fin pour
retourner $\mathcal{S}_{\mathcal{M},\mathcal{M}}$

Table 2.1 – Algorithme d'extraction de schéma de détection E-1

La complexité de l'algorithme 2.1 est linéaire en la taille de $\mathcal{M}$ soit $\mathcal{O}(n)$. Le principal intérêt de cet algorithme est sa capacité à extraire le motif $\mathcal{S}_{\mathcal{M},\mathcal{M}}$, quelle que soit sa structure (octets dispersés, faible nombre d'octets...). En revanche, son efficacité est limitée à une unique fonction de détection – certes la plus fréquemment utilisée – dont la DNF est donnée par :

$$f_{\mathcal{M}}(X_1, X_2, X_3, \ldots, X_s) = X_1 \wedge X_2 \wedge X_3 \wedge \ldots \wedge X_s.$$

Cette fonction de détection correspond à la technique de détection la plus fréquemment utilisée : recherche de « signatures » simples (technique basique). C'est la fonction ET. Contrairement à ce que prétendent la plupart des éditeurs d'antivirus, les résultats prouvent que cette technique est encore très largement utilisée. La section 10 présente des résultats détaillés pour certaines variantes de la famille *W32/Bagle*.

Remarque. La fonction de non détection correspondant à la fonction ET est très simple à calculer, en utilisant les règles élémentaires du calcul booléen. Nous avons :

$$\overline{f_{\mathcal{M}}}(X_1, X_2, X_3, \ldots, X_s) = X_1 \vee X_2 \vee X_3 \vee \ldots \vee X_s.$$

C'est la fonction OU. Elle exprime qu'en modifiant un quelconque des octets situés aux indices de $\mathcal{S}_{\mathcal{M},\mathcal{M}}$, cela suffit à rendre le code indétectable.

2.4.3 Approche par apprentissage de DNF : algorithme E-2

L'extraction de certains schémas de détection n'est pas réalisable avec l'algorithme E-1. Ce sont les schémas pour lesquels, la fonction de détection est

différente de la fonction ET. Nous allons donc considérer un algorithme général permettant de résoudre le problème de l'extraction d'un schéma de détection (ou de manière équivalente d'un schéma de contournement). Cet algorithme utilise des techniques d'apprentissage de formules booléennes. Rappelons tout d'abord quelques concepts de la théorie de l'apprentissage. Le lecteur intéressé pourra consulter [73] pour une présentation détaillée de cette théorie.

Apprentissage de concepts booléens

Nous nous intéresserons à l'apprentissage de *concepts booléens* dans lequel celui qui « apprend » – en d'autres termes un analyste en boîte noire dans notre contexte – a pour objectif d'inférer comment une fonction cible inconnue – la fonction de non détection dans notre cas – classe les éléments d'un espace donné selon la valeur (0 ou 1) que prend cette fonction pour un nombre donné d'éléments de cet espace.

Le *domaine d'instances* $\mathcal{X}$ est l'ensemble de tous les éléments (ou *instances*) possibles qu'il faut classer. Dans le contexte de la détection virale, nous considérerons l'hypercube booléen $\{0,1\}^n$ comme domaine de référence. Cela correspond aux différentes entrées possibles pour la fonction de non détection, avec les notations de la section 2.3.1. Un *concept booléen* ou *concept* est une fonction booléenne définie sur un domaine $\mathcal{X}$. Une *classe de concept* $\mathcal{C}$ est une famille de sous-ensembles de $\mathcal{X}$. En d'autres termes, $\mathcal{X} \subseteq \mathcal{P}(\mathcal{X})$. Dans notre contexte, $\mathcal{C}$ décrit l'ensemble de toutes les formes normales disjonctives de fonctions de non détection[11].

Maintenant, tout $x \in \mathcal{X}$ est classé selon son appartenance à un *concept cible* $f \in \mathcal{C}$ – la fonction de non détection $f_{\mathcal{M}}$. Un élément $x \in \mathcal{X}$ est un exemple *positif* de f si $f(x) = 1$ et un exemple *négatif* dans le cas contraire. Cette méthode d'apprentissage est dénommée *modèle d'apprentissage par requêtes* (*Query Learning Model*).

Algorithme général d'extraction

Dans notre contexte, la formule booléenne à apprendre[12] est la fonction de non détection $f_{\mathcal{M}}$ sous sa forme disjonctive normale. Chaque variable X_i désigne un octet du code malveillant $\mathcal{M}$ avec $i = 1, 2, \ldots, n$ ($n = |\mathcal{M}|$). La forme disjonctive normale est alors la réunion de conjonctions de variables booléennes pouvant être sous la forme « vraie » (notée X_i) ou « fausse » (notée $\overline{X_i}$; voir l'annexe 10.3 pour la signification de cette notation).

Un point important doit être précisé. Le problème de l'apprentissage de la formule DNF est depuis longtemps un problème ouvert même si des techniques efficaces d'apprentissage ont été imaginées pour certaines classes particulières de formules DNF. Toutefois, la complexité en mémoire et en temps

[11] Rappelons qu'une fonction peut avoir plusieurs formes disjonctives normales équivalentes. Ainsi, les DNFs $x_1 \vee x_2$ et $x_1x_3 \vee x_1\overline{x_3} \vee x_2$ décrivent la même fonction. La première est la forme simplifiée dite minimale.

[12] Le terme apprendre signifie que l'on cherche à trouver les termes de cette formule.

Entrée : un code malveillant $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$,
un détecteur $\mathcal{D}$ et une identification σ pour $\mathcal{M}$.
Sortie : le motif $\mathcal{S}_{\mathcal{M},\mathcal{M}}$ (indices) et
la fonction de non détection $\overline{f_{\mathcal{M}}}$ (DNF).

$\mathcal{S}_{\mathcal{M},\mathcal{M}} \leftarrow \{\}$; $\text{DNF}_{f_{\mathcal{M}}} \leftarrow \{\}$
$S \leftarrow 2^{\lfloor \log_2(n) \rfloor + 1}$; $S' \leftarrow 2^{\lfloor \log_2(n) \rfloor + 1}$
Tant que $S > 0$ faire
  $S \leftarrow \frac{S}{2}$; $q \leftarrow \frac{S'}{S}$
  Pour i = 0 à q -1 faire
    binf $\leftarrow i \times S$; bsup $\leftarrow$ binf $+ S$
    modifier les octets dans I = [binf, bsup[in $\mathcal{M}$
    Si $\mathcal{D}(\mathcal{M}) \neq \sigma$ alors
      $\mathcal{S}_{\mathcal{M},\mathcal{M}} \leftarrow \mathcal{S}_{\mathcal{M},\mathcal{M}} + I$
      $X = 1_I(X_1, X_2, \ldots, X_n)$
      $\text{DNF}_{f_{\mathcal{M}}} \leftarrow \text{DNF}_{f_{\mathcal{M}}} \cup X$
    Fin si
  Fin pour
Fin tant que
$\mathcal{S}_{\mathcal{M},\mathcal{M}} \leftarrow$ CombinatorialMinimize($\mathcal{S}_{\mathcal{M},\mathcal{M}}$)
$\text{DNF}_{f_{\mathcal{M}}} \leftarrow$ LogicalMinimize($\text{DNF}_{f_{\mathcal{M}}}, \mathcal{S}_{\mathcal{M},\mathcal{M}}$)

Table 2.2 – Algorithme d'extraction de schéma de détection E-2

de tout algorithme d'apprentissage dépend de manière évidente de celle du concept cible sous-jacent. Dans le cas de formules DNF, elle dépend du nombre de termes compris entre 0 (la fonction constante nulle) et 2^n (la fonction constante $f(x) = 1$). La fonction de détection ET que nous avons considérée dans la section 2.4.2 contient un seul terme. Cela explique pourquoi l'algorithme E-1 est bien mieux adapté que l'algorithme E-2 (dont la complexité n'est plus linéaire) pour ce cas très particulier, mais néanmoins le plus répandu, de fonctions. L'algorithme général d'extraction E-2 est présenté en figure 2.2. Par souci de clarté, nous donnons ici une version non récursive. La notation $X = 1_I(X_1, X_2, \ldots, X_n)$ la construction d'une terme logique de DNF, en utilisant la fonction caractéristique relativement à un intervalle I d'indices. Chaque variable X_i ou sa négation apparaît dans le terme en vertu de la règle suivante :

$$X_i = \begin{cases} \overline{X_i} & \text{si } i \in I \\ X_i & \text{si } i \notin I \end{cases}$$

L'algorithme E-2 se compose de trois parties :

1. La première phase est la phase initiale d'apprentissage. Elle consiste à

modifier des portions d'octets contigus du code $\mathcal{M}$. Cette modification se fait par approche dichotomique, en $\log_2(n)$ pas. Les portions sont de taille décroissante, égale à une puissance de deux. Comme dans l'algorithme de Christodorescu et Jha [24], le but premier est d'identifier les octets impliqués dans le motif de détection $\mathcal{S}_{\mathcal{M},\mathcal{M}}$. Cette phase permet de trouver quelques termes de la DNF de la fonction de non détection. Cette phase a une complexité de $\mathcal{O}(2n)$ (boucle WHILE).

2. Une phase de simplication combinatoire, appelée COMBINATORIALMINIMIZE, va éliminer les redondances entre les termes initiaux de la DNF. L'ensemble $\mathcal{S}_{\mathcal{M},\mathcal{M}}$ produit par la phase précédente contient en général quelques termes dont les intervalles supports (voir notation précédente) correspondants sont inclus dans d'autres intervalles. À titre d'exemple, le terme $X_1X_2X_3X_4$ et le terme X_1X_2 ont pour intervalle support respectif $[1,4]$ et $[1,2]$. Le second est inclu dans le premier. Cela signifie que les variables X_3 et X_4 ne jouent aucun rôle dans le motif de détection. Seul le terme X_1X_2 est conservé. Cette étape va donc simplifier, d'un point de vue combinatoire, l'ensemble $\mathcal{S}_{\mathcal{M},\mathcal{M}}$. Il s'agit de rechercher les éléments minimaux dans un ensemble partiellement ordonné par la relation d'inclusion entre intervalles. Cette étape possède une complexité en pire cas en $\mathcal{O}(t\log(t)+tn)$ (tri puis comparaison de termes consécutifs) avec $t = |\mathcal{S}_{\mathcal{M},\mathcal{M}}|$ en entrée de la procédure COMBINATORIALMINIMIZE. L'ensemble produit est de taille s.
3. Une phase LOGICALMINIMIZE dont le rôle est :
   - de finaliser l'apprentissage par une recherche exhaustive sur tous les s-uplets de variables. Avec les notations précédentes, à chaque s-uplets de $\mathbb{F}_2^s$ correspond une configuration de modifications des octets de $\mathcal{S}_{\mathcal{M},\mathcal{M}}$. Le code $\mathcal{M}$ est alors modifié selon chacune de ces configurations, puis testé vis-à-vis du détecteur $\mathcal{D}$. Si le code, pour une configuration donnée, n'est plus détecté par $\mathcal{D}$, le terme logique correspondant est ajouté à la formule DNF ;
   - sur la DNF finale, une étape de minimisation logique est effectuée. En effet, comme à l'issue de la phase finale d'apprentissage, la DNF contient des termes redondants, il est nécessaire d'éliminer ces redondances d'un point de vue logique en utilisant les règles du calcul booléen [31, chapitre 9] et [94]. L'algorithme utilise la méthode de Quine-McCluskey [89,107,108]. Le problème de la minimisation logique est un problème NP-dur [94, chapitre 5.8.3]. Sa complexité est en $\mathcal{O}(s2^s)$ [141].

   La complexité de la procédure LOGICALMINIMIZE est finalement en $\mathcal{O}(2^s + s2^s)$ si $s = |\mathcal{S}_{\mathcal{M},\mathcal{M}}|$ en entrée de cette procédure.

Nous avons donc le résultat suivant.

Théorème 2.1 *L'algorithme d'extraction d'un schéma de contournement d'un code $\mathcal{M}$ de taille n dont le motif de détection est de taille s, donné en figure 2.2 a une complexité en $\mathcal{O}(sn + s2^s)$.*

Preuve.
Avec la discussion et les notations précédentes, la complexité générale en pire cas est en $\mathcal{O}(2n + t\log(t) + tn + s^2 2^s)$. Les termes $2n$ et t sont négligeables par rapport à 2^s et nous avons également $t \cong s$ (ce qui est confirmé par l'expérience). D'où le résultat. ■

Il est intéressant de noter que la complexité mémoire est également un paramètre à considérer. La boucle WHILE produit un ensemble $\mathrm{DNF}_{f_{\mathcal{M}}}$ dont la taille est bornée par le nombre maximal possible de termes, soit $2n$. La procédure LOGICALMINIMIZE produit, elle, un ensemble $\mathrm{DNF}_{f_{\mathcal{M}}}$ d'une taille qui peut être plus importante selon la fonction de non détection $f_{\mathcal{M}}$. Les résultats expérimentaux obtenus sur la famille *I-Worm/Bagle* n'ont pas montré d'augmentation significative de cette taille.

Remarques

1. L'algorithme E-1 est un cas particulier de l'algorithme E-2, quand la fonction de détection est la fonction logique ET. Ce cas étant le plus fréquent, comme l'ont montré les nombreux tests effectués, l'utilisation de l'algorithme E-1 est un meilleur choix préalable, dans la mesure où cela permet de faire l'économie des étapes de minimisations combinatoire et logique.
2. L'apprentissage d'une formule DNF monotone (cas de la fonction de détection) possède un complexité bien meilleure lorsque l'on utilise l'algorithme d'Angluin [5]. En effet, en utilisant la méthode *membership and equivalence queries*, cet algorithme apprend une formule monotone, sur le domaine $\{0,1\}^n$, composée de m termes avec seulement $\mathcal{O}(nm)$ requêtes. Mais pour obtenir la fonction de non détection correspondante, la négation de la formule obtenue doit être calculée. Ce calcul possède une complexité en pire cas, exponentielle.
3. L'algorithme E-2 trouve la DNF exacte de la fonction de non detection alors que généralement les méthodes d'apprentissage peuvent ne fournir qu'une DNF non simplifiée.

2.4.4 Exemples de fonctions de détection

À toute technique de détection par analyse de forme, correspond une fonction booléenne de détection (ou de manière équivalente une fonction de non détection). Pour illustrer cela, et afin de montrer qu'il existe bien d'autres fonctions de détection que celle triviale (la fonction ET), nous allons donner quelques exemples simples.

Détection par caractères génériques *(Wildcards)*

Cette forme de détection considère généralement des motifs dans lesquels les indices d'octets peuvent en partie être variables. Considérons le motif de détection suivant emprunté à [130, section 11.1.2].

```
..... 07BB ??02 %3 33C9 ......
```

Le symbole `??` indique de négliger l'octet placé à cet endroit tandis que la chaîne `%3 33` demande de rechercher le caractère `33` dans l'un quelconque des trois octets suivants. Ce mode de recherche correspond en fait à la fonction de détection décrite par la DNF suivante (nous ne donnons que l'extrait pertinent de la DNF) :

$$\begin{aligned}
\ldots \quad & (X_i = 07) \wedge (X_{i+1} = BB) \wedge (X_{i+3} = 02) \wedge (X_{i+4} == 33) \wedge (X_{i+5} = C9) \\
\vee \quad & (X_i = 07) \wedge (X_{i+1} = BB) \wedge (X_{i+3} = 02) \wedge (X_{i+5} == 33) \wedge (X_{i+6} = C9) \\
\vee \quad & (X_i = 07) \wedge (X_{i+1} = BB) \wedge (X_{i+3} = 02) \wedge (X_{i+6} == 33) \wedge (X_{i+7} = C9) \\
\ldots &
\end{aligned}$$

Techniques de non-coïncidences partielles

Cette technique a été développée par IBM dans le cadre de ses recherches sur les scanners antiviraux. La technique dite de non-coïncidences partielles (ou technique *mismatch*) autorise, dans un motif de détection de taille s, que μ octets prennent une valeur quelconque. Par exemple, considérons le motif de détection `01 02 03 04` avec une valeur de non-coïncidences partielles de $\mu = 2$ (cet exemple a été pris dans [130, section 11.1.3]). Alors la fonction de détection correspondante (extrait de la DNF) est donnée par :

$$\begin{aligned}
\ldots \quad & (X_i = 01) \wedge (X_{i+1} = 02) \vee (X_i = 01) \wedge (X_{i+2} = 03) \\
\vee \quad & (X_{i+1} = 02) \wedge (X_{i+2} = 03) \vee (X_i = 01) \wedge (X_{i+3} = 04) \\
\vee \quad & (X_{i+1} = 02) \wedge (X_{i+3} = 04) \vee (X_{i+2} = 03) \wedge (X_{i+3} = 04)
\end{aligned}$$

D'une manière générale, pour un motif de taille s et de valeur de non-coïncidences partielles μ, la formule DNF contient $\binom{s}{\mu}$ termes, chacun d'entre eux ayant $s - \mu$ variables ; il s'agit d'une $(s - \mu)$-DNF monotone.

Identification quasi-exacte

Cette méthode est utilisée pour augmenter la qualité de la détection. Plutôt que de considérer un seul schéma de détection, on en considère deux (voire éventuellement plus) [130, section 11.2.3]. Il est alors facile de démontrer que si on utilise les schémas $(\mathcal{S}^1_{\mathcal{M},\mathcal{M}}, f^1_{\mathcal{M}})$ et $(\mathcal{S}^2_{\mathcal{M},\mathcal{M}}, f^2_{\mathcal{M}})$, il est possible de le décrire par un schéma unique $(\mathcal{S}_{\mathcal{M},\mathcal{M}}, f_{\mathcal{M}})$ tel que $\mathcal{S}_{\mathcal{M},\mathcal{M}} = \mathcal{S}^1_{\mathcal{M}} \cup \mathcal{S}^2_{\mathcal{M},\mathcal{M}}$ et $f_{\mathcal{M}} = f^1_{\mathcal{M}} \vee f^2_{\mathcal{M}}$ ou $f_{\mathcal{M}} = f^1_{\mathcal{M}} \wedge f^2_{\mathcal{M}}$ selon les cas (à une minimisation logique près car l'union logique ou l'intersection logique peuvent produire des DNF non minimales).

Notons que ce cas couvre également celui où plusieurs moteurs d'analyse de forme sont utilisés. Plusieurs antivirus combinent un moteur heuristique à un moteur classique.

Autres techniques

Les autres techniques par analyse de forme comme la technique *des signets (bookmarks techniques)*, le scan de motifs utiles *(smart scanning)*, la technique du squelette *(skeleton detection)*, la méthode du décrypteur statique *(static decryptor techniques)*, les heuristiques reposant sur la forme..., parmi de nombreuses autres (voir [130, chapitre 11]) peuvent être modélisées par des schémas de détection similaires aux précédents. Mais pour ces techniques, en général plus élaborées, quelques remarques doivent êtres faites :

- la taille s du motif de détection est plus importante. Le motif s'étend plus largement sur tout le code ;
- la fonction de détection est en général plus complexe (le nombre de termes est proche de 2^s). Cependant, il n'est pas sûr que la fonction de non détection soit tout aussi complexe. C'est là également un problème ouvert ;
- des considérations de nature combinatoire sur la structure de la DNF de la fonction de détection peuvent également intervenir, en plus du poids de cette dernière.

En ce qui concerne les techniques de contrôle de parité *(checksum)* ou de hachage partiel (code de redondance cyclique, fonction de hachage...voir [130, chapitre 11]), notre approche reste la même, mais plutôt que de considérer les octets du code, il est préférable de voir ce dernier sous une forme binaire. Alors, toute valeur de parité *(checksum)* ou tout haché partiel peut être décrit par un ensemble de fonctions booléennes [36]. La fonction de détection résultante est alors la conjonction logique de ces fonctions booléennes (à une étape de minimisation logique près). L'exemple (trivial) suivant illustre notre propos.

Exemple 2.1 *Soient deux octets, vus comme des suites binaires :*

$$O_i = (b_7^i, b_6^i, \ldots, b_0^i) \text{ et } O_j = (b_7^j, b_6^j, \ldots, b_0^j).$$

Considérons le contrôle de parité sur deux bits (p_1, p_0)*, définis comme suit :*

$$p_1 = \bigoplus_{k=0}^{7} b_k^i \text{ et } p_0 = \bigoplus_{k=0}^{7} b_k^j.$$

Un élément de détection (conjointement avec d'autres) sera que (p_1, p_0) *diffèrent de valeurs attendues (la fonction de détection correspondante vaut alors 0). La DNF de la fonction de détection sera alors définie par*

$$f = p_1 \wedge p_0.$$

Le lecteur calculera à titre d'exercice la DNF complète en fonction des b_k^i *et des* b_k^j.

Enfin, des techniques plus élaborées et qui intègrent des aspects quantitatifs à côté de caractéristiques essentiellement qualitatives ou structurelles, comme

l'analyse spectrale par exemple [38, chapitre 5], peuvent également être décrites par ce modèle (voir les exercices en fin de chapitre). Cependant, comme pour les techniques de contrôle de parité, la complexité du schéma est telle qu'il est impossible de représenter la fonction de détection, du fait de sa complexité mémoire. Nous verrons dans le chapitre 3 que la modélisation statistique des techniques antivirales permet d'appréhender de manière beaucoup plus puissante ces techniques très élaborées de détection.

2.5 Analyse des logiciels antivirus

Les algorithmes E-1 et E-2 ont permis de systématiquement extraire les schémas de détection et de contournement pour les antivirus que nous avons testés (voir annexe 10). Les résultats ont non seulement permis d'évaluer sur une base technique rigoureuse et reproductible les qualités respectives des différents produits du commerce testés, mais également de faire des observations surprenantes, notamment sur la variété réelle de l'offre antivirale. D'une manière générale, la qualité globale des schémas de détection tend à être faible voire très faible dans certains cas.

2.5.1 Transinformation

Quel que soit le produit testé, l'extraction du schéma de détection ou de contournement s'est avérée facile. En d'autres termes, dans la mesure où un antivirus révèle systématiquement tous les paramètres du schéma (fonctions $f_{\mathcal{M}}$ ou $\overline{f_{\mathcal{M}}}$, taille s, ensemble $\mathcal{S}_{\mathcal{M},\mathcal{M}}$), la transinformation est maximale. Nous pouvons alors écrire

$$I(\mathcal{S}_{\mathcal{M},\mathcal{M}}; \overline{f_{\mathcal{M}}}, \mathcal{S}_{\mathcal{M}}) = H(\mathcal{S}_{\mathcal{M},\mathcal{M}}).$$

Cela signifie que le plagiaire lève toute incertitude dès lors qu'il dispose du produit. Ce dernier ne cherche pas à contrarier l'analyse en boîte noire. De ce point de vue, tous les produits testés se sont révélés très faibles.

2.5.2 Rigidité du schéma

La rigidité quantifie la difficulté à effectivement contourner un schéma de détection donné, lorsque ce dernier a été extrait. Cette propriété est liée au nombre de modifications qu'il est possible d'effectuer afin de contourner la détection par un détecteur $\mathcal{D}$. Les équations (2.1) et (2.2) de la section 2.3.2 impliquent à la fois le paramètre s et la fonction $f_{\mathcal{M}}$ (ou de manière équivalente la fonction $\overline{f_{\mathcal{M}}}$).

L'évaluation précise de la rigidité requiert d'évaluer précisément l'ensemble $\{X = (X_1, X_2, \ldots, X_s) | \overline{f_{\mathcal{M}}}(X) = 1\}$. La formule (2.3) de la section 2.3.2 permet ensuite de calculer le nombre total de modifications d'octets, susceptibles de produire une absence de détection.

Pour résumer :

- pour la fonction de détection ET et un motif de taille s, nous avons

$$\Delta = 256^s - 1.$$

  Il apparaît que les signatures courtes sont préférables aux signatures plus longues. Ces dernières autorisent en effet plus de modifications pouvant finalement résulter en une absence de détection ;
- pour les autres fonctions de non détection $\overline{f_{\mathcal{M}}}$, l'évaluation de Δ a une complexité en $\mathcal{O}(2^s)$. Dans ce contexte, la tâche du plagiaire sera rendue plus difficile quand s croît. Les motifs de détection longs sont préférables aux motifs courts.
  L'analyse en profondeur des antivirus testés a révélé que les tailles de motifs sont plutôt faibles (de l'ordre de 15 octets en moyenne). Les efforts du plagiaire ne sont donc nullement contrariés en pratique, malgré la complexité de l'algorithme E-2. Le paramètre s devrait être plus important ($s > 45$).

Enfin, notons que le poids de la fonction de non détection a un impact non négligeable sur la rigidité du schéma et sur le paramètre Δ. Comme $\mathrm{wt}(\overline{f_{\mathcal{M}}}) + \mathrm{wt}(f_{\mathcal{M}}) = 2^s$, les fonctions de détection les plus adaptées pour contrer l'action d'un plagiaire sont celles ayant un poids élevé. Les résultats expérimentaux, pour les produits testés, montrent une faiblesse générale pour cette propriété.

2.5.3 Efficacité

L'analyse en boîte noire des produits testés a montré que ces derniers utilisaient en général des motifs plutôt courts. Nous avons aussi observé qu'il existait une opposition entre la notion de rigidité et celle d'efficacité dans la plupart des cas. Pour la fonction ET – la plus courante –, la probabilité de fausse alarme (et donc le nombre de faux positifs) augmente avec la rigidité. Un compromis est alors à établir, lequel n'est pas aisé. L'usage d'autres fonctions de détection doit permettre de faire abstraction d'un tel compromis.

Il est intéressant de noter que les motifs de détection utilisés par la plupart des produits (octets impliqués, fonction de détection) présentent des ressemblances particulièrement frappantes d'un produit à un autre (voir annexe 10). Cela tend à démontrer que la lutte antivirale est un effort commun et mutualisé entre les principaux éditeurs. L'offre n'est donc pas si variée qu'il n'y paraît, du moins pour les techniques d'analyse de forme. Les propriétés structurelles du format PE, par exemple, ne peuvent à elles seules justifier de telles similarités.

2.6 Schéma de détection sécurisé

L'étude précédente montre qu'aucun antivirus testé n'offre de résistance en matière d'extraction de schéma de détection ou de contournement. L'action de tout plagiaire s'en trouve donc facilitée. Il est donc intéressant de considérer les solutions qui permettraient de limiter cette action. Nous allons présenter un

schéma de détection sécurisé, totalement paramétrable en fonction des besoins, et qui offre une sécurité très satisfaisante contre l'extraction de schéma sans requérir de ressources significativement plus importantes que celle utilisées par les antivirus actuels.

L'extraction d'un schéma pour cette solution sécurisée reste malgré tout possible, mais elle n'est réalisable que dans le cas où un groupe important de pirates s'unissent et partagent leurs ressources[13]. Toutefois, même dans ce cas-là, la complexité de l'algorithme E-2, pour les paramètres utilisés, rendra cette collusion illusoire. Il en résulte que toute tentative de produire des variantes non détectées à partir de variantes connues est, en pratique, condamnée à l'échec.

2.6.1 Constructions combinatoires et probabilistes

Principe général

L'approche générale consiste à considérer un motif de détection principal de taille s octets (répartis dans tout le code). Chaque fois qu'un moteur d'analyse de forme est sollicité par un processus de détection, seul un sous-motif de taille k est utilisé avec les contraintes suivantes :

- le paramètre k doit être relativement faible par rapport au paramètre s ;
- chaque sous-motif est choisi aléatoirement ;
- tout sous-motif est fixe pour un utilisateur et une machine donnée. Son choix dépendra de données caractérisant de manière unique un environnement ;
- le schéma de détection ou de contournement ne peut être reconstruit, même partiellement (par exemple, seul l'ensemble $\mathcal{S}_{\mathcal{M},\mathcal{M}}$) à moins de disposer d'au moins τ sous-motifs ;
- le nombre π de sous-motifs doit être relativement important ainsi que le paramètre τ.

Ces contraintes – complètement paramétrables, comme nous allons le voir – ont été choisies dans le but de maximiser l'incertitude et les efforts du pirate confronté au problème de l'extraction de schéma. Autrement dit, il ne pourra reconstruire un schéma de détection ou de contournement à moins de réunir des conditions rédhibitoires en pratique. Enfin, s'il parvient à produire une variante non détectée avec un ordinateur donné, relativement à un détecteur donné $\mathcal{D}$ et à un sous-motif donné, la probabilité de rester détectable sur d'autres machines relativement à tout autre sous-motif doit rester élevée. Il s'ensuit, sous ces contraintes, que la prolifération de variantes reste d'une portée très limitée.

Le problème principal a été de trouver des objets combinatoires réalisant les contraintes sus-mentionnées. Les meilleurs candidats sont sans aucun doute les objets dénommés *designs combinatoires*[14] [12, 27]. Ces structures particulières

[13] Une autre solution serait pour un pirate d'émuler plusieurs environnements afin de simuler cette collusion. Le schéma proposé non seulement résiste à cette éventualité mais également l'interdira par une implémentation adéquate (voir la section 2.6.1).

[14] Le terme est difficile à traduire et ne semble pas avoir d'équivalent simple en français. Nous garderons le terme anglo-saxon.

présentent de très intéressantes propriétés et caractéristiques permettant de satisfaire aux contraintes souhaitées. En outre, leur implémentation et leur mise en œuvre requiert des ressources temps/mémoire relativement limitées.

La quatrième contrainte suggérait fortement d'utiliser des structures de partage de secret ou des (π, τ)-*schéma à seuil* [92, chapitre 12]. Dans notre contexte, le secret à partager est le motif de détection complet $\mathcal{S}_{\mathcal{M},\mathcal{M}}$ et les différentes *parts* de secret auraient été les différents sous-motifs. Malheureusement, les structures de partage de secret ou de schéma à seuil, qui ont été proposées jusqu'ici, ne concernent que des cas dans lesquels les parts sont des nombres[15]. Dans notre cas, les parts sont des objets plus complexes que des nombres (typiquement des ensembles de nombres). Quelques constructions ont été proposées récemment pour étendre les schémas de partage de secret à des structures complexes comme les graphes [79]. Toutefois, l'intérêt de ces généralisations reste pour le moment purement théorique. Elles requièrent encore trop de ressources, notamment en mémoire, pour être viables dans des antivirus destinés à un usage commercial.

Enfin, notons que la troisième contrainte peut également aider les enquêteurs dans le cadre d'expertises ayant pour but d'incriminer ou d'innocenter l'auteur d'une nouvelle variante non détectée.

Description technique du schéma sécurisé

Nous devons considérer deux aspects pour ce schéma : les objets combinatoires décrivant le motif $\mathcal{S}_{\mathcal{M}}$ lui-même et la fonction de détection $f_{\mathcal{M}}$ (et par conséquent, la fonction $\overline{f_{\mathcal{M}}}$ également). Dans ce qui suit, nous ne rappellerons que les concepts de base concernant les objets combinatoires utilisés. Le lecteur qui souhaiterait les étudier de manière plus détaillée consultera [12, 27].

Les objets combinatoires Nous avons considéré, d'une manière générale des $2-(s,k,\lambda)$ designs (également connus sous le nom de *Balanced Incomplete Block Designs* [BIBD] ou *Designs par blocs, équilibré et complet*). Nous ne rappellerons que la définition de ces objets ainsi que leurs propriétés les plus intéressantes.

Définition 2.7 *Un* Balanced Incomplete Block Design *(BIBD) est une paire* $(\mathcal{V}, \mathcal{B})$ *où* $\mathcal{V}$ *est un ensemble contenant* v *éléments, ou* points *et* $\mathcal{B}$ *est une famille de* b *parties de* $\mathcal{V}$ *(ou* blocs*), chacune ayant une taille constante* k*. Cette paire est telle que tout point de* $\mathcal{V}$ *est contenu dans exactement* r *blocs et telle que toute paire de points de* $\mathcal{V}$ *est contenue dans exactement* λ *blocs. Les valeurs* v, b, r, k, λ *sont les paramètres du BIBD.*

Tout BIBD satisfait alors les propriétés suivantes :

- *un BIBD existe si et seulement si* $vr = bk$ *et si* $r(k-1) = \lambda(v-1)$ *;*
- $r = \frac{\lambda(v-1)}{k-1}$ *;*

[15] Il est intéressant de noter que ces structures sont également réalisables par des *designs combinatoires*.

- $b = \frac{vr}{k}$.

Dans notre contexte, nous avons $\mathcal{S}_{\mathcal{M}} = \mathcal{V}$, $s = v$, $\pi = b = \frac{\lambda v(v-1)}{k(k-1)}$ et $\mathcal{B}$ décrit la famille de sous-motifs, notés $\mathcal{S}^i_{\mathcal{M}}$ pour $i = 1, \ldots, \pi$.

La fonction de détection Nous avons considéré pour ce schéma une fonction booléenne totale à s variables $f : \mathbb{F}_2^s \rightarrow \mathbb{F}_2$ de poids égal à 2^{s-1}. Pour chaque sous-motif de taille k, nous considérons la restriction f^i de f au sous-motif $\mathcal{S}^i_{\mathcal{M}}$, en fixant à une valeur constante les variables X_i non présentes dans ce sous-motif. Ainsi, chaque fonction de détection partielle f^i est de poids 2^{k-1}. En pratique, le choix de la fonction, d'un point de vue structurel (répartition des entrées à valeur non nulle), est intimement lié au choix des octets (indices) contenus dans les $\mathcal{S}^i_{\mathcal{M}}$.

Ainsi, le poids de la fonction totale f et des fonctions partielles f^i est optimal. En effet, ces valeurs maximisent l'effort que l'analyste doit faire pour extraire la fonction de non détection (compte tenu de l'étape LogicalMinimize dans l'algorithme E-2). La fonction retenue est la fonction linéaire (notée, pour limiter l'espace, sous sa forme algébrique normale) :

$$f(X_1, X_2, \ldots, X_s) = X_1 \oplus X_2 \oplus \ldots X_s.$$

L'intérêt de cette fonction, outre ses propriétés intéressantes, tient au fait qu'elle peut être implémentée de manière très compacte. Notons que du point de vue de l'analyste, retrouver la forme algébrique normale (compacte) précédente, ne peut se faire qu'à partir de la DNF produite par l'algorithme E-2. Cette conversion a une complexité exponentielle en $\mathcal{O}(2^s)$. Elle n'est également possible qu'en réalisant une collusion de taille τ.

Proposition 2.3 *La fonction $X_1 \oplus X_2 \oplus \ldots X_s$ est faiblement contournable à l'ordre $s-1$.*

Preuve.
Par calcul de la transformée de Walsh, on montre que le seul coefficient de Walsh $\widehat{\chi_f(u)} = 2^s \neq 0$ est celui pour $u = (1, 1, 1, \ldots, 1)$. D'où le résultat. ■

En considérant la fonction de non détection correspondante $1 \oplus X_1 \oplus X_2 \oplus \ldots X_s$, le plagiaire ne disposera pas de variables ou de groupe de variables plus favorables que d'autres pour produire des variantes non détectées. Il devra les considérer toutes simultanément. En outre, cette fonction de détection, selon la définition 2.4, ne dépend statistiquement que de l'ensemble de variables.

Un autre type de fonction, appartenant à la classe des fonctions fortement contournable a été également considéré. Il s'agit des fonctions MAJORITÉ.

Définition 2.8 *On appelle fonction* MAJORITÉ *à n variables, notée* MAJ_n *la fonction booléenne de $\mathbb{F}_2^n$ dans $\mathbb{F}_2$ telle que*

$$f(x) = 1 \Leftrightarrow \begin{cases} wt(x) \geq \frac{n+1}{2} & \textit{si } n \textit{ impair} \\ wt(x) \geq \frac{n}{2} + 1 & \textit{si } n \textit{ pair} \end{cases}$$

En outre, quand n est pair, exactement $\binom{n}{\frac{n}{2}}$ valeurs x de poids $\frac{n}{2}$ sont telles que $f(x) = 1$.

Nous ne considérerons que les fonctions MAJ_{2p+1}. Les fonctions MAJ_n sont connues pour être équilibrées quelle que soit la valeur n de variables [19].

Les fonctions MAJ_n sont des fonctions fortement contournables, sauf asymptotiquement. En effet nous avons la proposition suivante.

Proposition 2.4 *Les fonctions booléennes MAJ_n sont immunes aux corrélations à l'ordre 0, pour toutes les variables x_i et*

$$P[\mathrm{MAJ}_n(x) = x_i] = \frac{1}{2} + \frac{\binom{n-1}{\frac{n-1}{2}}}{2^n}$$

Le lecteur trouvera la démonstration de cette proposition dans [35]. Cette proposition montre que si les fonctions MAJ_n sont statistiquement dépendantes de chacune de leurs entrées, en revanche elles le sont identiquement pour toutes ces variables. Donc, aucune ne joue un rôle plus important qu'une autre sur la sortie de la fonction.

Mais les fonctions MAJ_n ont, en tant que fonction de détection (ou de non détection), un intérêt particulier comme le prouve la proposition suivante.

Proposition 2.5 *Soit une fonction MAJ_{2p+1}. Sa formule DNF contient alors $\binom{2p+1}{p+1}$ termes. La fonction $\overline{\mathrm{MAJ}_{2p+1}}$ contient également $\binom{2p+1}{p+1}$ termes, chacun d'entre eux contenant $p+1$ variables de la forme x_i (négation).*

Le lecteur trouvera la démonstration dans [53].

Ce résultat montre que si une fonction de détection est une fonction MAJ_{2p+1}, alors le plagiaire devra modifier au minimum $p+1$ variables (octets) pour produire une variante non détectée. Il suffit de considérer des valeurs de p importantes pour compliquer sensiblement sa tâche.

Exemple 2.2 *Soit la fonction MAJ_5 donnée par sa DNF :*

$$\begin{aligned}\mathrm{MAJ}_5 \;=\; & x_4x_3x_2 \vee x_4x_3\overline{x_2}x_1 \vee \overline{x_4}x_3x_2x_1 \vee x_4\overline{x_3}x_2x_1 \vee x_4x_3\overline{x_2x_1} \vee \\ & x_4\overline{x_3}x_2\overline{x_1}x_0 \vee x_4\overline{x_3}\overline{x_2}x_1x_0 \vee \overline{x_4}x_3x_2\overline{x_1}x_0 \vee \overline{x_4}x_3\overline{x_2}x_1x_0 \vee \\ & \overline{x_4x_3}x_2x_1x_0\end{aligned}$$

La DNF de la fonction de non détection correspondante est alors :

$$\begin{aligned}\overline{\mathrm{MAJ}_5} \;=\; & \overline{x_0}\overline{x_1}\overline{x_2} \vee \overline{x_0}\overline{x_1}x_2\overline{x_3} \vee \overline{x_0}\overline{x_1}x_2x_3\overline{x_4} \vee \overline{x_0}x_1\overline{x_2}\overline{x_3} \vee \overline{x_0}x_1\overline{x_2}x_3\overline{x_4} \vee \\ & \overline{x_0}x_1x_2\overline{x_3}\overline{x_4} \vee x_0\overline{x_1}\overline{x_2}\overline{x_3} \vee x_0\overline{x_1}\overline{x_2}x_3\overline{x_4} \vee x_0\overline{x_1}x_2\overline{x_3}\overline{x_4} \vee x_0x_1\overline{x_2}\overline{x_3}\overline{x_4}\end{aligned}$$

La fonction de non détection ne vaudra 1 que si au moins trois octets sont modifiés (voir annexe 10 pour la signification des ces notations).

Le protocole d'analyse Durant la phase initiale d'installation, le logiciel antivirus collecte un certain nombre d'informations concernant à la fois l'utilisateur et le système :

- le numéro de série du processeur (CPUID) et celui du disque dur (HDID) ;
- l'adresse MAC si elle est disponible (notée `MACadr`) ;
- le nom utilisateur `USRname` et son adresse *e-mail* `@adr` ;
- une valeur secrète ν, dissimulée dans le logiciel antivirus[16].

Précisons, que d'autres paramètres peuvent être considérés. Cette phase est pleinement paramétrable. Les paramètres peuvent également être choisis aléatoirement dans une liste.

Ensuite, le logiciel calcule l'index i du sous-motif qui sera utilisé de manière fixe, comme suit :

$$i = g(H(\text{CPUID} \oplus \text{HDID} \oplus \texttt{MACadr} \oplus \texttt{USRname} \oplus \texttt{@adr} \oplus \nu) \oplus \mathcal{N}).$$

La fonction H est une fonction de hachage dont l'entrée est la somme bit à bit, modulo 2, des données collectées et codées sous forme d'entiers. La fonction g produit une valeur aléatoire comprise entre 1 et π, à partir du résultat de H et d'un entier d'initialisation N, éventuellement public. En conséquence, le même sous-motif de détection i sera utilisé en permanence lors de toute opération de scan par l'utilisateur sur cette machine. L'objectif est double :

- d'une part, l'auteur potentiel d'une variante la produira relativement à un unique sous-motif donné ;
- d'autre part, en cas d'enquête, l'analyse permettra de prouver ou d'infirmer l'implication de l'utilisateur dans la production de cette variante. Il suffira à l'enquêteur de recalculer l'indice i à partir des informations contenues dans la machine du suspect pour déterminer si le contournement de détection a pu être réalisé relativement au sous-motif i.

2.6.2 Analyse mathématique

Analysons maintenant ce schéma de détection sécurisé. Notons $\{\mathcal{S}_\mathcal{M}, f_\mathcal{M}\}$ le schéma de détection complet et $\{\mathcal{S}^i_\mathcal{M}, f^i_\mathcal{M}\}$ le schéma de détection partiel relativement au sous-motif i.

Transinformation du schéma Avec les notations précédentes et par définition de notre schéma, nous avons de manière évidente :

$$I(\mathcal{S}_{\mathcal{M},\mathcal{M}}; \overline{f_\mathcal{M}}, \mathcal{S}_\mathcal{M}) = H(\mathcal{S}^i_{\mathcal{M},\mathcal{M}}) << H(\mathcal{S}_{\mathcal{M},\mathcal{M}}) - H(\mathcal{S}_{\mathcal{M},\mathcal{M}} | f^i_\mathcal{M}, \mathcal{S}^i_\mathcal{M}).$$

L'analyse en boîte noire ne permet de retrouver que $\{\mathcal{S}^i_\mathcal{M}, f^i_\mathcal{M}\}$.

[16] Elle peut être obtenue par rétro-ingénierie logicielle mais, en cas d'enquête, son utilisation pour produire une variante non détectée constituera une preuve de contrefaçon.

Rigidité du schéma Comme nous l'avons vu précédemment, elle dépend directement du poids de la fonction $\mathrm{wt}(f^i_{\mathcal{M}})$ et de k. Dans le cas de ce schéma, nous avons par conséquent :

$$\mathcal{R}(\mathcal{S}^i_{\mathcal{M}}, f^i_{\mathcal{M}}) = \frac{2^k - 1}{k2^k} = \frac{1}{k} - \frac{1}{k2^k} = \mathcal{O}(\frac{1}{k}).$$

Il est donc préférable d'avoir des valeurs de k relativement petites pour que le schéma possède la meilleure rigidité possible (voir section 2.3.2).

Impact d'une collusion Nous pouvons supposer qu'un groupe d'attaquants se constitue afin de mettre en commun leurs efforts pour extraire un schéma de détection donné. Déterminons la taille minimale que doit avoir cette collusion pour résoudre le problème de l'extraction.

Proposition 2.6 *Le schéma $\mathcal{SD} = \{\mathcal{S}_{\mathcal{M}}, f_{\mathcal{M}}\}$ ne peut être extrait que par une collusion d'au moins $\tau = \lceil \frac{s}{k} \rceil$ membres.*

Preuve.
La preuve est évidente en considérant que chaque bloc contient k points. De là, nous avons $\tau \geq \lceil \frac{s}{k} \rceil$. Cela correspond au cas dans lequel les blocs (sous-motifs) utilisés par la collusion ont une intersection nulle. Dans ce cas, le design est une classe parallèle d'un design dit *résolvable*[17]. ■

Lorsque τ attaquants forment une collusion, ils extraient un motif de détection $\mathcal{S}_{\mathcal{M}}$ de taille s. Voyons à présent quelle forme possède la formule DNF décrivant la fonction de détection.

Proposition 2.7 *La formule DNF de la fonction de détection extraite par une collusion d'au moins $\tau = \lceil \frac{s}{k} \rceil$ analystes contient au plus $\frac{s}{k}(2^{k-1})$ termes.*

Preuve.
La preuve est évidente en considérant que l'intersection de toute famille de blocs ne contient aucun bloc du design et que deux quelconques blocs peuvent être d'intersection non vide. La DNF contiendra exactement $\frac{s}{k}(2^{k-1})$ termes quand cette intersection est systématiquement vide (cas d'une classe parallèle). ■

Ce résultat implique que même si τ analystes venaient à s'unir, la complexité de la procédure LogicalMinimize leur serait extrêmement défavorable et la

[17] Dans un RBIBD *(Resolvable Balanced Incomplete Bloc Design)*, la famille $\mathcal{B}$ est une partition dont chaque partie est une *classe parallèle*. Une classe parallèle est un ensemble de blocs partitionnant l'ensemble $\mathcal{V}$. Deux conditions sont nécessaires pour qu'un BIBD soit résolvable : (1) $k|v$ et (2) $b \geq v + r - 1$. À titre d'exemple, considérons le $(9, 3, 1)$-RBIBD. L'ensemble $\mathcal{B}$ est alors

$$\begin{array}{cccc} \{1,2,3\} & \{1,4,7\} & \{1,5,9\} & \{1,6,8\} \\ \{4,5,6\} & \{2,5,8\} & \{2,6,7\} & \{2,4,9\} \\ \{7,8,9\} & \{3,6,9\} & \{3,4,8\} & \{3,5,7\}. \end{array}$$

fonction de non détection ne pourrait être extraite (pour des valeurs appropriées de s et de k).

Probabilité de détection résiduelle Supposons à présent qu'un plagiaire parvienne à extraire un schéma de détection partiel $\{\mathcal{S}^i_{\mathcal{M}}, f^i_{\mathcal{M}}\}$ relativement au sous-motif i et pour un détecteur $\mathcal{D}$ implémentant la schéma sécurisé proposé. Il est alors capable de produire une nouvelle variante, non détectée par $\mathcal{D}$. Quelle est alors la probabilité que cette variante reste détectée une fois disséminée par le plagiaire (probabilité de détection résiduelle) ?

Proposition 2.8 *La connaissance du schéma de détection partiel* $\{\mathcal{S}^i_{\mathcal{M}}, f^i_{\mathcal{M}}\}$ *permet de générer une variante qui reste détectée avec une probabilité notée* $P_{détection}$ *telle que*

$$\frac{1}{2} \leq P_{détection} \leq 1.$$

Preuve.
Supposons que le plagiaire utilise $\mathcal{S}^i_{\mathcal{M}}$ et $f^i_{\mathcal{M}}$ pour produire une variante non détectée. Supposons que cette variante se propage sur une machine dont le détecteur utilise le sous-motif j. Les modifications opérées pour produire cette variante vont affecter la détection relativement au sous-motif j avec une probabilité (en pire cas) de $\frac{1}{2}$, sauf si les sous-motifs i et j sont d'intersection vide. Dans ce dernier cas, il est évident, par construction, que $P_{\text{détection}} = 1$. D'où le résultat. ■

Ce résultat montre que les objets combinatoires et la fonction de détection doivent être soigneusement choisis. En particulier, l'utilisation de `RBIBD` est préférable à celle de BIBD simples. Mais, même dans ce dernier cas, les différentes implémentations et expériences réalisées ont montré que $P_{\text{détection}}$ reste plus proche de 1 que de $\frac{1}{2}$. Dans le cas le plus favorable (cas des `RBIBD`), nous avons $P_{\text{détection}} = \frac{\pi-1}{\pi}$.

2.6.3 Implémentations et performances

Les différentes implémentations et leur test ont montré, jusqu'à présent, que les meilleurs objets combinatoires, pour les contraintes fixées, restent les classes parallèles de `RBIBD`. D'autres `BIBD` non résolvables ont également été utilisés avec succès mais dans ce dernier cas, le choix de la fonction de détection est plus délicat. Il faut en effet tenir compte du fait que les blocs peuvent être d'intersection disjointe.

Les ressources mémoire requises par ce schéma sont en $\mathcal{O}(\frac{s^2}{k})$ si l'on considère un $2-(s,k,\lambda)$ design générique (cela est dû essentiellement à la taille de la matrice d'incidence du design). Mais des considérations de phylogénie de codes [60, 71] devraient permettre de choisir des objets combinatoires bien meilleurs, pour gérer plusieurs variantes à la fois et ainsi améliorer la complexité mémoire. En termes de complexité de calcul et donc de temps de traitement,

les expériences ont montré qu'il n'existait aucune différence significative entre les plus rapides des scanners actuellement disponibles et une implémentation relativement optimisée du schéma sécurisé.

Même si des progrès en termes d'implémentation peuvent encore être faits dans l'avenir, les résultats expérimentaux prouvent que ce schéma peut être utilisé en pratique et peut constituer une alternative, commercialement viable, aux techniques d'analyse de forme actuellement utilisées... avec le bénéfice non négligeable d'une meilleure lutte contre la prolifération virale.

2.7 L'analyse comportementale

Le concept de détection comportementale a été originellement introduit par Fred Cohen [26, pp. 73]. Mais ce type de détection, comme son homologue fondée sur la forme, correspond également à un problème indécidable. Cependant, la détection fonctionnelle – ce sont les « comportements » ou les action du code qui sont étudiées – est présentée comme une technique prometteuse. Certes, faire la différence entre des comportements légitimes et malicieux reste un problème difficile sans réelle solution efficace : problème de fausses alarmes ou pire de mimétisme malicieux avec des fonctions légitimes (voir chapitre 3). Mais la détection comportementale, au moins sur le plan marketing, reste un bon critère. Qu'en est-il vraiment ?

Jusqu'à présent, les méthodes d'évaluation concernent la détection sur la forme [52, 70] et à ce jour pratiquement aucune méthode d'évaluation des moteurs comportementaux n'existe vraiment. Les techniques de désassemblage ne pouvant être officiellement utilisées, restent les techniques d'analyse en boîte noire comme celles présentées précédemment. Or, les produits antivirus actuels ne permettent pas de sélectionner les techniques utilisées. Il est par conséquent impossible d'isoler véritablement la partie comportementale de celle travaillant exclusivement sur la forme du code.

Pour finir avec la partie consacrée à l'analyse de la défense, nous allons présenter une technique d'analyse en boîte noire des moteurs comportementaux. Cette technique [53] en est à ses débuts mais constitue une base prometteuse, comme les quelques résultats présentés vont le montrer. Le principe est d'utiliser en quelque sorte ce nous pourrions appeler du polymorphisme/métamorphisme comportemental. Autrement dit, plutôt que de modifier sélectivement des octets, nous allons modifier sélectivement les fonctions et comportements du code. Cela implique une phase préalable d'analyse du code source pour identifier les actions réalisées. Cette approche a permis de généraliser la notion de schéma de détection, telle que définie dans la section 2.3.1, à celle de stratégie de détection, dans laquelle il n'est plus fait réellement de différence entre octets du code et fonctions du code.

2.7.1 Modèle de stratégie de détection

La définition 2.1 de la section 2.3.1 ne considérait que l'une ou l'autre forme de détection. Nous allons donc utiliser une définition plus générale de la notion de détection avec le concept de *stratégie de détection.*

Définition 2.9 *(Stratégie de détection) Une* stratégie de détection $\mathcal{SD}$ *relativement à un code malveillant donné* $\mathcal{M}$ *est le triplet* $\mathcal{DS} = \{\mathcal{S}_\mathcal{M}, \mathcal{B}_\mathcal{M}, f_\mathcal{M}\}$*, où* $\mathcal{S}_\mathcal{M}$ *est un ensemble d'octets,* $\mathcal{B}_\mathcal{M}$ *un ensemble de fonctions de programme et* $f_\mathcal{M} : \mathbb{F}_2^{|\mathcal{S}_\mathcal{M}|} \times \mathbb{F}_2^{|\mathcal{B}_\mathcal{M}|} \rightarrow \mathbb{F}_2$ *une fonction booléenne.*

Il est intéressant de noter que cette définition concerne à la fois les codes malveillants connus et ceux éventuellement inconnus (mais utilisant néanmoins des techniques ou des modes opératoires connus). En effet, lorsqu'un code malveillant inconnu $\mathcal{M}$ déclenche une alerte, c'est précisément l'ensemble $\mathcal{B}_\mathcal{M}$ qui est alors impliqué. C'est là tout l'intérêt de la notion de stratégie de détection par rapport à celle de schéma de détection. Si la nature de l'ensemble $\mathcal{S}_\mathcal{M}$ est facile à appréhender – un ensemble d'octets –, celle de l'ensemble $\mathcal{B}_\mathcal{M}$ l'est probablement moins.

En fait, cet ensemble peut être considéré comme un méta-ensemble d'octets de la manière suivante : les comportements peuvent être décrits par des structures d'octets qui correspondent à chaque procédure réalisant une action ou un comportement donné. Accéder à un fichier en lecture, créer un mutex sont des actions pouvant être décrites au moyen de telles structures plus ou moins complexes d'octets, localisées soit sur le disque dur (le code est inactif, l'analyse se fait par émulation de code) ou en mémoire (le code est actif).

À titre d'exemple, pour surveiller le comportement consistant à tenter d'ouvrir en écriture le secteur de démarrage maître ou secondaire, il est possible de dérouter l'interruption 13H, service 3 de la manière suivante.

```
INT_13H:
     cmp     CX, 1    ; est-ce le cylindre 0, secteur 1 ?
     jnz     DO_OLD   ; sinon on rend la main à l'appel 13H
                      ; original
     cmp     DH, 0    ; est-ce la tête 0 ?
     jnz     DO_OLD   ; sinon on rend la main à l'appel 13H
                      ; original
     cmp     AH, 3    ; est-ce un service en écriture ?
     jnz     DO_OLD   ; sinon on rend la main à l'appel 13H
                      ; original
     .....

DO_OLD:
     jmp     dword ptr CS:[OLD_13H] ;
```

Cette tentative d'écriture est identifiée par un ensemble d'octets décrivant en détail la nature du service et les paramètres afférents. Quand à l'exécution du

code, cette structure d'octets est réalisée, le comportement est alors identifié. D'un point de vue formel donc, nous avons $\mathcal{B}_{\mathcal{M}} \subset \mathbb{N}_{256}^{\infty}$ (une famille de séquences d'octets de longueur indéfinie). Dans ce qui suit, nous parlerons simplement de comportement. Dire que le comportement $b \in \mathcal{B}_{\mathcal{M}}$ est réalisé signifie que le code contient ou réalise une structure d'octets lors de son exécution.

Comme nous l'avons fait dans la section 2.3.1, nous allons préciser le modèle mathématique utilisé. Il est assez proche du précédent mais il est préférable de détailler le formalisme pour mieux expliciter les différences avec le modèle restreint. Expliquons, comme précédemment, comment fonctionne un détecteur antiviral donné $\mathcal{D}$ sur un fichier $\mathcal{F}$ suspecté d'être infecté par un code malveillant $\mathcal{M}$. Nous définissons tout d'abord les $s + b$ variables binaires X_j $(1 \leq j \leq s + b)$ comme suit :

$$X_j = \begin{cases} 1 & \text{si } \mathcal{F}(i_j) = b_{\sigma(j)} \\ 0 & \text{sinon.} \end{cases}$$

Cette notation permet effectivement de considérer indifféremment octets de code et fonctions de code. Elle permet tout autant de décrire une éventuelle modification d'octets de $\mathcal{S}_{\mathcal{M}}$ (approche présentée avec l'analyse en boîte noire des schémas de détection) que des modifications de comportements de $\mathcal{B}_{\mathcal{M}}$, destinées à contourner la détection de $\mathcal{D}$. Ainsi, $X_j = 0$ signifie que nous avons effectivement modifié $\mathcal{S}_{\mathcal{M}}(j)$ ou $\mathcal{B}_{\mathcal{M}}(j)$. L'association de tout octet de $\mathcal{S}_{\mathcal{M}}$ et tout comportement de $\mathcal{B}_{\mathcal{M}}$ à une variable booléenne permet de considérer les ensembles booléens $\mathbb{F}_2^{|\mathcal{S}_{\mathcal{M}}|}$ et $\mathbb{F}_2^{|\mathcal{B}_{\mathcal{M}}|}$ respectivement. Nous avons donc $s = |\mathcal{S}_{\mathcal{M}}|$ et $b = |\mathcal{B}_{\mathcal{M}}|$. Par souci de clarté, nous considérerons, à une permutation des indices près, uniquement l'ensemble $\mathbb{F}_2^{|\mathcal{S}_{\mathcal{M}} \cup \mathcal{B}_{\mathcal{M}}|}$ dont le cardinal est donné par $2^n = 2^{s+b}$.

Considérons à présent une fonction booléenne $f_{\mathcal{M}} : \mathbb{F}_2^n \to \mathbb{F}_2$. Nous dirons qu'un détecteur décide que $\mathcal{F}$ est infecté par $\mathcal{M}$, relativement à la fonction de détection $f_{\mathcal{M}}$, si et seulement si $f_{\mathcal{M}}(X_1, X_2, \ldots, X_n) = 1$. Plus précisément :

$$f_{\mathcal{M}}(X_1, X_2, \ldots, X_n) = \begin{cases} 1 & \mathcal{F} \text{ est infecté par } \mathcal{M} \\ 0 & \mathcal{F} \text{ n'est pas infecté par } \mathcal{M}. \end{cases}$$

Comme précédemment, la fonction de détection sera considérée sous sa forme disjonctive normale. Nous définissons également la notion de fonction de non détection $\overline{f_{\mathcal{M}}} = 1 \oplus f_{\mathcal{M}}$. Elle décrit la manière dont un code malveillant peut être modifié afin de ne plus être détecté par la stratégie de détection $\{\mathcal{S}_{\mathcal{M}}, \mathcal{B}_{\mathcal{M}}, f_{\mathcal{M}}\}$. Ces modifications correspondent aux n-uplets $(x_1, x_2, \ldots, x_n)$, pour lesquels $\overline{f_{\mathcal{M}}} = 1$. Pour un tel n-uplet, toute modification peut être définie comme suit :

$$\begin{cases} \text{si } x_i = 0 & \text{l'octet ou le comportement } i \text{ dans } \mathcal{S}_{\mathcal{M}} \cup \mathcal{B}_{\mathcal{M}} \\ & \text{doit être modifié} \\ \text{si } x_i = 1 & \text{l'octet ou le comportement } i \text{ dans } \mathcal{S}_{\mathcal{M}} \cup \mathcal{B}_{\mathcal{M}} \\ & \text{peut rester non modifié.} \end{cases}$$

Nous généraliserons également la notion de schéma de contournement par la définition suivante.

Définition 2.10 *Une* statégie de contournement $\mathcal{SC}$ *relativement à un code malveillant donné* $\mathcal{M}$ *est le triplet* $\mathcal{SC} = \{\mathcal{S}_{\mathcal{M}}, \mathcal{B}_{\mathcal{M}}, \overline{f_{\mathcal{M}}}\}$.

Nous ne développerons pas les choses ici mais il est important de dire que les propriétés présentées dans la section 2.3.2 s'appliquent et se généralisent également à la notion de stratégie de détection.

2.7.2 Méthode d'évaluation de la détection comportementale

Dans l'étude menée [53, 67], il a été supposé[18] que la détection comportementale est effectivement utilisée et qu'à côté des bases de signatures classiques et des techniques d'analyse de forme, des bases de comportements interdits étaient utilisées. L'approche par analyse en boîte noire et extraction des données de détection, qui a été présentée au début de ce chapitre, a été généralisée. Mais dans le cas présent, nous avons sélectivement modifié des actions et des comportements de programmes. Une fois soumis à la détection des produits testés, les résultats – détection ou non, messages de détection – ont permis de reconstruire la stratégie de détection réellement mise en œuvre. La modification d'un ou plusieurs comportements dans un code a permis de réaliser ce que l'on peut qualifier de polymorphisme/métamorphisme comportemental. Nous allons montrer comment cela a été réalisé sur le code du ver de courrier électronique *W32/MyDoom*.

Polymorphisme/métamorphisme comportemental du ver *W32/MyDoom*

L'idée sous-jacente est de simuler la variabilité fonctionnelle du code, d'une manière sélective et contrôlée. Différentes versions fonctionnelles ou comportementales du ver ont ainsi été produites, comme si le ver avait lui-même réalisé cette variation fonctionnelle. Le ver *W32/MyDoom* a été choisi car, d'une part, il constitue une référence dans le domaine fonctionnel viral [39]. Beaucoup de codes malveillants ont repris par la suite un grand nombre de ses comportements. D'autre part, son code source est disponible dans sa version d'origine ce qui évite de passer par une phase de désassemblage. Bien sûr, cette technique est applicable à tout autre type de code malveillant, pour peu qu'il présente un minimum de diversité fonctionnelle.

Ce polymorphisme/métamorphisme comportemental a été réalisé manuellement. Le concept de polymorphisme/métamorphisme comportemental n'a pratiquement jamais été étudié, au contraire de son homologue appliqué à la forme d'un code. L'automatisation de cette approche – laquelle est en cours au laboratoire de virologie et de cryptologie de l'École Supérieure et d'Application des

[18] Cette supposition a été faite sur la base des affirmations marketing de la plupart des éditeurs dont les produits ont été testés lors de cette étude.

Transmissions – nécessite au préalable d'étudier de manière formelle la notion de variabilité fonctionnelle.

Identification des comportements Avant de procéder à une quelconque modification de comportements dans le code, la première étape consiste déjà à identifier les différents comportements mis en œuvre par le ver. Or cette analyse est beaucoup plus complexe que d'identifier des signatures ou autres caractéristiques de forme.

L'analyse en profondeur du code source du ver *W32/MyDoom* a permis d'identifier sa structure fonctionnelle ainsi que les relations et interdépendances des fonctions. Chaque comportement peut se décomposer en une ou plusieurs actions caractéristiques. Une action peut être elle-même une fonctionnalité basique ou simplement une façon de programmer. Le tableau 2.3 résume les principaux comportements identifiés dans le ver *W32/MyDoom* ainsi que les actions qui révèlent leur expression. Le choix des comportements dans un code est

Référence	Comportements	Actions
DUPLI	Réplication du code	Copie du fichier dans le répertoire système
RESID	Mise en résidence	Utilisation d'une clef dans la base de registre
PROPA	Propagation	Envoi en masse de courriers avec le ver en attachement
OVINF	Test de surinfection	Test de présence d'une clef donnée dans la base de registre
ACTIV	Test d'activité	Test de présence d'un mutex actif en mémoire
STEAL	Furtivité	Mise en place de sa propre pile de protocoles réseau
POLYM	Polymorphisme	Chiffrement de la librairie de *backdoor*
INFOR	Collecte d'informations	Scan récursif des fichiers
FINAL	Charge finale	Installation d'une *backdoor*
SOCIA	Ingénierie sociale	Simulation de récupération de courriers perdus

Table 2.3 – Comportements identifiés dans le ver *W32/MyDoom*

avant tout un choix subjectif. Selon le degré de finesse souhaité, il sera possible de jouer sur la granularité des actions. Toutefois, à l'heure actuelle, les produits existants peuvent être facilement testés et évalués avec une granularité relativement grossière.

Modification du code par comportements alternatifs Après avoir identifié les comportements susceptibles d'être considérés dans la détection, la seconde étape consiste à les modifier selon le principe « action différente/résultat identique ». En d'autres termes, la modification des fonctions ne doit pas produire un résultat global différent. Cette étape est essentielle et assez délicate

Comportements	Référence	Nature des modifications
DUPLI	DUP_SH_CUT	Recopie dans un fichier raccourci
	DUP_NAM_PATH	Recopie dans un répertoire de désinstallation de correctifs
RESID	RES_SERV_KEY	Inscription du ver sous une clef liée à une service
	RES_WIN_INI	Modification du fichier win.ini
OVINF	INF_DIF_KEY	Test d'existence d'une clef différente
	INF_SUP_HID	Test d'existence d'un fichier de type « supercaché »
	INF_ENV_VAR	Test d'existence d'une variable d'environnement
ACTIV	ACT_MUTEX	Test d'existence d'un *mutex* différent
	ACT_EVENT	Test d'existence d'un objet événement
POLYM	POL_PLAIN_LIB	Librairie de *backdoor* en clair
	POL_FLOW_LIB	Librairie de *backdoor* chiffrée par système par flot
	POL_PLAIN_STR	Chaînes de caractères en clair
	POL_FLOW_STR	Chaînes de caractères chiffrées
FINAL	FIN_TRIG_TARG	Cibles et gâchettes différentes pour l'attaque DDOS
	FIN_NO_BDOOR	Suppression de la librairie de *backdoor*

Table 2.4 – Nature des modifications comportementales

à envisager. Il est important de rappeler que nous devons penser en terme de fonctionnalités de programme. Il faut trouver des comportements différents mais néanmoins équivalents et qui doivent aboutir à une action identique du ver *W32/MyDoom*. À titre d'illustration, si un courtisan veut tuer le roi, il peut utiliser du poison. Si un goûteur (que l'on peut comparer à l'antivirus) est employé, il suffira de placer une bombe, sachant que les services du roi n'emploient pas de détecteurs d'explosifs.

Le tableau 2.4 résume les actions alternatives qui ont été retenues pour remplacer les comportements originels du ver *W32/MyDoom*. D'autres mo-

difications auraient pu être considérées, éventuellement plus élaborées. Mais d'une part, l'objectif est de valider une approche d'évaluation. D'autre part, les résultats obtenus (voir section 2.7.3) ont montré que pour la détection comportementale réellement mise en œuvre par les antivirus actuels, les actions retenues sont déjà très élaborées. De plus triviales auraient grandement suffi. Le lecteur consultera [67, pp. 26-29] pour la description détaillée de ces modifications. Nous allons en détailler deux afin de mieux comprendre comment les modifications fonctionnelles sont imaginées et mises en œuvre.

Test de surinfection. L'objectif pour le code malveillant est de déterminer s'il a déjà infecté la cible en cours. Dans le cas du ver *W32/MyDoom*, cette vérification se fait en recherchant une clef donnée dans la base de registre de Windows. Cette clef, qui sert de marqueur d'infection, a été mise en place par le ver lors de la primo-infection. Une première variante consisterait à modifier la valeur de la clef mais ce faisant, nous serions dans le cas d'un polymorphisme classique (modification seule de la forme). Parmi de nombreuses autres possibilités, nous avons décidé de tester au contraire la présence d'un fichier de type supercaché (preuve d'infection) ou son absence (hôte non encore infecté). Ce type de fichier permet de dissimuler tout fichier même lorsque la fonction d'affichage des fichiers cachés est activée. Pour forcer les fichiers de type supercaché à être visibles, des droits administrateurs ainsi que des modifications dans la base de registres sont requis. Notons que cette caractéristique est non documentée. Pour déclarer un fichier de type supercaché, il faut juste définir le paramètre suivant :

```
#define FILE_ATTRIBUTE_SUPERHIDDEN 0x00000006
```

Ainsi pour marquer l'infection, le code utilise la fonction *CreateFile* pour créer un fichier de type supercaché dans un endroit secret connu de lui seul (les endroits me manquent pas sous Windows). Pour tester la présence d'une infection, le ver recherche ce fichier qui n'existe que pour lui.

Test d'activité mémoire. Dans ce cas, le ver *W32/MyDoom* recherche un *mutex* particulier [39]. La première modification qui vient à l'esprit est de changer le nom du *mutex* mais conceptuellement nous serions dans un cas de polymorphisme classique, lié à la forme. Cependant, le système d'exploitation Windows utilise bien d'autres objets impliqués dans la synchronisation inter-processus. Ainsi, pour tester l'activité du ver en mémoire, nous avons utilisé un objet différent et une action différente. Quoique simple, nous avons utilisé l'objet suivant :

```
TaskmonInitialised                    Event
```

Dans ce cas – comme dans tous les autres cas possibles –, le code vérifie le code d'erreur retourné chaque fois que l'objet est créé.

Protocole expérimental Le premier problème à résoudre tient au fait que la plupart des produits antivirus ne permettent pas de choisir entre la détection par analyse de forme et celle par étude comportementale. À vrai dire, aucune distinction claire n'apparaît même dans les différentes fenêtres de configuration

ou les « aides » en ligne. Il est pratiquement toujours impossible de déterminer lequel des deux types de détection est réellement mis en œuvre. D'un point de vue pratique, une régle empirique générale, qui a fonctionné jusqu'à présent de manière très satisfaisante, permet de déterminer laquelle des techniques de détection semble être utilisée. Toute identification exacte (incluant nom du code et numéro de version, par exemple *W32/MyDoom.A*) se fait au moyen d'une analyse de forme classique. En revanche, toute identification générique (par exemple *MassMailing.gen*) est susceptible de plus ou moins mettre en œuvre de la détection comportementale.

Cette incapacité à désactiver la détection par analyse de forme oblige, dans une première phase, à extraire les caractéristiques de cette dernière (par une analyse en boîte noire par exemple). Une fois identifiées et modifiées au niveau de la forme seule, la détection du code malveillant peut éventuellement être opérée sur les seuls comportements par des moteurs appropriés. En effet, une première détection avec succès d'un code par analyse de forme s'arrêtera là et les moteurs comportementaux ne seront pas utilisés (question d'optimisation).

Notre méthodologie comporte donc deux étapes. La première vise à tester les différentes versions produites de manière statique, par un classique scan manuel (détection à la demande). Cela permet de rapidement déterminer les éléments de forme impliqués[19]. Lors de la seconde étape, la protection résidente a été activée. C'est en effet elle qui est supposée mettre en œuvre la détection comportementale[20]. En corrélant les informations obtenues dans chacune des phases, il est alors possible d'extraire l'information relative à la détection comportementale réellement utilisée.

Sept antivirus ont été testés (voir tableau 2.5). Nous avons choisi ceux qui affirment utiliser de la détection comportementale. Les résultats détaillés, produit par produit sont disponibles dans [53] et partiellement dans [67].

2.7.3 Résultats expérimentaux et interprétation

Il est essentiel de rappeler que les résultats et interprétations présentés ici concernent un code malveillant donné. Il reste à reproduire ces expériences pour avoir une vision complète et en profondeur de la manière dont la détection comportementale est mise en œuvre, si cela est le cas.

En première approche, on peut affirmer que la modélisation par comportements suspects a été choisie par les développeurs d'antivirus. Il est probable qu'une vision fondée sur l'observation de comportements légitimes, comme cela est le cas dans les systèmes de détection d'intrusion (IDS), serait peut-être plus efficace [97].

[19] Avec suffisamment de temps, la technique présentée en début de chapitre peut être utilisée, mais dans le cas de *W32/MyDoom*, il a été plus facile de procéder ainsi.

[20] Nous n'avons pas considéré le cas de l'émulation de code [38, chapitre 4]. Pour les produits actuels, seule la détection par analyse de forme est utilisée. Une évolution future des antivirus serait de systématiquement utiliser des moteurs comportementaux dans les techniques de détection par émulation de code.

Produits	Version	Base virale
Avast	4.6.763	0611-2
AVG	7.1.375	267.9.2/52
DrWeb	4.33.2.12231	10062006
F-Secure 2005	6.12-90	2006-06-02-02
G-Data	AVK 16.0.3	KAV-6.818/BD-16.864
KAV Pro	6.0	07062006
Viguard	11	NA

Table 2.5 – Logiciels antivirus évalués vis-à-vis de la détection comportementale (version et base virale)

En termes de stratégie de détection, les résultats actuels montrent clairement que la détection comportementale n'est pas réellement utilisée, excepté pour l'antivirus *Viguard*. Cela peut être formulé par les deux hypothèses suivantes, concernant la fonction de détection :

- $\mathcal{H}_1$: la détection comportementale n'est pas implémentée ou elle est inefficace,
- $\mathcal{H}_2$: la détection comportementale est ignorée sans une validation par la détection par analyse de forme (signature par exemple).

Pour le produit *Viguard*, une hypothèse particulière peut être formulée.

- $\mathcal{H}_3$: la détection comportementale consiste à considérer que tout comportement est potentiellement menaçant.

Un fait simple permet de bâtir ces hypothèses. Chaque fois qu'un virus est détecté de manière précise, la détection par analyse de forme est utilisée (bases de signatures, par exemple). En revanche, quand seule une détection générique intervient, il est possible de supposer que seule de la détection comportementale est impliquée.

D'un point de vue mathématique, l'utilisation des fonctions booléennes (voir section 2.7.1) permet de définir les hypothèses précédentes plus rigoureusement :

$$\begin{aligned}
&\mathcal{H}_1 : T_{sig} \\
&\mathcal{H}_2 : T_{sig} \vee (T_{sig} \wedge T_{behav}) = T_{sig} && \text{(par la loi d'absorption)} \\
&\mathcal{H}_3 : T_{behav} = \bigvee_{i=0}^{b} X_i && \text{où } X_i \text{ décrit le } i\text{-ième élément} \\
& && \text{dans la base comportementale base } \mathcal{B}_{\mathcal{M}}
\end{aligned}$$

La notation T_{sig} désigne la restriction fonctionnelle $f_{\mathcal{M}}^{\mathcal{S}_{\mathcal{M}}}$ de la fonction générale de détection $f_{\mathcal{M}}$ à l'ensemble $\mathcal{S}_{\mathcal{M}}$. Cela signifie que les variables booléennes relatives à l'ensemble $\mathcal{B}_{\mathcal{M}}$ n'apparaissent pas dans la forme normale disjonc-

tive de $f_{\mathcal{M}}^{\mathcal{S}_{\mathcal{M}}}$. Inversement, s'agissant de l'hypothèse $\mathcal{H}_3$, nous considérons la restriction fonctionnelle $f_{\mathcal{M}}^{\mathcal{B}_{\mathcal{M}}}$ à l'ensemble des comportements $\mathcal{B}_{\mathcal{M}}$, avec les différences importantes que :

- l'ensemble $\mathcal{B}_{\mathcal{M}}$ contient tous les comportements possibles qui peuvent potentiellement être utilisés par un code malveillant (incluant ceux qui sont également utilisés par les programmes non malveillants) ;
- la fonction de détection est la fonction logique OU, dont le poids est $2^{|\mathcal{B}_{\mathcal{M}}|} - 1$.

La formalisation mathématique de ces hypothèses montre clairement que les deux premières sont équivalentes et aboutissent à la même et unique conclusion. Dans les deux cas, l'utilité de la détection comportementale, s'il y en a, est sujette à caution. Sans aucun doute, cela s'explique par un choix inadapté de la fonction de détection dans la stratégie de détection. La volonté d'obtenir une probabilité de fausses alarmes la plus faible possible ainsi que les compromis et choix algorithmiques faits inhibent totalement la propriété essentielle de la détection comportementale : la détection de virus inconnus mais utilisant des techniques connues (le cas le plus fréquent). Les deux précédentes formules logiques sont les plus triviales que l'on puisse imaginer. Il existe bien d'autres fonctions booléennes possibles.

À l'opposé, la stratégie *Viguard* est tout aussi inefficace. Un poids maximal est accordé à tout comportement possible, accroissant d'une manière absurde la probabilité de fausses alarmes.

2.8 Problèmes ouverts et conclusion

La recherche, l'exploration et le classement des des « bonnes » fonctions de détection est un champ de recherche essentiel. Il est également prometteur. Un autre axe de recherche non moins essentiel consiste à déterminer si d'autres propriétés sont à considérer pour ces fonctions, afin d'augmenter à la fois l'efficacité des schémas de détection et la résistance à l'analyse en boîte noire et par conséquent la production de nouvelles variantes.

Tout cela constitue un ensemble de problèmes ouverts auxquels il est essentiel d'apporter des solutions. Parmi eux (la liste est non exhaustive), signalons :

- la classification et l'exploration de bonnes fonctions de détection $f_{\mathcal{M}}$;
- la recherche de propriétés structurelles pour ces fonctions, permettant d'améliorer l'efficacité des motifs de détection, tout en considérant des tailles de motifs s relativement restreintes ou des sous-ensembles de comportements de taille réduite et tout en offrant une sécurité plus importante ;
- l'étude et l'exploration d'objets combinatoires plus intéressants [12,27], et offrant des propriétés encore meilleures que celles mises en évidence dans le schéma sécurisé proposé dans ce chapitre (*pairwise designs*, designs presque résolvables...) ;
- l'utilisation de schémas de partage de secret ou de schéma à seuil...

Dans le cas de l'analyse comportementale, la méthodologie proposée pour évaluer cette méthode de détection a montré toute l'importance des fonctions booléennes pour modéliser la détection. Même si les résultats obtenus sont encore insuffisants pour se faire une idée précise de l'état de l'art dans ce domaine, il est d'ores et déjà possible d'affirmer, malgré les déclarations des éditeurs, que nous sommes encore loin d'une prise en compte sérieuse de la notion de détection comportementale. Des expérimentations plus nombreuses doivent encore être conduites, à la fois sur d'autres codes déjà connus et sur des codes encore inconnus des éditeurs. Cela doit concourir à affiner les modèles. Cependant, la méthodologie proposée a d'ores et déjà prouvé son efficacité.

Les travaux actuels concernent l'automatisation du polymorphisme/métamorphisme fonctionnel. Cela passe par une étude théorique préalable de la notion de comportement de programme. L'investissement théorique est lourd.

Notons que le schéma de détection sécurisé, proposé dans la section 2.6, se généralise sans difficulté à la notion de stratégie de détection. Il faut juste travailler également au niveau fonctionnel pour prendre en compte l'ensemble $\mathcal{B}_{\mathcal{M}}$.

Exercices

1. Soit la chaîne suivante, permettant de détecter le virus *W95/Mad* (exemple emprunté dans [130, pp. 445], par la méthode du décrypteur statique :

   ```
   8BEF 33C0 BF?? ???? ??03 FDB9 ??0A 0000 8A85 ???? ????
   3007 47E2 FBEB
   ```

   Donner le schéma de détection correspondant (on suppose que cette chaîne est localisée à l'offset i). Donnez ensuite la fonction de non détection correspondante.

2. Exprimer la DNF $f = p_1 \wedge p_0$ donnée dans l'exemple 2.1, en fonction des bits b_k^i et b_k^j.

3. La technique de détection par code de redondance cyclique (voir [92, pp. 363] et [90, chapitre 7]) ou CRC permet de détecter toute modification de code et donc, sous certaines conditions, de conclure à l'infection. Un algorithme k-CRC associe à toute chaîne de caractères (vue comme une suite binaire de longueur t) de longueur quelconque, une chaîne binaire de longueur constante égale à k. D'un point de vue pratique, on considère un polynôme ayant des propriétés mathématiques adéquates, $g(x)$ de degré égal à k. La suite binaire en entrée (de taille t) est représentée formellement par un polynôme à coefficients binaires, $d(x)$ de degré égal à $t-1$. La valeur de *crc* est alors représentée par un polynôme $c(x)$ de degré inférieur strictement à k. Le polynôme $c(x)$ est calculé comme le reste de la division euclidienne de $x^k.d(x)$ par $g(x)$.
   On considère $k = 16$ et $g(x) = x^1 \oplus x^2 \oplus x^{15} \oplus x^{16}$. La détection s'opère en calculant $c(x)$ sur les dix premiers octets du code. Calculez alors les

formes disjonctives normales des fonctions de détection et de non détection correspondantes (l'utilisation d'un logiciel de calcul formel de type *Maple* ou d'un programme en langage C est recommandée). Concluez.

4. Démontrez, pour le schéma sécurisé présenté dans ce chapitre, l'égalité suivante concernant la transinformation du schéma :

$$I(\mathcal{S}_{\mathcal{M},\mathcal{M}}; \overline{f_{\mathcal{M}}}, \mathcal{S}_{\mathcal{M}}) = H(\mathcal{S}^i_{\mathcal{M},\mathcal{M}}) << H(\mathcal{S}_{\mathcal{M},\mathcal{M}}) - H(\mathcal{S}_{\mathcal{M},\mathcal{M}} | f^i_{\mathcal{M}}, \mathcal{S}^i_{\mathcal{M}}).$$

Indication : utilisez ce qui a été explicité dans la note de bas depage de la section 2.3.2.

5. Considérons le spectre d'exécutable suivant (cas d'école simplifié) pour la technique de détection par analyse spectrale [38] : $(I = \texttt{NOP}, n_{\text{nop}})$. La règle de décision conduisant à suspecter une infection est alors la suivante : soit $\widehat{n_{\text{nop}}}$ la fréquence observée de l'instruction `NOP`. Si $\widehat{n_{\text{nop}}} > n_{\text{nop}}$ alors le code étudié est suspect. Montrez que cette technique de détection est modélisable par un schéma de détection dont vous donnerez la forme générale pour la fonction de détection. Quelle est la complexité mémoire de cette fonction ?

6. Considérons le $(288, 8, 1)$-RBIBD dont la construction a été proposée par R. J. R. Abel [1]. L'ensemble de points est $(\mathbb{Z}_7 \times \mathbb{Z}_{41}) \cup \{\infty\}$. La première classe parallèle est obtenue à l'aide du 8-uplet suivant

$$\{(0, 9t), (0, 32t), (1, 3t), (1, 38t), (2, t), (2, 40t), (4, 14t), (4, 27t)\}$$

pour $t = 1, 37, 16, 18, 10$ en développant modulo $(7, *)$. Il suffit alors de rajouter le 8-uplet

$$\{\infty, (0, 0), (1, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0)\}$$

pour compléter la première classe parallèle. La résolution finale est obtenue en développant cette classe modulo $(*, 41)$. Implémentez le schéma de détection sécurisé proposé à l'aide de ce RBIBD, en choissant aléatoirement, en plus du sous-motif, la classe parallèle dans laquelle est prise le sous-motif. Cette solution est-elle meilleure que celle consistant à utiliser une classe parallèle fixe ?

Chapitre 3

Modélisation statistique de la détection virale

3.1 Introduction

Nous avons vu dans le chapitre 2 comment modéliser les schémas de détection, que cette dernière fonctionne par analyse de forme ou par analyse des comportements. Cette modélisation a permis de montrer comment l'attaquant pouvait analyser un logiciel antivirus afin de déterminer comment le contourner le plus efficacement possible. Les possibilités techniques de contournement qui s'en dégagent sont certes efficaces, voire très efficaces, mais elles restent limitées à la production de variantes virales à partir de souches ou de variantes déjà identifiées. En outre, si un schéma de détection sécurisé est mis en œuvre, la marge de manœuvre de l'attaquant devient très limitée.

Mais ce n'est pas là le seul moyen permettant de produire des codes non détectés pour un attaquant. La conception directe d'une nouvelle souche se révèle le moyen le plus efficace. Cependant, elle nécessite des compétences en matière d'algorithmique virale que le simple plagiaire, heureusement, ne possède généralement pas. Autrement dit, si la menace est plus aiguë, elle est aussi limitée à un nombre d'acteurs également restreint, du moins si l'on considère la production de codes efficaces, correctement conçus et implémentés.

La capacité à produire de nouveaux codes nécessite cependant de connaître encore plus finement les mécanismes de détection des logiciels antivirus. Si le modèle défini dans le chapitre précédent permet de comprendre leur fonctionnement face à des virus connus – pour l'analyse de forme – ou utilisant des techniques connues – pour l'analyse de forme et l'analyse comportementale –, il peut devenir rapidement difficile, voire impossible, à appréhender en pratique pour des fonctions de détection $f_{\mathcal{M}}$ complexes et des motifs de détection $\mathcal{S}_{\mathcal{M}}$ soigneusement choisis (taille s et répartition des octets dans le code). La complexité des problèmes théoriques sous-jacents peut très vite jouer en défaveur du

pirate. En outre, dans le cadre de la conception de codes originaux, le nombre de paramètres à considérer simultanément devient tel, qu'à moins de compétences très fortes et de ressources de calcul importantes, l'objectif devient impossible à atteindre. Enfin, des techniques comme l'analyse spectrale [38], dans lesquelles une part de quantitatif entre en jeu, à côté de propriétés purement structurelles, rendent le modèle présenté dans le chapitre 2 inutilisable en pratique.

Une autre approche consiste alors à prendre de la hauteur pour décrire le mode de fonctionnement des antivirus et plus largement des logiciels de sécurité associés ou assimilés (systèmes de détection d'intrusions, sondes réseaux, pots de miel...). Il est intéressant de considérer un autre modèle pour décrire les techniques de détection mises en œuvre par ces produits, pour comprendre pourquoi, comment et quand ces derniers sont faillibles. Ce modèle peut permettre alors au programmeur de codes malveillants de cerner les limites inhérentes aux techniques de détection pour mieux savoir comment les exploiter. L'autre intérêt, pour la défense cette fois, réside dans la possibilité d'expliquer et d'illustrer d'une manière plus « appliquée » les résultats théoriques [38, chapitre 3] concernant le problème général de la détection virale : l'indécidabilité de la détection. Cela permet également d'expliquer rigoureusement des problèmes comme celui des fausses alarmes.

Cet autre modèle est de nature statistique. Plutôt que de décrire les mécanismes antiviraux de manière structurelle, comme nous l'avons fait dans le chapitre 2, nous allons considérer une approche certes moins précise et plus globale mais néanmoins très efficace.

D'une certaine manière, là où le chapitre précédent considérait des populations (constituées de tous les octets d'un code, de toutes les fonctionnalités virales connues, de tous les motifs de détection répertoriés, de tous les modes de recherche imaginés...), cette nouvelle approche va considérer des échantillons de ces différentes populations. Et à un problème déterministe comme celui de l'extraction de schéma de détection va correspondre celui de l'identification des limites de la détection. Sa résolution passe par l'approche « probabiliste » et celle de l'échantillonnage.

Mais avant de pouvoir mettre en œuvre une telle approche, il est d'abord indispensable de décrire les mécanismes antiviraux selon ce modèle et de montrer que toute technique antivirale peut en fait se ramener à un test statistique. Une fois que ceci est admis et compris, l'attaquant peut « jouer » avec le modèle, à des fins de contournement. Et le moindre des aspects n'est pas la description des limites des différentes techniques antivirales, résumées dans les risques d'erreurs naturels attachés à tous tests statistiques. Au contraire, cette description montre comment il est alors possible, non plus seulement de contourner l'antivirus, mais également de se jouer de lui en exploitant les certitudes de l'utilisateur.

3.2 Les tests statistiques

Nous allons rappeler dans cette section ce que l'on appelle un test statistique, comment il fonctionne et quels en sont les paramètres essentiels. Nous nous limiterons au strict nécessaire afin de ne pas alourdir inutilement le propos. Le lecteur intéressé par un exposé complet de la théorie des statistiques pourra consulter [13, 32, 82]. Nous supposerons connues du lecteur les notions de variables aléatoires, de loi de probabilité et de fonction de répartition. La connaissance des lois de probabilité usuelles (discrètes et continues) est également supposée acquise.

Les tests statistiques permettent de prendre des décisions, de faire des choix entre plusieurs alternatives. Ils représentent une partie importante de la statistique dite inférentielle qui vise à généraliser une information obtenue à partir d'un ou plusieurs échantillons, tirés de l'univers à étudier. Cette généralisation a pour but d'inférer sur la base de ces échantillons une propriété concernant la loi de l'univers considéré. Les tests permettent également – ce qui correspond plus au cas de la détection virale – de décider lequel, parmi plusieurs univers, est à considérer.

3.2.1 Le cadre d'étude

On considère une variable aléatoire X décrivant un phénomène aléatoire donné soumis à l'analyse. Le but est d'estimer les principales caractérisques de la loi de probabilité $\mathcal{P}$, gouvernant X (sa valeur moyenne, sa fonction de densité...). Pour ce faire, n expériences sont conduites qui, à chaque fois, donnent aux valeurs $X_1, X_2, \ldots, X_n$ les valeurs observées $x_1, x_2, \ldots, x_n$. Il s'agit de réaliser un échantillon d'étude.

Pour estimer une caractéristique donnée de X sur la base de la réalisation de cet échantillon, on utilise un *estimateur*. Il s'agit d'une fonction d'échantillon, autrement dit d'une statistique calculée sur $X_1, X_2, \ldots, X_n$. C'est en fait une mesure notée $\theta_n^* = S(X_1, X_2, \ldots, X_n)$ servant à estimer un paramètre inconnu θ de la loi $\mathcal{P}$. Par exemple, l'estimateur $\overline{X} = \frac{X_1+X_2+\ldots+X_n}{n}$ sera utilisé pour estimer la taille moyenne théorique μ de la population.

Deux classes de problèmes peuvent alors être traitées :

- *l'estimation de paramètres inconnus.* À partir de l'échantillon $(X_1, X_2, \ldots, X_n)$, il faut estimer θ, paramètre de la loi $\mathcal{P}$. La plupart du temps, l'ensemble Θ des valeurs possibles de θ est spécifié. La taille des individus d'une population, par exemple, est comprise dans l'intervalle $[1, 2]$ (en mètres). En outre, le plus souvent, on connaît a priori une information sur la loi $\mathcal{P}$ qui permet de la rattacher à une famille définie de lois.
- *le test des hypothèses statistiques.* Il s'agit là de tester une ou plusieurs hypothèses concernant une loi inconnue $\mathcal{P}$. Le plus souvent, il s'agit de déterminer si $\mathcal{P}$ a une forme donnée plutôt qu'une autre.

La théorie statistique permet de démontrer que ces deux classes sont de nature très proche et que les problèmes de chacune d'elles ne diffèrent pas radicalement.

Nous considérerons le cadre paramétrique[1]. La loi observée est supposée appartenir à une famille de lois décrites par des densités de probabilité $\{f(x,\theta);\theta \in \Theta\}$ qui ne dépendent que d'un seul paramètre inconnu θ. L'ensemble Θ est appelé espace paramétrique. Il est, dans le cas général, inclus dans $\mathbb{R}^k$ où k est la dimension de θ (pour une moyenne, par exemple, $k = 1$ alors que $k = 2$ pour une variance). Dans le cas de la détection virale, nous nous restreindrons à l'ensemble $\mathbb{N}^k$. La forme de la fonction de densité est en général connue.

3.2.2 Définition d'un test statistique

L'objectif est de prendre une décision concernant le rejet ou l'acceptation d'une hypothèse $\mathcal{H}$. Cette décision doit considérer uniquement l'échantillon $(X_1, X_2, \ldots, X_n)$ et une information concernant la forme de la loi selon laquelle X est distribuée (X désigne la variable aléatoire de la loi mère de l'échantillon). Dans le cas général, nous pouvons avoir à choisir entre r hypothèses $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_r$. En fait, pour chaque $\mathcal{H}_i$, on suppose que la variable X est distribuée selon une loi $\mathcal{P}_i$. Sans perte de généralités, nous nous limiterons au cas $r = 2$. Le lecteur intéressé par le cas général pourra utilement se référer à [13, chapitre 3].

Un test statistique consiste à décider ou à rejeter une hypothèse selon laquelle θ appartient à un ensemble de valeurs Θ_0. Cette hypothèse de référence est appelée traditionnellement *hypothèse nulle*, notée $\mathcal{H}_0$. Cette hypothèse correspond à la valeur présumée du paramètre θ. On définit également une (ou plusieurs) *hypothèse alternative*, notée $\mathcal{H}_1$ et selon laquelle il est supposé que $\theta \in \Theta_1 = \Theta - \Theta_0$. En résumé, un test consiste à opposer deux hypothèses

$$\mathcal{H}_0 : \theta \in \Theta_0 \qquad \text{contre} \qquad \mathcal{H}_1 : \theta \in \Theta_1.$$

Il existe alors plusieurs types de tests que l'on classe en fonction de la nature des ensembles Θ_0 et Θ_1. Trois cas se présentent :

- $\mathcal{H}_0$ et $\mathcal{H}_1$ sont toutes deux simples et nous avons $\Theta = \{\theta_0, \theta_1\}$:

$$\mathcal{H}_0 : \theta = \theta_0 \qquad \text{contre} \qquad \mathcal{H}_1 : \theta = \theta_1.$$

- $\mathcal{H}_0$ est simple et $\mathcal{H}_1$ est multiple ($|\Theta| > 2$) :

$$\mathcal{H}_0 : \theta = \theta_0 \qquad \text{contre} \qquad \mathcal{H}_1 : \theta \neq \theta_0.$$

- $\mathcal{H}_0$ et $\mathcal{H}_1$ sont toutes deux multiples ($|\Theta| > 2$) :

$$\mathcal{H}_0 : \theta \in \Theta_0 \qquad \text{contre} \qquad \mathcal{H}_1 : \theta \in \Theta_1.$$

Nous nous limiterons au premier cas. Il permet de décrire simplement ce qu'est un test et les enjeux qui en découlent. En outre, il est possible de décrire les

[1] Il existe un autre cadre dit *non paramétrique* dans lequel la loi étudiée ne fait plus partie d'une famille paramétrable de lois. En d'autres termes, θ_n^* converge faiblement vers une loi $\mathcal{P}$ lorsque $n \to \infty$ et $\mathcal{P}$ ne dépend pas de la distribution de la variable X.

autres cas par des tests d'hypothèses simples (avec $r > 2$ en posant $\mathcal{H}_i$ pour $\theta = \theta_i \in \Theta$). La différence tient au fait que l'étude des différents risques d'erreur est plus délicate, mais conceptuellement, les choses s'expliquent de manière similaire (voir [82, pp. 213 et suiv] et [13, pp. 310 et suiv.]). Nous n'aborderons pas non plus les tests pour variables catégorielles ou d'ajustement à une loi [82, chapitre 10]. Leur principe, d'un point de vue conceptuel et méthodologique, n'est pas très éloigné de celui de la famille de tests présentées ici. Pour ce qui nous intéresse, la notion de type d'erreur est la même. Nous présenterons et mettrons en œuvre un tel test dans la section 3.6.5.

Le principal outil utilisé pour un construire un test est un *estimateur* que nous noterons E. Cet estimateur est également dénommé *statistique de test.* Nous noterons e la valeur réalisée sur un échantillon pour l'estimateur E. Il s'agit d'une statistique (mesure) faite sur les individus d'un échantillon. Selon l'une ou l'autre des hypothèses du test, cet estimateur suit une loi de probabilité différente (loi selon $\mathcal{H}_0$ ou loi selon $\mathcal{H}_1$). Un test d'hypothèses revient donc à décider, en vertu des valeurs d'un estimateur sur un ou plusieurs échantillons, quelle est la loi gouvernant la population que l'on étudie. Pour plus de facilité, nous considérerons le cas général où $E \in \mathbb{R}$.

Construction d'un test d'hypothèses Voyons les différentes étapes présidant à la construction d'un test. Nous étudierons ensuite les propriétés et caractéristiques (essentiellement les différents risques d'erreur). Pour construire un test, les étapes sont les suivantes :

1. **Formulation des hypothèses**. Elles sont définies par rapport à l'estimateur E donné (variable aléatoire d'un échantillon à un autre) censé être représentatif, dans son comportement moyen, d'un paramètre théorique d'une population. Le choix de l'hypothèse nulle doit être fait avec soin. En règle générale, elle correspond à l'hypothèse selon laquelle on est conduit à conserver la valeur présumée θ_0 du paramètre étudié pour la population. L'hypothèse nulle est généralement celle que l'on ne rejette qu'à contrecœur. Ainsi, dans le cadre de la détection antivirale, c'est l'hypothèse selon laquelle un fichier analysé n'est pas infecté.
2. **Choix du seuil de signification du test**. Noté α, il correspond au risque de rejeter $\mathcal{H}_0$ alors que cette hypothèse est vraie. Le choix de ce seuil est assez délicat[2]. C'est également le choix de ce seuil qui permet (éventuellement) de jouer sur le résultat du test. Nous verrons un peu plus loin le problème des erreurs attachées à un test. Choisissons ce seuil égal à 1 %.
3. **Déterminer la loi de probabilité**. Il faut établir la nature de cette loi correspondant à la distribution d'échantillonnage pour chaque hypo-

[2] D'une manière générale, ce risque est déterminé *a priori* par les conséquences d'une éventuelle erreur de décision. Ces conséquences, par exemple, ne sont pas les mêmes selon qu'il s'agit d'évaluer les effets secondaires sur l'organisme d'une substance susceptible d'être commercialisée ou bien d'estimer quel choix des électeurs vont faire lors d'une consultation politique locale.

thèse. En d'autres termes, quel est le modèle mathématique décrivant le comportement moyen de l'estimateur E choisi, d'un échantillon à l'autre, selon que l'hypothèse $\mathcal{H}_0$ est valide ou non ? Il est essentiel de noter que si l'on est toujours en mesure de le faire pour l'hypothèse nulle, ce n'est pas le cas pour $\mathcal{H}_1$, dont la loi est assez souvent inconnue. Dans le domaine de la détection virale, nous pouvons considérer que c'est quasi-systématiquement le cas, dans la mesure où l'hypothèse $\mathcal{H}_1$ concerne des codes malveillants non encore connus (détectés). Tous les cas sont donc possibles.

4. **Calcul du seuil de décision**. Ce seuil S, selon la valeur de l'estimateur E, permet de choisir laquelle des hypothèses est décidée valide. Le calcul de S est directement déterminé par le niveau d'erreur que l'on consent *a priori* (voir plus loin). Notons que ce seuil partage l'ensemble des valeurs possibles pour E en deux parties de $\mathbb{R}$ que nous noterons A (région d'acceptation) et $\overline{A} = \mathbb{R} \backslash A$ (région de rejet). Quand la loi de l'hypothèse $\mathcal{H}_1$ est connue, le calcul de ce seuil S peut intégrer les deux types d'erreur (voir plus loin sur l'exemple détaillé).
5. **Tester l'hypothèse**. Autrement dit, on calcule sur l'échantillon la valeur observée e de l'estimateur E et la décision est alors établie de la manière suivante :
   (i) si $e < S$ (autrement dit $e \in A$), alors on décide de conserver l'hypothèse nulle (en fait on ne la rejette pas, ce qui signifie simplement que l'on ne dispose pas d'assez de données pour la rejeter ; pour cela il faudrait explorer totalement toute la population). En d'autres termes, les variations observées par rapport à la théorie s'expliquent par les variations d'échantillonnage ;
   (ii) sinon ($e > S$) on décide que c'est l'hypothèse alternative qui vraisemblablement s'applique. Dans ce cas, il est estimé que les variations observées par rapport à la théorie sont trop importantes pour être dues aux seules variations d'échantillonnage.

Types d'erreur et puissance du test Toute décision étant fondée sur des résultats aléatoires (les échantillons), deux erreurs différentes sont possibles et peuvent être évaluées par leur probabilité respective, dite *probabilité d'erreur* ou *risque*. Le tableau 3.1 résume les deux types d'erreur possible, attachés à tout processus de décision.

Décision	$\mathcal{H}_0$ vraie	$\mathcal{H}_1$ vraie
Accepter $\mathcal{H}_0$	$1-\alpha$	β
Rejeter $\mathcal{H}_0$	α	$1-\beta$

Table 3.1 – Probabilités des deux types d'erreur

Explicitons ces deux probabilités d'erreur avec les définitions suivantes.

Définition 3.1 *(Risque de première espèce) On appelle* risque de première espèce, *la valeur* α *telle que*[3]

$$\alpha = P[E \in \overline{A} | \mathcal{H}_0 \text{ vraie}] = P[e > S | \mathcal{H}_0 \text{ vraie}],$$

c'est-à-dire la probabilité de rejeter l'hypothèse $\mathcal{H}_0$ *alors qu'elle est vraie.*

Dans le contexte de la lutte antivirale, si l'hypothèse nulle est l'hypothèse selon laquelle il n'y a pas d'infection, le risque de première espèce correspond à la probabilité de fausse alarme.

La notion de fausse alarme que nous avions déjà évoquée dans la section 2.3.2 du chapitre 2 est donc définie maintenant rigoureusement. Le problème tient encore une fois à l'évaluation des paramètres régissant la loi gouvernant l'hypothèse nulle. Il existe en règle générale une différence très importante entre sa valeur théorique (donnée par l'analyse d'un grand nombre de fichiers, par exemple) et sa valeur réelle (au niveau de la population), laquelle dépend elle-même fortement du format (exécutable, image, son...). À titre d'illustration sur la difficulté de son évaluation et sur les « effets de bord » que cela peut engendrer, en mars 2006, le logiciel antivirus *McAfee* détectait par erreur comme infecté plusieurs applications légitimes comme le tableur *Excel* de Microsoft en les identifiant comme un virus dénommé *W95/CTX* [20]. Tous les antivirus, à des degrés divers, ont connu ce genre de désagréments.

Définition 3.2 *(Risque de seconde espèce) On appelle* risque de seconde espèce *la valeur* β *définie par*

$$\beta = P[E \in A | \mathcal{H}_1 \text{ vraie}] = P[e < S | \mathcal{H}_1 \text{ vraie}],$$

c'est-à-dire la probabilité d'accepter l'hypothèse $\mathcal{H}_0$ *alors que c'est l'hypothèse* $\mathcal{H}_1$ *qui est valide.*

Le risque de seconde espèce correspond donc à la probabilité de non détection, dans le cas où l'hypothèse nulle est l'hypothèse de non infection.

Ces deux risques d'erreur sont illustrés par la figure 3.1. Il est essentiel (voir exemple détaillé dans la section 3.3) de noter que ces deux risques sont interdépendants et s'opposent. En effet, ils reposent chacun sur un ensemble, A et $\overline{A}$, lesquels sont complémentaires. Si on accroît la taille de A, alors celle de $\overline{A}$ décroît et inversement. En pratique, dans la plupart des cas, c'est le risque α qui est privilégié. Le test est bâti de manière à construire une partition de $\mathbb{R}$ en deux ensembles A et $\overline{A}$ à l'aide d'un seuil S, de telle sorte que, pour α fixé, la probabilité $P[e > S | \mathcal{H}_0 \text{ vraie}] = \alpha$. Il est donc impératif de connaître au minimum la loi sous l'hypothèse nulle sinon le test ne pourrait être construit. Si la loi sous l'hypothèse alternative est inconnue, il n'est alors pas possible de maîtriser le risque de seconde espèce β.

[3] La notation $P[E \in \overline{A} | \mathcal{H}_0 \text{ vraie}]$ ne désigne pas une probabilité conditionnelle. Il s'agit juste d'une notation synthétique commode.

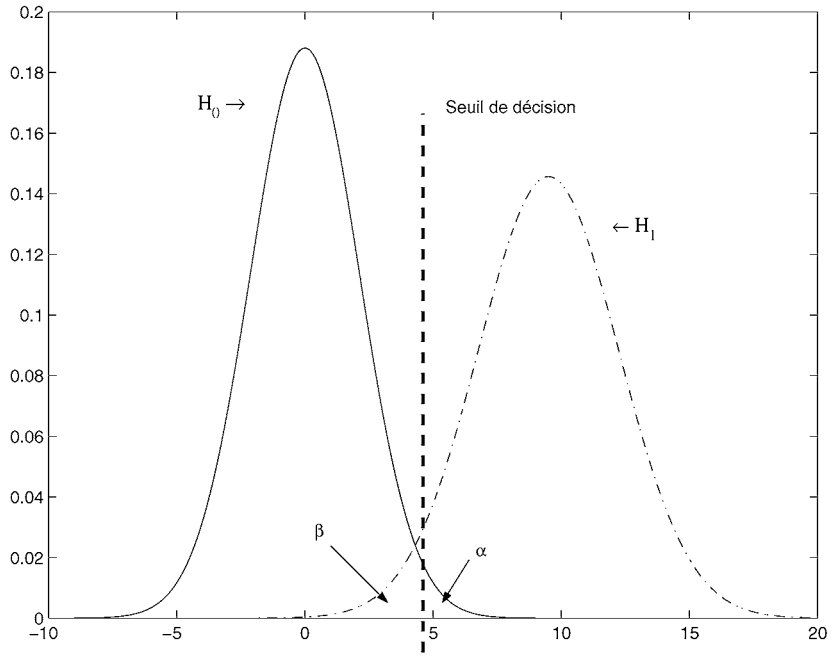


Figure 3.1 – Modélisation statistique de la détection virale

Enfin, il est important de préciser que les risques α et β ne sont par définition jamais nuls, excepté lorsque les domaines de définition des lois (fonctions de densité) gouvernant respectivement $\mathcal{H}_0$ et $\mathcal{H}_1$ sont disjoints[4]. Cela en outre conduirait à considérer un seuil S infini.

Nous ne présenterons pas les notions de puissance d'un test et de test sans biais. Elles n'ont d'intérêt que pour le choix des tests à mettre en œuvre. Dans notre approche, nous avons supposé que les tests $\mathcal{T}_i$ étaient tous les tests les plus puissants.

Seule compte, pour notre modélisation, l'existence des risques α_i et β_i. Le lecteur intéressé par les concepts de puissance d'un test pourra consulter utilement [82, pp. 204-214].

3.3 Modélisation statistique de la détection antivirale

Maintenant que nous avons présenté le principe du test statistique, nous allons expliquer comment modéliser simplement la détection antivirale à l'aide

[4] De manière évidente, s'ils sont disjoints, un test statistique n'est alors pas nécessaire.

de tels tests. Nous montrerons ensuite, dans la section 3.4, que toutes les techniques de détection connues peuvent être décrites par un test statistique.

3.3.1 Définition du modèle

Un détecteur antiviral D va conduire un ou éventuellement plusieurs tests pour décider si un fichier F soumis à l'analyse est infecté ou non. Assez souvent, un seul test est conduit, consistant à rechercher une signature présente dans la base de signatures. Ce test peut être modélisé par un véritable test statistique classique (voir section 3.4). Ce dernier sera mis systématiquement en défaut par tout virus inconnu (non présent dans la base). L'évolution des techniques virales et leur compréhension, au fur et à mesure que ces dernières sont identifiées et étudiées, nécessitent de considérer des tests plus évolués et d'en appliquer plusieurs lors de chaque détection. Nous allons tout d'abord présenter le modèle général de la détection antivirale. Nous illustrerons ensuite, à l'aide de deux exemples génériques, la notion de test statistique antiviral, présentée dans la section précédente.

Supposons que le détecteur D conduise n tests de détection, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$. Nous considérerons le cas où ils sont appliqués en série, chacun d'entre eux s'exerçant sur le résultat du précédent. Cette approche a pour objectif de réduire peu à peu les probabilités d'erreur. Cela correspond au cas où les événements « rester indétecté » (alors que le fichier est infecté) et « provoquer une fausse alarme » sont réalisés conjointement pour chacun des n tests.

Ces tests sont, par définition, tous entâchés de deux risques α_i et β_i. Dans ce qui suit, nous nous limiterons à la probabilité de non détection β_i. Il est important de rappeler que d'une manière générale les risques β_i sont impossibles à maîtriser, à moins de connaître la loi de $\mathcal{H}_1$. C'est là, la raison pour laquelle les antivirus privilégient la probabilité de fausse alarme (le risque α) au détriment de la probabilité de non détection (le risque β). Il n'est en effet pas possible de maîtriser cette dernière. Le parti pris généralement adopté consiste à minimiser le risque de fausse alarme. Dans ce qui suit, les valeurs β_i sont généralement inconnues. Le point important pour nous est de savoir que, pour tout test i, nous avons $\beta_i \neq 0$.

Sans perte de généralité, nous supposerons que les tests $\mathcal{T}_i$ sont indépendants, autrement dit que le résultat de tout test $\mathcal{T}_i$ n'influe par sur le résultat de tout autre test $\mathcal{T}_j$ pour $i \neq j$, ni n'en dépend. Nous pouvons alors donner la proposition suivante.

Proposition 3.1 *Soient n tests indépendants $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$, appliqués en série, chacun ayant une probabilité d'erreur de seconde espèce β_i. Le détecteur D aura une probabilité de non détection résiduelle égale à*

$$\beta = \prod_{i=1}^{n} \beta_i.$$

Preuve.
La démonstration, sous l'hypothèse d'indépendance des test $\mathcal{T}_i$ est assez simple. Il suffit de le démontrer pour $n = 2$. Le test résultant $\mathcal{T}_{1,2}$ (des tests $\mathcal{T}_1$ et $\mathcal{T}_2$) sera ensuite considéré avec le test $\mathcal{T}_3$, et ainsi de suite.

Notons $\mathcal{H}_0^i$ et $\mathcal{H}_1^i$ les hypothèses respectivement nulle et alternative du test $\mathcal{T}_i$. La probabilité β est définie de la manière suivante :

$$\beta = P[<< \text{ non détection } >> | \mathcal{H}_1^1 \text{ et } \mathcal{H}_1^2 \text{ vraie }].$$

Comme les tests $\mathcal{T}_1$ et $\mathcal{T}_2$ sont indépendants, les hypothèses alternatives correspondantes le sont aussi. On a donc

$$\beta = P[<< \text{ non détection } >> | \mathcal{H}_1^1 \text{ vraie}] \times P[<< \text{ non détection } >> | \mathcal{H}_1^2 \text{ vraie}].$$

et donc

$$\beta = \beta_1 \times \beta_2.$$

■

Ce résultat montre d'une manière assez simple que si l'on peut faire tendre la probabilité de non détection vers 0, cette dernière ne sera jamais nulle. C'est en quelque sorte une « version statistique » de la preuve de l'indécidabilité de la détection virale. Si une série de tests suffisamment grande (éventuellement $n \to \infty$) produisait une probabilité de non détection nulle, on aboutirait à la conclusion contradictoire que le problème de la détection est décidable ; d'autre part le modèle montre que la probabilité de fausse alarme serait de 1, comme l'illustre la figure 3.1. Autrement dit, on détecterait systématiquement tout fichier comme infecté... ce qui est une méthode peu « pertinente » de résolution de la détection virale.

Remarque. Dans le cas général, les tests peuvent ne pas être indépendants – c'est *a priori* assez souvent le cas –, la probabilité d'erreur β_2 dépendra de β_1. Il sera alors nécessaire de considérer des probabilités conditionnelles. La formule générale qui suit doit alors s'appliquer, en notant A_i l'événement « erreur de non détection pour le test i ».

$$P[A_1 A_2 A_3 \dots A_n] = P[A_1] P[A_2|A_1] P[A_3|A_1 A_2] \dots P[A_n|A_1 A_2 \dots A_{n-1}]$$

Cette situation décrit en particulier le cas de l'utilisation de plusieurs antivirus, lesquels utilisent rarement des tests indépendants (voir les résultats présentés dans le chapitre 2). Plus la dépendance des tests est forte, moins vite la probabilité d'erreur résiduelle β converge vers 0.

En ce qui concerne la probabilité de fausse alarme (risque α), nous avons le résultat suivant.

Proposition 3.2 *Soient n tests indépendants $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n$, appliqués en série, chacun ayant une probabilité d'erreur de première espèce α_i. Le détecteur D aura une probabilité de fausse alarme résiduelle égale à*

$$\alpha = \prod_{i=1}^{n} \alpha_i.$$

Preuve.
Démontrons-le pour deux tests. Il suffira ensuite d'itérer la démonstration. Le résultat du test $\mathcal{T}_1$ détecte parmi N fichiers, en moyenne $\alpha_1 \times N$ d'entre eux, comme infectés alors qu'il ne le sont pas. Sur ces $\alpha_1 \times N$ fichiers, le test $\mathcal{T}_2$ est ensuite appliqué. Par définition, en moyenne $\alpha_1 \times N \times \alpha_2$ fichiers sont détectés par erreur. Soit au final, une probabilité de fausse alarme résiduelle de $\alpha = \alpha_1 \times \alpha_2$. ■

Remarque. Si les tests sont appliqués en parallèle, c'est-à-dire que chaque test est appliqué sur la totalité des données à chaque fois et non sur les données produites par un test précédent, et qu'une stratégie de détection différente de la réalisation conjointe d'événements est appliquée pour la décision finale, les probabilités résiduelles de fausse alarme et de non détection sont tout à fait différentes. Elles dépendront de la stratégie choisie. En fait, d'une certaine manière, la stratégie choisie revient à considérer une fonction booléenne dont les variables valent 0 (en cas d'échec) et 1 en cas de succès. Un exemple d'une telle stratégie sera proposée en exercice.

Nous allons maintenant illustrer d'une manière générique la mise en œuvre de tests de base, à travers deux cas très fréquents dans le domaine de la détection antivirale. Le premier cas concerne la détection de codes ou de techniques virales connues. L'analyse de ces dernières permet de connaître la loi alternative. Le second cas concerne plutôt la détection de codes inconnus, mettant en œuvre des techniques virales classiques.

3.3.2 Modèle de détection avec loi alternative connue

C'est une situation relativement courante pour les méthodes de détection de code connus (voir section 3.4). Dans ce cadre précis, la connaissance de la loi alternative permet de contrôler le risque de seconde espèce. La connaissance de la loi alternative résulte de l'analyse de codes connus appartenant à une même famille ou mettant en œuvre les mêmes techniques. Elle résulte donc de la connaissance *a posteriori* acquise par l'analyste. Ce type de test est illustré par la figure 3.1.

Nous allons considérer un cas didactique assez trivial (conçu pour illustrer simplement les choses) qui consiste à détecter une technique triviale de polymorphisme, totalement inefficace de nos jours lorsqu'elle est utilisée seule. Dans cette technique, chaque instruction du type `XOR Reg, Reg` est transformée en instruction du type `MOV Reg, 0`. Cette règle de réécriture est utilisée, conjointement avec beaucoup d'autres, par le virus `Win32/MetaPHOR` [132] (voir section 3.6.5).

L'échantillon sera formé des N instructions assembleur de base du code à analyser. L'estimateur E mesure la fréquence de l'instruction assembleur `MOV Reg, 0`. Formulons les hypothèses nulles et alternatives[5].

[5] Précisons que les valeurs données ici sont littérales et n'ont pour seul but que d'illustrer le principe du test, et non pas de donner un test « clef en main ». La détermination des valeurs

Dans le cas d'un fichier sain, on suppose que l'instruction `MOV Reg, 0` a une probabilité d'apparition p_0. Donc, sur N instructions, elle possède une moyenne d'apparition[6] $\mu = \mu_0 = N \times p_0$ (le paramètre θ testé ici n'est autre que la fréquence moyenne d'apparition μ). En revanche, l'expérience a montré que, pour les fichiers infectés par cette technique, cette instruction a une probabilité d'apparition p_1 et $\mu = \mu_1 = N \times p_1$. Par définition, nous avons $\mu_0 < \mu_1$. En outre, afin de simplifier le propos, nous supposerons connu également l'écart type de la population, dans chaque situation (σ_0 et σ_1)[7]. Les hypothèses à tester sont donc

$$\mathcal{H}_0 : \mu = \mu_0 \qquad \text{contre} \qquad \mathcal{H}_1 : \mu = \mu_1.$$

Nous choisirons un risque de première espèce $\alpha = 0,05$. Cela signifie que 5 % des fichiers testés seront indûment détectés comme infectés alors qu'ils ne le sont pas. Comme nous connaissons la loi alternative, nous pouvons maîtriser le risque β (non détection). Nous décidons que $\beta = 0,01$. Cela signifie que nous ne voulons pas laisser passer plus de 1 % des fichiers infectés, en moyenne.

Pour chacune des hypothèses, nous utiliserons la loi normale comme approximation de la distribution du paramètre μ. En effet, la taille de l'échantillon est suffisamment importante ($N > 30$) et les valeurs d'écart type sont connues[8]. Donc, pour l'hypothèse $\mathcal{H}_0$, nous supposons que l'effectif de l'instruction `MOV Reg, 0` suit une loi normale $\mathcal{N}(\mu_0, \sigma_0)$ et sous l'hypothèse alternative $\mathcal{H}_1$, il suit la loi normale $\mathcal{N}(\mu_1, \sigma_1)$.

Le calcul du seuil de décision S est un peu plus délicat que dans le cas général car il faut tenir compte simultanément des deux types d'erreur. Exprimons chacun d'entre eux mathématiquement. Pour cela, nous considérerons les variables centrées et réduites $\frac{E-\mu_i}{\sigma_i}$. Notons $q_i = 1 - p_i$. Sous l'hypothèse $\mathcal{H}_0$, nous avons

$$\alpha = P[e > S | \mathcal{H}_0 \text{ vraie}].$$

Après avoir centré et réduit par les paramètres de la loi sous l'hypothèse nulle, nous avons

$$\alpha = 1 - \frac{1}{\sqrt{2\pi}} \int_{\infty}^{\frac{S-Np_0}{\sqrt{Np_0q_0}}} \exp(-\frac{x^2}{2}) = 1 - \Phi\left(\frac{S - Np_0}{\sqrt{Np_0q_0}}\right),$$

réelles des paramètres pour chaque hypothèse résulte d'une analyse statistique d'un grand nombre de fichiers sains (hypothèse nulle) ou infectés par la ou les techniques virales que l'on veut précisément détecter (hypothèse alternative).

[6] Plus précisément, on considère chaque instruction comme la réalisation d'une variable de Bernoulli, pour l'événement « est du type `MOV Reg, 0` », de paramètre p_0. On supposera, en première approximation et pour la simplicité du propos, que les épreuves sont mutuellement indépendantes. Cela n'est pas tout à fait le cas mais cette supposition permet malgré tout d'obtenir des résultats intéressants pour la détection. On considère ensuite la somme E d'une série de N épreuves de Bernoulli. Les résultats fondamentaux de la théorie des probabilités permettent alors de considérer la loi normale comme loi d'approximation pour E.

[7] Dans le cas d'une somme de série d'épreuves de Bernoulli, nous avons $\sigma_i = \sqrt{N \times p_0 \times (1 - p_0)}$

[8] Dans le cas contraire, lorsque l'écart type σ est inconnu et que la taille de l'échantillon est trop petite ($N < 30$), on calcule l'écart type de l'échantillon et on utilise la distribution t de Student.

où $\Phi(.)$ désigne la fonction de répartition de la loi normale centrée réduite. Une lecture inversée de la table correspondant à cette loi, donne une valeur a (la valeur α étant fixée) telle que

$$a = \frac{S - Np_0}{\sqrt{Np_0q_0}}. \tag{3.1}$$

Sous l'hypothèse $\mathcal{H}_1$, nous avons, de la même manière,

$$\beta = P[e < S|\mathcal{H}_1 \text{ vraie}].$$

Après avoir centré et réduit par les paramètres de la loi sous l'hypothèse alternative, nous pouvons écrire

$$\beta = \frac{1}{\sqrt{2\pi}} \int_{\infty}^{\frac{S-Np_1}{\sqrt{Np_1q_1}}} \exp(-\frac{x^2}{2}) = \Phi\left(\frac{S - Np_1}{\sqrt{Np_1q_1}}\right),$$

Une lecture inversée de la table de la loi normale centrée réduite donne une valeur b (la probabilité β étant fixée) telle que

$$b = \frac{S - Np_1}{\sqrt{Np_1q_1}}. \tag{3.2}$$

Nous avons un système à deux équations – équations (3.1) et (3.2) – et deux inconnues, S et N.

$$\begin{cases} \frac{S-Np_0}{\sqrt{Np_0q_0}} &= a \\ \frac{S-Np_1}{\sqrt{Np_1q_1}} &= b \end{cases}$$

Nous considérons N comme une inconnue *a priori* car lorsqu'un fichier est soumis à l'analyse, on ne sait pas par avance combien d'instructions seront présentes. Or, pour que le modèle puisse fonctionner, il est nécessaire de considérer, pour les risques α et β considérés, un nombre minimal d'instructions. C'est précisément la valeur de $N_{\min}$ fournie par le système précédent. Si la taille du fichier (en nombre d'instructions) est au moins égale à cette valeur, le test est valide, sinon il faut en imaginer un autre nécessitant une taille d'échantillon moindre. L'autre solution consiste à jouer sur les probabilités α et β et ainsi diminuer la taille minimale requise pour un fichier. À l'inverse, si la taille du fichier est en moyenne plus grande que la valeur $N_{\min}$, on dispose alors de plus d'informations et les valeurs de α ou β peuvent être diminuées. Dans la pratique, si $N > N_{\min}$, c'est la valeur N qui est prise en compte dans le calcul du seuil.

La résolution du système précédent donne la valeur de $N_{\min}$:

$$N_{\min} = \left(\frac{b\sqrt{p_1q_1} - a\sqrt{p_0q_0}}{p_0 - p_1}\right)^2.$$

La valeur du seuil S est, elle, calculée à partir soit de l'équation (3.1) soit de l'équation (3.2).

si $\frac{e-\mu_0}{\sigma_0} < \frac{S-\mu_0}{\sigma_0}$, on ne peut rejeter $\mathcal{H}_0$,
le fichier est probablement sain,
sinon, on rejette $\mathcal{H}_0$,
le code est vraisemblablement infecté.

La règle de décision est alors la suivante, pour $E = e$ sur le fichier en cours d'analyse :

L'intérêt de connaître la loi gouvernant la loi alternative (cas des codes connus ou de techniques virales connues) est la complète maîtrise des risques d'erreur. Il faut cependant se rappeler que ces deux types d'erreurs s'opposent et un compromis doit donc être trouvé entre le risque de ne pas détecter un code malveillant et celui de décider par erreur qu'un fichier est infecté.

3.3.3 Modèle de détection avec loi alternative inconnue

Nous allons maintenant considérer le cas correspondant à la détection de codes inconnus ou de techniques virales inconnues. Dans ce cas précis, la seule connaissance que nous avons, pour un ou plusieurs estimateurs donnés, concerne la loi gouvernant l'hypothèse nulle, à savoir

$$\mathcal{H}_0 : \theta = \theta_0.$$

La famille de tests qui sont alors mis en œuvre est celle des tests pour lesquels l'hypothèse alternative est l'une des trois suivantes :

$$\begin{aligned} \mathcal{H}_1 &: \theta > \theta_0 \\ \mathcal{H}_1 &: \theta < \theta_0 \\ \mathcal{H}_1 &: \theta \neq \theta_0 \end{aligned}$$

Nous allons traiter le cas général correspondant à la dernière situation. Il s'agit d'un test bilatéral (puisque nous ne faisons aucune supposition sur la valeur du paramètre θ par rapport à la valeur θ_0).

Considérons un estimateur E décrivant la proportion de l'instruction assembleur `XOR Reg, Reg` dans des programmes sains. Dans ces derniers, cette instruction apparaît en moyenne μ_0 fois. Cela correspond donc à l'hypothèse nulle $\mathcal{H}_0 : \mu = \mu_0$. Sachant que beaucoup de techniques de polymorphisme/métamorphisme utilisent des techniques de réécriture, cette instruction peut voir sa fréquence d'apparition soit diminuer (si par exemple elle est remplacée par l'instruction `MOV Reg, 0`), soit augmenter (cas où des instructions de bourrage

ou de code mort sont insérées). Tout programme infecté par des codes mettant en jeu ce genre de techniques sera gouverné par l'hypothèse alternative $\mathcal{H}_1 : \mu \neq \mu_0$.

Comme dans le cas de la section précédente, nous supposerons que la loi de $\mathcal{H}_0$ est la loi normale $\mathcal{N}(\mu_0, \sigma_0)$ où σ_0 est l'écart type de la population. Il est en effet possible de se ramener à cette situation dans la très grande majorité des cas ou à une situation similaire – utilisation de l'écart type d'échantillon, utilisation de la loi de Student [32, pp. 273-274].

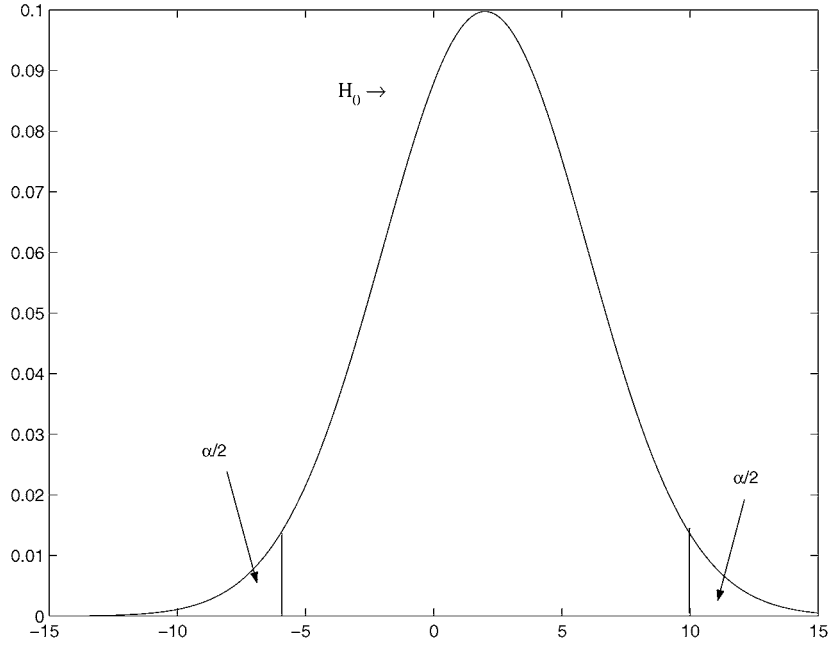


Figure 3.2 – Modélisation statistique de la détection d'un virus inconnu

Le test se construit alors de la manière suivante. On fixe un risque d'erreur $\alpha = 0,05$ (par exemple). Cela donne une région de rejet que l'on divise en deux parties égales (voir figure 3.2), chacune avec un probabilité de rejet de $\frac{\alpha}{2}$ (puisque, *a priori*, la probabilité est la même que $\mu < \mu_0$ ou que $\mu > \mu_0$). En revanche, on ne peut plus maîtriser la risque β, et donc évaluer la probabilité de non détection relative à ce test. On ne prend donc en compte que le seul risque α réparti en deux régions de rejet. Nous avons donc une valeur de seuil S telle que

$$P\left[\frac{|E-\mu_0|}{\sigma_0} > \frac{S-\mu_0}{\sigma_0}\right] = \frac{\alpha}{2}.$$

La valeur de $z_{\frac{\alpha}{2}} = \frac{S-\mu_0}{\sigma_0}$ se lit dans la table de la loi normale centrée réduite,

en fonction de $\frac{\alpha}{2}$.

Ensuite, si $E = e$ pour le code en cours d'analyse, on calcule la valeur (appelé *rapport critique*) :

$$\text{R.C.} = \frac{|e - \mu_0|}{\sigma_0}.$$

La variable E est en fait centrée et réduite relativement à la distribution de $\mathcal{H}_0$. La règle de décision est alors la suivante :

si R.C. $< z_{\frac{\alpha}{2}}$,	on ne peut rejeter $\mathcal{H}_0$, le fichier est probablement sain,
sinon,	on rejette $\mathcal{H}_0$ le code est vraisemblablement infecté.

Le signe de la valeur $e - \mu_0$ ne signifie pas forcément que $\mu < \mu_0$ ou que $\mu > \mu_0$. Il faut pour cela conduire des tests unilatéraux pour lesquels la région de rejet est en une seule partie et d'aire $\alpha = 0.05$.

Ce type de test permet donc de supposer qu'un fichier soumis à l'analyse est suspect (rejet de $\mathcal{H}_0$, c'est-à-dire que le fichier est sain relativement à l'estimateur E). C'est alors qu'une analyse systématique par désassemblage devra intervenir. L'intérêt est alors d'appliquer une série de tests pour avoir une probabilité de fausse alarme résiduelle la plus basse possible et ainsi limiter le recours à des techniques d'analyse plus lourdes. En revanche, aucune information ne permet d'évaluer la probabilité de non détection. Le modèle statistique présenté ici permet de bien montrer toute la difficulté de ce point de vue.

Précisons enfin que les résultats des tests statistiques peuvent servir à affiner la loi gouvernant l'hypothèse nulle, soit pour le test utilisé, soit pour d'autres tests, et ce dans le cadre de l'approche bayésienne. Une loi *a priori* peut ainsi, en fonction des observations, évoluer vers une loi *a posteriori*. Nous ne décrirons pas plus l'approche bayésienne en statistique. Elle dépasse le cadre de cet ouvrage. Le lecteur pourra consulter [112] pour plus de détails. Il suffit de retenir que la considération de lois *a posteriori* ne change pas la nature du problème, à savoir que les risques α et β sont non nuls.

3.4 Techniques de détection et tests statistiques

Après avoir présenté le modèle statistique général de la détection de codes malveillants, voyons comment les méthodes classiques de détection se décrivent facilement par des tests statistiques.

3.4.1 Cas de la recherche classique de signature

Dans ce cas trivial de détection, mais néanmoins encore très utilisé (voir chapitre 2), un code malveillant $\mathcal{M}$ est caractérisé par une séquence d'octets[9] non nécessairement contigus $\sigma_{\mathcal{M}}$. Nous avons donc $\sigma_{\mathcal{M}} \in \{0, 1, 2, \ldots, 255\}^s$ avec $s = |\sigma_{\mathcal{M}}|$. Il est tout aussi intéressant de représenter cette signature par l'ensemble des indices dans un fichier F où elle est localisée. C'est alors un sous-ensemble de $\mathbb{N}_n^* = \{1, 2, \ldots, n\}$.

Soit un fichier F que l'on soumet à un détecteur $\mathcal{D}$. On modélise le test en considérant le fait que les algorithmes de recherche de chaînes de caractères (algorithme naïf de parcours de tous les sous-ensembles possibles de s octets, algorithme Rabin-Karp [72], recherche par automate fini [3], algorithme de Knuth-Morris-Pratt [80], algorithme de Boyer-Moore [14]...) évaluent un grand nombre de sous-ensembles de s octets avant de détecter, éventuellement, la signature.

Cette modélisation pourra sembler quelque peu artificielle mais le but est de montrer que la recherche simple de signatures peut être décrite à l'aide d'un test statistique. Rappelons que dans beaucoup de cas, pour ne pas dire la plupart, les détecteurs recherchent en réalité une chaîne d'octets donnée à un endroit bien précis (une série de positions dans le code ; voir chapitre 2). Mais si cette signature vient à être déplacée ne serait-ce que d'un seul octet, alors le code n'est plus détecté. Il s'agit là d'une recherche des plus triviales, pour ne pas dire naïve, même si elle correspond à ce que font la plupart des produits. Dans le cas d'un véritable algorithme de recherche de chaînes de caractères, non contigus de surcroît, nous ne sommes pas très éloignés du modèle naïf, consistant à parcourir sinon tous les sous-ensembles possibles de s octets, du moins un grand nombre d'entre eux.

La recherche de la signature $\sigma_{\mathcal{M}}$ peut être décrite par la somme d'une série d'épreuves de Bernoulli. On définit la variable de Bernoulli suivante[10] relativement à un sous-ensemble $\mathcal{S}_i$ de s octets (non nécessairement contigus) du code :

$$X_{\mathcal{S}_i} = \begin{cases} \mathcal{S}_i = \sigma_{\mathcal{M}} & p_0 \\ \mathcal{S}_i \neq \sigma_{\mathcal{M}} & q_0 = 1 - p_0 \end{cases}$$

Pour l'évaluation de la probabilité p_0, le lecteur se référera à la section 2.3.2. Si le fichier F contient n octets, il y a au plus $N = \binom{n}{s}$ sous-ensembles[11] de

[9] On supposera que la signature est efficace (voir section 2.3.2). Le cas où elle ne l'est pas est laissé à titre d'exercice.

[10] Nous faisons là une approximation pratique en supposant que les épreuves de Bernoulli correspondantes sont d'une part indépendantes et d'autre part de probabilité constante. Cette approximation peut être considérée comme globalement acceptable pour notre modèle. En particulier, on ne fait *a priori* aucune supposition sur le fichier F soumis à l'analyse et qui *a priori* relève de l'hypothèse nulle. Ce dernier peut être de nature aléatoire.

[11] Ce nombre correspond en fait au nombre de sous-ensembles de s positions dans le code, ce qui est différent de considérer les séquences d'octets qui se trouvent à ces positions, lesquelles sont moins nombreuses du fait des octets répétés. Toutefois l'approximation par excès qui est faite ici ne nuit pas à la modélisation.

taille s dans F. On considère alors l'estimateur suivant :

$$Z = \sum_{\mathcal{S}_i \in \mathcal{P}(F), |\mathcal{S}_i| = s} X_{\mathcal{S}_i}.$$

Nous avons alors deux hypothèses. L'hypothèse nulle $\mathcal{H}_0$ est celle selon laquelle F n'est pas infecté. La loi qui la gouverne alors est la loi normale $\mathcal{N}(N.p_0, \sqrt{Np_0q_0})$. L'hypothèse alternative $\mathcal{H}_1$ est quant à elle simple à définir. Nous avons $\mathcal{H}_1 : \mu = 1$. En effet, si le code est infecté, dès que l'algorithme rencontre $\sigma_{\mathcal{M}}$, il s'arrête. On peut alors décrire la « loi » de $\mathcal{H}_1$ par la fonction de densité $f(1) = 1$ et $f(x) = 0$ si $x \neq 1$. Les paramètres sont alors $\mu_1 = 1$ et $\sigma_1 = 0$.

N'oublions pas que seulement deux cas sont possibles :

- le fichier est infecté par le code $\mathcal{M}$, auquel cas avec notre modèle, on a certainement $\mu = \mu_1 = 1$ (avec une probabilité égale à 1) ;
- le fichier est infecté par un code $\mathcal{M}'$ non répertorié dans la base de signatures, alors relativement à la signature $\sigma_{\mathcal{M}}$, on a $\mu = \mu_0$.

Le test est alors simple à décrire. Pour un risque α fixé, on calcule les valeurs $z_{\frac{\alpha}{2}}$ et R.C comme indiqué dans la section 3.3.3 et on applique la règle de décision. Notons que dès lors que n et s sont grands, un seuil empirique de 1 est suffisant.

Calculer la probabilité d'erreur β n'a ici pas grand sens. Si le fichier F est infecté par un code $\mathcal{M}$ présent dans la base, la probabilité $P[Z < 1 | \mathcal{H}_1 \text{ vraie }] = 0$ relativement à $\mathcal{M}$. En effet, on a $P[Z \geq 1 | \mathcal{H}_1 \text{ vraie }] = 1$ (probabilité constante). Si le code est connu, il sera systématiquement détecté. Pour évaluer la probabilité de non détection de la recherche de signature classique, il ne faut plus le faire par rapport à un code malveillant donné. Dans cette perspective, on peut empiriquement définir la probabilité de non détection de la recherche de signature classique ainsi :

$$P[\text{non-détection}] = \frac{|\mathcal{B}_{\mathcal{D}}|}{\aleph_0} = \frac{|\mathcal{B}_{\mathcal{D}}|}{\infty} \to 0,$$

où $\mathcal{B}_{\mathcal{D}}$ désigne la base de signatures du détecteur $\mathcal{D}$ et où $\aleph_0$ est le nombre total de virus possibles (voir [38, chapitre 3]).

3.4.2 Cas général des schémas ou des stratégies de détection

Dans le cas général, le modèle précédent reste applicable. La seule différence notable tient au fait que la fonction de détection a un poids supérieur à 1 (poids de la fonction de détection ET ; voir chapitre 2).

Si on note $\omega = \text{wt}(f)$, la probabilité théorique p_0 vaut $\frac{\omega}{256^s}$ au lieu de $\frac{1}{256^s}$. Comme il a été noté dans la section 2.5.2, la probabilité de fausse alarme augmente. Mais cette augmentation reste négligeable dès lors que s augmente. En revanche, le gain s'exprime en terme de rigidité.

3.4.3 Autres cas

Pour un certain nombre de techniques de détection, la fonction de détection est quasi-impossible à établir du fait de sa complexité. De fait, la connaissance de la loi gouvernant l'hypothèse nulle est impossible *a priori*. Un loi empirique est alors établie, affinée au gré des observations (approche bayésienne) et des améliorations techniques.

Les auteurs de ces techniques fournissent généralement des résultats expérimentaux pour les risques d'erreur α et β. Ainsi, à titre d'exemple, une technique de détection exotique comme celle proposée par I. Yoo et U. Ultes-Nitsche [146], annonce des valeurs $\alpha = 0,3$ et $\beta = 0,16$. Le lecteur pourra aussi consulter [96] concernant l'évaluation des antivirus pour environnements mobiles (téléphones, PDA et PocketPC). Toute nouvelle technique de détection devrait être publiée avec ces informations et les données techniques nécessaires pour reproduire le modèle.

Du point de vue de l'attaquant, pour ces tests comme pour tous les autres, l'expérimentation inverse permettra de savoir comment jouer avec les paramètres d'erreur. Mais les résultats du chapitre 2 ont montré que si souvent la fonction de détection est complexe, il n'en est pas de même de la fonction de non détection. L'attaquant sera alors en mesure de savoir comment manipuler les tests utilisés par tel ou tel antivirus. Cet aspect-là sera présenté dans la section 3.6.

3.5 Les techniques heuristiques

À côté des techniques traditionnelles de détection, d'autres techniques sont utilisées lorsque le problème lié est de nature trop complexe : il s'agit des fameuses « *heuristiques* ».

3.5.1 Définition des heuristiques

Une heuristique est en fait une stratégie ou une technique qui améliore l'efficacité d'un processus de recherche, en sacrifiant éventuellement l'exactitude ou l'optimalité de la solution. Une heuristique permet de trouver une solution satisfaisante (au regard de critères ou de contraintes prédéfinis), avec un coût acceptable, plutôt qu'une solution exacte, au prix d'un coût calculatoire exponentiel, comme c'est le cas notamment dans les problèmes d'optimisation (NP-complets).

Dans le cadre de la lutte antivirale, ces stratégies sont utilisées pour rechercher du code correspondant à des fonctions virales identifiées (ce qui exclut des codes réellement novateurs, les codes k-aires par exemples, présentés dans le chapitre 4). L'analyse heuristique relève de l'analyse de forme et considère le fichier analysé hors contexte d'exécution. Les antivirus recherchent via les heuristiques non pas des séquences fixes d'instructions spécifiques à un virus donné, mais un type d'instruction présent sous quelque forme que ce soit. Par

exemple, dans le cas d'un virus polymorphe, une suite d'instructions de lecture suivie d'une suite d'instructions d'écriture sera recherchée. Globalement, le choix d'une heuristique suppose de connaître déjà certaines propriétés statistiques sur l'ensemble des instances du problème que l'on souhaite résoudre. Cette approche statistique montre déjà les limites des heuristiques et les mêmes types d'erreur existent que pour les tests statistiques. Il serait d'ailleurs parfaitement possible de décrire les heuristiques par de tels tests.

La théorie des heuristiques est vaste et il n'est pas possible de la présenter, même succintement, ici. Il existe une très grande variété de familles de méthodes. On distingue deux grandes classes d'heuristiques :

- les heuristiques simples qui sont spécifiques d'un problème ou d'une classe de problèmes donnés [101],
- les métaheuristiques qui, elles, reposent sur des principes généraux et qui peuvent s'adapter au problème (recherche dite *tabou*, *recuit simulé*, algorithmes génétiques, algorithmes de colonies de fourmis, optimisation par essaims particulaires, algorithmes glouton...) [33].

Deux propriétés essentielles caractérisent une méthode heuristique :

- la *complétude* ; un algorithme est dit *complet* s'il débouche sur une solution lorsqu'il en existe une. Or toutes les heuristiques sont loin d'être complètes ;
- la *complexité* ; même si une heuristique bien choisie (sur un ensemble d'instances correctement modélisé d'un point de vue probabiliste) peut diminuer la complexité moyenne et éventuellement ramener le problème dans une classe inférieure de complexité, le temps de calcul peut rester suffisamment important pour interdire une utilisation opérationnelle, dans un antivirus par exemple.

D'un point de vue algorithmique, les structures de données utilisées pour mettre en œuvre des méthodes heuristiques sont le plus souvent des arbres ou des graphes (le plus souvent dirigés et valués). Selon la profondeur de l'arbre ou le diamètre du graphe considéré (distance la plus longue possible entre deux points), il est aisé de concevoir que le parcours, même efficace, d'une telle structure peut être rédhibitoire et faire échouer n'importe quelle méthode, surtout si une contrainte de temps d'exécution est impérative. Une fois que l'attaquant connaît les heuristiques utilisées, les structures de données impliquées, il lui sera facile de concevoir un code faisant échouer ces méthodes.

3.5.2 Analyse heuristique antivirale

Bien que le terme d'« heuristique » en virologie informatique ait été utilisé dès la fin des années 1990, en réalité les premières études sérieuses datent de la fin de cette décennie [119, 120]. Cependant, les quelques études existantes n'ont malheureusement pas formalisé suffisamment les choses pour exploiter tout le potentiel des techniques heuristiques classiques. Pour le moment, les heuristiques antivirales relèvent de simples techniques de détection, quelquefois très élaborées, où le cas d'espèce – un nouveau code malveillant ou une nouvelle

technique virale – est le fait inspirateur de l'heuristique. Aucune étude théorique de grande ampleur n'a permis de définir des modèles généraux à partir desquels décliner des instances particulières de stratégies de détection.

Nous allons présenter, pour les principales métaheuristiques utilisées, comment fonctionne l'analyse heuristique antivirale et quelles en sont les limites. Ces exemples sont tirés de [67,136]. Un moteur heuristique comporte un certain nombre de composants [120] :

- un désassembleur intégré ;
- un module d'émulation de code et de mémoire ;
- un analyseur statique qui réalise une première vérification du code complet ;
- un analyseur par flot qui suit le déroulement du programme ;
- un système d'association de valeur.

Considérons un arbre comme structure de données. Chaque sommet correspond à une instruction ou à un groupe d'instruction. L'association de valeurs peut se faire selon deux modes différents :

- le mode pondéré *(weight-based)* qui associe une valeur numérique appelée poids (ou valuation) à chaque séquence d'instructions rencontrée et ce en fonction de son niveau de criticité. S'il existe un chemin dont la somme des valeurs dépasse au final un certain seuil, le programme sera jugé infecté. On utilise alors des algorithmes de recherche de chemins optimaux dans un graphe[12]. La complexité des meilleurs algorithmes est en $\mathcal{O}(M.\log(N))$ où M est le nombre d'arcs et N est le nombre de sommets. Notons que si le graphe contient des circuits absorbants (chemin fermé de coût positif) le problème est alors NP-difficile [57]. Un programmeur de codes malveillants peut donc faire en sorte que de tels circuits existent pour mettre en échec ce mode ;
- le mode à base de règles *(rule-based)* qui est le plus utilisé. Des drapeaux (marqueurs) D_i sont utilisés pour représenter chaque instruction ou séquence d'instructions correspondant à une fonctionnalité répertoriée dans une base. Les règles sont alors établies soit simplement, soit en construisant un ensemble global des drapeaux recherchés – du type $\{D_1, D_2, D_3\}$ –, soit de manière plus précise, en redécoupant selon l'organisation en procédure – ensemble du type $\{\text{subroutine\_start}, D_1, D_2, \text{subroutine\_end}\}$. Le premier type de règles s'avère plus adapté à la détection de virus inconnus – mais utilisant des techniques connues – de manière générique alors que le deuxième type peut permettre d'identifier plus précisément la souche du code malveillant.

Nous nous limiterons au second cas (mode *rule-based*) dans la mesure où le premier est plus classique et ses fondements algorithmiques bien documentés [104]. Le tableau 3.2 présente des exemples de drapeaux utilisés par l'antivirus *TbScan*. On notera dans le tableau 3.2 qu'une majorité de drapeaux sont susceptibles de concerner des programmes légitimes. Pour chaque virus, on recherche

[12] La notion d'optimalité est ici inversée. La fonction de coût considère non pas un chemin de poids minimal mais de poids maximal ou supérieur à une valeur donnée.

Drapeau	Description
F	Accès suspicieux à un fichier
R	Relocation suspicieuse de code
A	Allocation mémoire suspicieuse
N	Extension de fichier fausse
S	Routine de recherche de fichiers exécutables (COM ou EXE)
#	Routine de déchiffrement
E	Point d'entrée « flottant »
L	Interception de chargement de programme
D	Accès disque en écriture (hors DOS)
M	Code résident en mémoire
!	Opcode invalide
T	Date fichier incorrecte
J	Structure de saut suspicieuse
?	En-tête EXE inconsistante
G	Instructions inutiles (code mort)
U	Appel à interruption DOS non documenté
Z	Vérification du statut exécutable (EXE/COM) d'un autre fichier
O	Écrasement ou déplacement d'un fichier en exécution
B	Retour au point d'entrée
K	Pile de programme inhabituelle ou suspicieuse

Table 3.2 – Drapeaux utilisés par l'antivirus TbScan (mode *rule-based*)

alors une séquence caractéristique de drapeaux. Le tableau 3.3 donne quelques exemples de telles séquences pour quelques virus.

Sur le graphe décrivant le flot d'exécution, ces règles permettent de modéliser un arbre de recherche. Différentes stratégies peuvent alors être considérées pour les parcourir. Nous allons présenter les trois principales métaheuristiques utilisées. Rappelons que l'intérêt des métaheuristiques, contrairement aux heuristiques simples, tient au fait qu'elles reposent sur des principes généraux, lesquels peuvent s'adapter à un grand nombre de problèmes. Selon la métaheuristique choisie, les résultats peuvent être très différents. Mais les risques α et β, pour chacune d'elles, sont non nuls.

Algorithme glouton

Il s'agit d'une métaheuristique simple et d'usage général[13], connue également sous le nom de méthode du plus proche voisin. Le principe est qu'à chaque étape durant le processus de recherche, l'option localement optimale est choisie, et ce jusqu'à trouver une solution, optimale ou non. Le processus décisionnel au

[13] De la méthode glouton sont issus plusieurs algorithmes importants de recherche de chemins optimaux comme l'algorithme de Dijkstra par exemple [104].

Virus	Motif heuristique
Jerusalem/PLO	FRLMUZ
Backfont	FRALDMUZK
Minsk_Ghost	FELDTGUZB
Murphy	FSLDMTUZO
Ninja	FEDMTUZOBK
Tolbuhin	ASEDMUOB
Yankee_Doodle	FN#ELMUZB

Table 3.3 – Motifs heuristiques de l'antivirus TbScan [136]

niveau de chaque nœud est dur : les choix faits ne sont jamais remis en cause. Aucun retour en arrière n'est possible car les étapes antérieures du parcours ne sont pas sauvegardées.

À titre d'exemple, l'antivirus TbScan sur le virus *Jérusalem* fournit le motif heuristique suivant (tableau 3.3) : FRLMUZ. Un algorithme glouton parcourt l'arbre de recherche selon le schéma de la figure 3.3. Avec la méthode glouton,

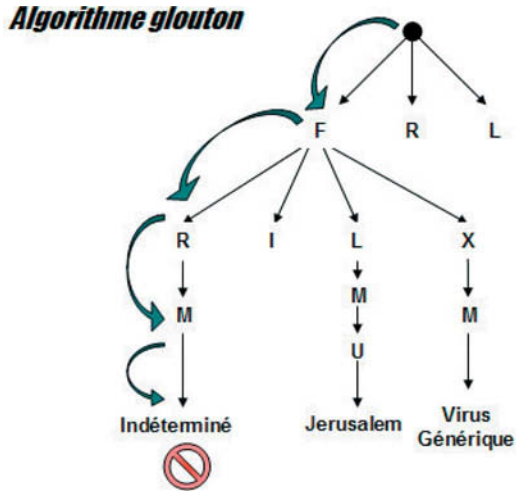


Figure 3.3 – Détection du virus *Jérusalem* par algorithme glouton (TbScan)

le virus *Jérusalem* n'est pas détecté. En effet, le drapeau R (relocation suspiscieuse) n'était pas représentatif et aurait dû être ignoré ou ce choix aurait dû pouvoir être modifé. Les algorithmes gloutons sont cependant très rapides et faciles à implémenter. Ces deux critères sont essentiels pour un antivirus.

Méthode taboue

La méthode de recherche *avec tabous* (ou encore recherche de type tabou) a été proposée par Fred Glover en 1986 [58]. C'est une recherche itérative avec recherche locale. La recherche de type tabou consiste, à partir d'une position donnée, à en explorer le voisinage et à choisir la position dans ce voisinage qui optimise la fonction objectif (ou coût). Cela peut localement conduire à dégrader la valeur de la fonction objectif mais c'est grâce à cela que l'on peut échapper à des optima locaux dont on resterait prisonnier avec d'autres méthodes (comme les méthodes gloutons).

Pour interdire la situation qui consisterait à retomber, à l'étape suivante, dans l'optimum local dont on vient de s'échapper, la recherche mémorise ces événements désormais interdits (d'où le nom de tabou) et empêche ainsi de revenir sur des positions antérieurement explorées. Ces positions sont stockées dans les structures de type FIFO (*First in First out* ou pile) que l'on appelle liste taboue. Le prix à payer est une quantité de mémoire plus importante pour stocker cette structure. La trace du cheminement est ainsi gardée en mémoire, ce qui permet de s'y référer pour améliorer la recherche et ainsi autoriser un processus décisionnel souple. Le parcours de l'arbre de recherche avec une méthode taboue par l'antivirus TbScan sur le virus *Jérusalem* est décrit par la figure 3.4. Dans ce cas précis, le virus est détecté. Le parcours de l'arbre de re-

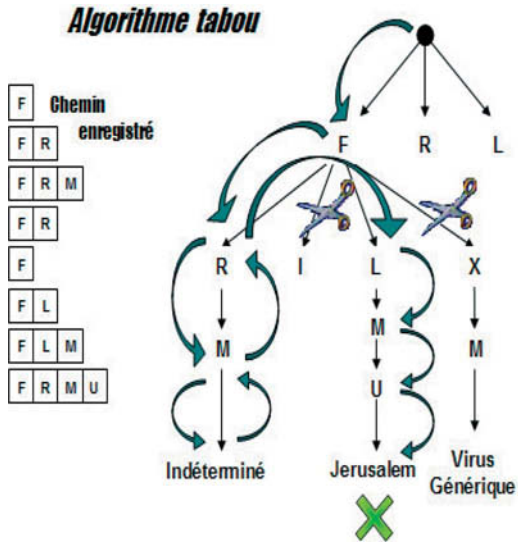


Figure 3.4 – Détection du virus *Jérusalem* par algorithme de type tabou (TbScan)

cherche avec une méthode tabou par l'antivirus TbScan sur le virus *Jerusalem*

est décrit sur la figure 3.4. Dans ce cas précis, le virus est détecté.

Ces métaheuristiques sont plus lourdes à implémenter (stockage de la liste taboue, algorithmique liée aux possibilités de retour en arrière dans le parcours..., en revanche la pertinence du résultat s'en trouve améliorée. Les métaheuristiques de type tabou sont en outre très flexibles selon la nature du problème à résoudre [33, chapitre 2]. En effet, le mécanisme de décision déterminant quelles branches sont marquées taboues est indépendant de la méthode générale et peut donc être facilement adapté lors de l'implémentation.

Le seul défaut des méthodes de type tabou concerne leur « convergence » (capacité à trouver l'optimum global si le temps imparti tend vers l'infini). Pour certaines instances, elles peuvent être incapables de franchir certains optima, en particulier si l'arbre de recherche est mal défini (dans le cas d'un code obfusqué par exemple). Dans un contexte antiviral, cela conduira à la non détection du code. En outre, en pratique, la convergence n'est réalisée qu'avec des temps de recherche prohibitifs pour un antivirus.

Recuit simulé

La méthode dite du *recuit simulé* est une métaheuristique inspirée du processus métallurgique de refroidissement des métaux. Dans ce processus, les cycles de refroidissements lents alternent avec des cycles de réchauffage *(recuit)* qui tendent à minimiser l'énergie du métal. Cette méthode a été proposée en 1983 par des chercheurs d'IBM [76].

En partant d'une solution donnée et en la modifiant, on en obtient une seconde. Soit cette dernière améliore le critère d'optimisation, soit elle le dégrade. Dans le premier cas, on privilégie un optimum local au voisinage considéré pour la solution de départ. Dans le second cas, cela permet d'explorer une plus grande partie de l'espace et de sortir de voisinages bloquants. Pour plus de détails, le lecteur pourra consulter [33, chapitre 1].

Le recuit simulé donne par conséquent la possibilité, durant le parcours dans l'arbre de recherche, de choisir des branches ayant peu de chance d'aboutir mais avec une probabilité très faible (paramétrable). La figure 3.5 illustre la recherche du virus *Jérusalem* par l'antivirus TbScan, par la méthode du recuit simulé.

On voit que le virus sera détecté avec une probabilité de 95 % et identifié comme virus générique dans 5 % des cas. Le recuit simulé offre des résultats assez précis et permet un grande flexibilité. Il est en revanche assez complexe à mettre en œuvre du fait du grand nombre des paramètres à considérer et de leur grande sensibilité. Par exemple, pour une même série de drapeaux, en entrée, deux parcours successifs peuvent donner des résultats plus ou moins différents (voir figure 3.5). La difficulté consiste à attribuer de manière empirique des probabilités aux différentes branches lors du parcours et ce, en fonction de leurs chances d'aboutir. Enfin, la convergence du recuit simulé exige souvent un temps également prohibitif pour une application antivirale.

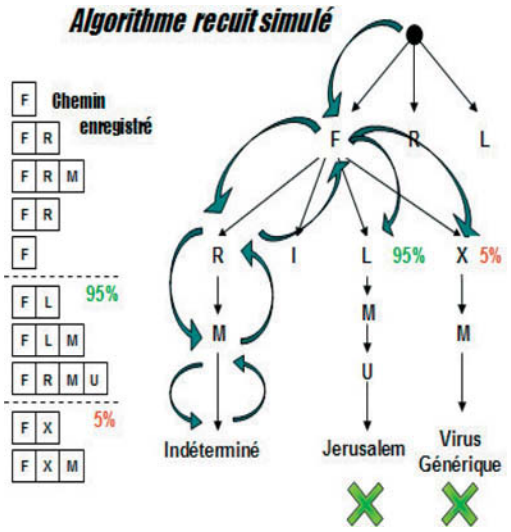


Figure 3.5 – Détection du virus *Jérusalem* par recuit simulé (TbScan)

La détection par analyse heuristique

Que peut-on dire des techniques heuristiques et de leur utilité ? Il n'existe malheureusement pratiquement pas de données techniques vérifiables. Il est cependant sûr que l'apparition de virus plus complexes, avérée et à venir, posera de sérieux problèmes aux techniques heuristiques, quelles qu'elles soient. Rappelons que ces techniques peuvent être modélisées par des tests statistiques et avec eux les deux types d'erreur α et β.

Pour illustrer cela, citons les données communiquées par le concepteur de l'antivirus TbScan [136], en 1999. Sur une collection de 7 120 virus connus, la version 6.02 de cet antivirus a fourni les résultats donnés dans le tableau 3.4. On notera que 0,22 % des virus – pourtant connus – n'étaient pas détectés

Méthode de détection	Nombre de fichiers détectés	Proba. de détection
Scanning classique	7 056	97.86
Heuristiques	6 465	89.67
Scanning + heuristiques	7 194	99.78

Table 3.4 – Résultats de détection par l'antivirus TbScan selon les méthodes utilisées

malgré l'utilisation combinée des deux techniques. La situation en 2006 ne

s'est pas arrangée[14] avec la concurrence acharnée entre les différents éditeurs qui tous doivent avoir le produit le plus fluide possible car l'utilisateur refuse tout ralentissement de sa machine.

L'autre aspect qu'il faut en effet garder à l'esprit est que les techniques heuristiques, avec la complexité croissante et alarmante des nouveaux virus, réclament, pour un minimum d'efficacité, des temps de calcul définitivement rédhibitoires pour un antivirus. Cela signifie que lorsqu'une technique est trop gourmande (heuristique ou autre), soit elle n'est pas mise en œuvre, soit elle l'est partiellement (nombre d'itérations limité). Au final, le risque β est donc plus élevé.

3.6 La simulabilité des tests statistiques

La présentation des tests statistiques de la section 3.2.2 et des techniques heuristiques dans la section 3.5 nous a permis de comprendre les limites inhérentes à toute technique de détection virale. Mais, outre ces risques permaments et incompressibles d'échec (non détection et fausse alarme), il est une situation encore plus grave à considérer : celle où l'attaquant, connaissant les outils statistiques de détection, les utilise contre le « défenseur ». Il s'agit de la *simulabilité* des tests [50].

En effet, si les limites des tests et des heuristiques que nous avons exposées peuvent être considérées comme « naturelles » (la variabilité de la « Nature » étant indépendante de toute considération de malveillance), quand l'attaquant les utilise à son profit, non seulement il est en mesure de contourner les protections de la défense mais, plus grave, il peut manipuler totalement cette dernière. Donnons une définition générale de la simulabilité de tests.

Définition 3.3 *Simuler un test consiste, pour un adversaire, à introduire, dans une population donnée $\mathcal{P}$, un biais statistique non détectable pour un évaluateur, relativement à un ensemble de tests donnés.*

La notion de simulabilité s'applique de manière égale à des tests mais aussi à des heuristiques. Sans restriction conceptuelle particulière, nous nous limiterons aux cas des tests, dans la mesure où nous avons montré précédemment que toute technique de décision pouvait se ramener à ce cas.

Il existe deux formes de simulabilité :

- la première ne dépend pas des paramètres du ou des tests considérés par celui qui défend. C'est ce que nous appellerons la *simulabilité forte* ;
- la seconde, au contraire, dépend des paramètres du ou des tests que l'on souhaite simuler. C'est la *simulabilité faible.*

Dans ce qui suit, nous appellerons « *testeur* » celui qui utilise et met en œuvre des tests statistiques pour prendre une décision.

[14] Les résultats disponibles sur `http://www.virus.gr/english/fullxml/default.asp?id=82&mnu=82` sont à ce titre particulièrement édifiants.

3.6.1 La simulabilité forte

Donnons tout d'abord une définition du concept.

Définition 3.4 *(Simulabilité forte d'un test statistique) Soit une propriété P et soit un test T destiné à vérifier si P est valide pour une population $\mathcal{P}$ donnée. Simuler fortement ce test, c'est concevoir ou modifier la population $\mathcal{P}$ de manière à ce que T décide systématiquement, au risque d'erreur près, que P est vérifiée sur $\mathcal{P}$, mais qu'il existe un test T' tel que ce dernier décide du contraire. De la même manière, on dira que l'on simule fortement t tests $(T_1, T_2, \ldots, T_t)$ si leur application conjointe conduit à décider que P est vraie sur $\mathcal{P}$ mais ne l'est plus si on considère un $(t+1)$-ième test T_{t+1}.*

En terme de sécurité, la simulabilité forte des tests revêt une importance capitale, notamment quand le testeur n'utilise que t tests pour évaluer une population $\mathcal{P}$ proposée par un tiers qui lui seul connaît le $(t+1)$-ième test T_{t+1}. Toujours dans le contexte de la sécurité et de l'évaluation de la sécurité, on peut alors résumer les principaux cas de la manière suivante :

- le testeur et le tiers ne connaissent pas le test T_{t+1}. C'est le cas où un produit de sécurité – un algorithme de cryptologie, un logiciel antivirus...– est mis sur le marché et où tout le monde estime qu'il présente un « bon niveau » de sécurité (autrement dit, la propriété P attendue est vérifiée sur la population $\mathcal{P}$). Quand un chercheur imagine un test[15] T_{t+1}, alors le produit est déclaré vulnérable. En cryptanalyse, par exemple, le travail consiste précisément à imaginer des tests mettant en lumière des faiblesses exploitables. Dans le domaine de la détection virale, l'évolution des stratégies de détection suit également ce principe ;
- le testeur ignore T_{t+1} mais pas le tiers – qui alors devient un attaquant. C'est le cas où le tiers a inséré une trappe dans le système. Le test T_{t+1} constitue alors un secret qui permet au tiers seulement de mettre à mal la sécurité d'un produit. On parle alors de trappe dans le produit. Une instance de ce cas a été présentée lors de la conférence SSTIC 2003 [142] : la trappe est une structure algébrique. Indétectable avec les tests classiques, elle le devient en appliquant un test de plus, connu, le *test du rang*. En modifiant alors le système, dans le but de simuler également le test du rang, ce dernier conclut à l'absence de biais après modification du système qui est alors décidé bon pour le service. D'autres exemples ont été proposés dans [50].
  Dans un contexte de détection virale, le programmeur de code malveillant, qui connaît tous les tests mis en œuvre par l'éditeur du produit, génère des codes indétectables pour ces tests mais qui ne le sont pas pour le programmeur. D'une manière presque caricaturale mais néanmoins pertinente, la simple recherche de signatures constitue le meilleur exemple possible : la base de t signatures représente les tests connus, la nouvelle signature – avant le processus de mise à jour – correspond au test T_{t+1}.

[15] Le lecteur notera qu'il existe en général un très grand nombre de tests T_{t+1} possibles.

3.6.2 La simulabilité faible

Donnons tout d'abord la définition du concept.

Définition 3.5 *(Simulabilité faible d'un test statistique) Soit une propriété P et soit un test T destiné à vérifier si P est valide pour une population $\mathcal{P}$ donnée. Simuler faiblement ce test, c'est introduire dans la population $\mathcal{P}$ une nouvelle propriété P' modifiant partiellement la propriété P, de manière à ce que T décide systématiquement, au risque d'erreur près, que P est vérifiée sur $\mathcal{P}$.*

La simulabilité faible se distingue de la simulabilité forte par le fait que l'on travaille avec les mêmes tests que ceux utilisés par le testeur. L'attaquant va donc introduire un biais que la sensibilité des tests utilisés ne permettra pas de détecter, en vertu des risques généralement utilisés par le testeur.

La propriété P' de la définition s'oppose en règle générale à la propriété P. Elle constitue précisément une vulnérabilité que le tiers veut créer et exploiter sans que le testeur ne s'en rende compte lors de l'évaluation. Mettre en œuvre la simulabilité faible est assez délicat et réclame de bien connaître la structure et les propriétés mathématiques du ou des tests à simuler, en particulier quand il y a plusieurs tests à appréhender simultanément.

La philosophie générale de la simulabilité faible consiste à introduire la propriété P' de sorte que le ou les estimateurs E utilisés restent dans la zone de convergence (région d'acceptation du test). Lors de la phase de décision, rappelons que l'on regarde si $E < S$ (comparaison de l'estimateur avec un seuil de décision). Simuler faiblement le test, consiste alors à jouer avec la valeur $S - E$ tout en s'assurant qu'elle reste positive. Pour ce faire, on exploite les propriétés de la distribution d'échantillonnage tout en exploitant la marge offerte par les paramètres du test.

Nous allons maintenant considérer quelques exemples de simulabilité de tests statistiques (et plus généralement des techniques assimilées comme les heuristiques). Il est essentiel de bien comprendre que la première étape passe par l'analyse du ou des antivirus que l'attaquant aura à affronter et qu'il devra contourner. Cette étape, qui peut être sinon complexe du moins fastidieuse selon les produits[16], vise à déterminer quels sont les paramètres statistiques (distributions, moments de premier et deuxième ordre) décrivant ces produits. Cette étape est malheureusement assez facile à réaliser en pratique, dans la mesure où les modèles statistiques mis en œuvre sont plutôt frustres. Des techniques d'échantillonnage et de tests permettront en général à l'attaquant d'établir assez vite un modèle opérationnellement satisfaisant. En pratique, l'étude concernera essentiellement la ou les fonctions de détection définies dans le chapitre 2 et, pour les produits les plus évolués, les fonctions de stratégie antivirale, présentées dans la section 2.7.1.

[16] Elle n'est à faire qu'une seule fois. En outre, les éditeurs font très rarement varier leurs produits, et donc les modèles statistiques les décrivant.

3.6.3 Application : contourner la détection des flux

La détection des vers et autres codes malveillants utilisant les réseaux pour se propager est fondée sur une approche statistique. Il n'est en effet pas possible, au niveau d'un centre de surveillance du trafic Internet (localisé chez les fournisseurs d'accès, les éditeurs d'antivirus ou les sociétés spécialisées comme WebSense, MessageLabs...), d'analyser systématiquement chaque paquet d'un trafic. Or la détection d'un ver inconnu, notamment dans les premiers instants de l'épidémie, est essentielle[17]. La détection peut se faire sur le contenu des paquets, c'est un problème de détection antivirale classique avec toutes les limitations propres aux antivirus. La détection peut également se faire sur la dynamique du flux elle-même.

La modélisation de la propagation des vers est un champ de recherche vital. Si, en effet, on dispose de modèles adéquats, il est possible, avec une probabilité de succès non négligeable, de prévoir le comportement de la prochaine attaque, pour peu qu'elle corresponde au modèle. Ainsi pour les vers simples (*CodeRed, Blaster* ou *Slammer*) a été développé le modèle RCS *(Random Constant Spread)* [128]. Notons N le nombre total de serveurs vulnérables à une attaque de ce type (du fait d'une faille logicielle par exemple). Soit K, le taux moyen de compromission de ces serveurs (nombre moyen de serveurs infectés par minute, en phase initiale d'épidémie, par un serveur déjà infecté). Notons que K est défini de sorte à modéliser la propagation indépendamment des considérations de vitesse de processeur, de bande passante, de localisation dans le réseau... Si $a(t)$ est la proportion de serveurs qui ont été compromis, à l'instant t, le modèle de propagation RCS peut alors être décrit par une simple équation différentielle :

$$\frac{da}{dt} = Ka(1-a) \tag{3.3}$$

La solution de cette équation est la courbe appelée *courbe logistique* donnée par l'équation suivante :

$$a\frac{\exp(K(t-T))}{1+\exp(K(t-T))},$$

où T est un paramètre temporel représentant le pic maximal du taux de propagation. Dans le cas du ver *Codered*, nous avons $K = 1,8$ et $T = 11,9$. La courbe logistique correspondante est celle de la figure 3.6. D'autres modèles dérivés ou différents ont également été proposés (par exemple [121]).

Comment peut alors faire un pirate pour contourner ces modèles de prédiction ? Il peut jouer sur la constante K en utilisant un générateur d'adresses IP aléatoires moins agressif, de manière à infecter moins d'hôtes par unité de temps. La propagation est par conséquent ralentie.

Comment ces modèles sont-ils exploités en pratique ? Les centres d'opération et d'analyse de trafic Internet auscultent en permanence ce qui passe au

[17] Notons que depuis 2004 avec le ver *Sasser*, on n'observe plus de grandes épidémies soudaines et de grande intensité. Les modèles de propagation semblent avoir changé, précisément pour contourner les sondes mises en place dans le monde.

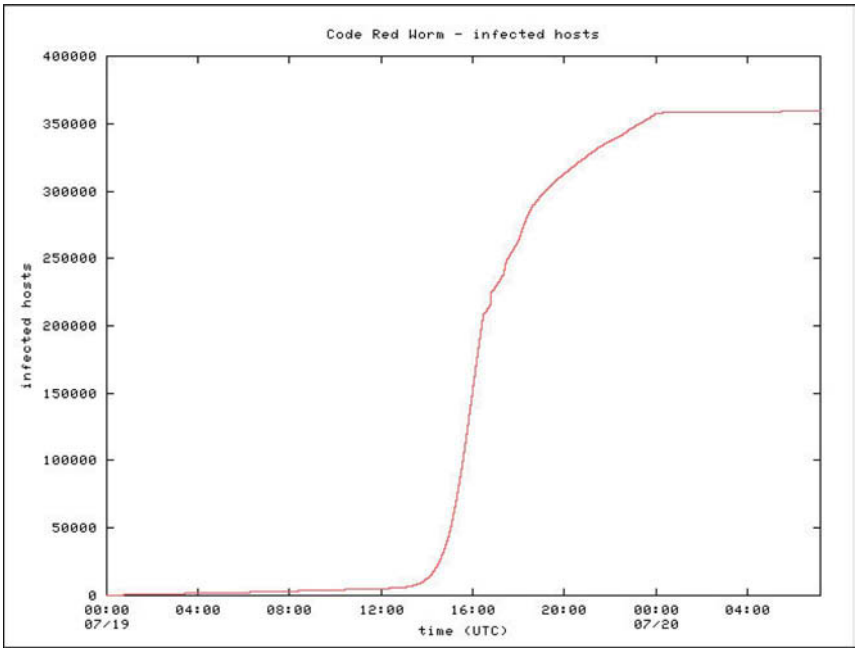


Figure 3.6 – Courbe logistique du ver *Code Red*

moyen de sondes disposées dans le monde entier. Ces sondes détectent immédiatement toute augmentation anormale du trafic. Autrement dit, si l'on note N_p le nombre de paquets transitant via un port donné, par unité de temps, le test statistique suivant est mis en œuvre[18]

$$\begin{cases} \mathcal{H}_0 & N_p \text{ suit une loi } \mathcal{L}(\mu_0, \sigma_0), \\ \mathcal{H}_1 & N_p \text{ suit une loi } \mathcal{L}(\mu_1, \sigma_1). \end{cases}$$

La technique est alors celle d'un test statistique : détermination d'un risque α et d'un seuil de décision S. Si $N_p > S$, alors une alerte est donnée et les paquets sont systématiquement analysés. La figure 3.7 illustre ce principe de détection statistique dans le cas de l'infection par le ver *Blaster* en 2003 [40]. L'accroissement soudain d'activité sur le port 135 était inmanquablement détectable. Si l'attaquant connaît ou devine empiriquement les paramètres du test de détection (la loi gouvernant l'hypothèse $\mathcal{H}_0$), il sait alors comment échapper à la détection par surveillance de la dynamique de flux.

[18] Les lois de probabilités seront notées $\mathcal{L}$. Décrivant des flux de données, ces lois peuvent être très différentes selon le flux considéré.

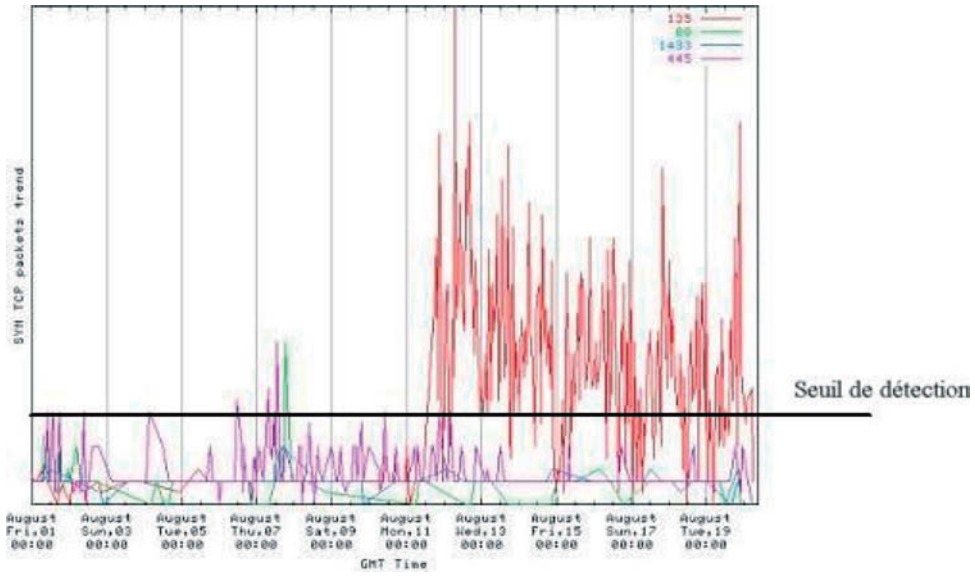


Figure 3.7 – Détection statistique de dynamique de flux du ver *Blaster*

3.6.4 Application : contourner le contrôle du contenu

Le chiffrement est une technique très utilisée par les codes malveillants dans un but de polymorphisme/métamorphisme (traité dans le chapitre 6) ou de blindage (technique présentée dans le chapitre 8). Si le chiffrement se révèle une protection potentiellement efficace, il présente un inconvénient majeur dans le domaine de la détection. En effet, caractériser un fichier ou un flux chiffré est très simple. Un simple calcul d'entropie permet de détecter la présence de données chiffrées ou compressées. La stratégie mise en œuvre par certains fournisseurs consiste alors à supprimer les courriers électroniques dont la pièce jointe est chiffrée – ou au minimum compressée en utilisant un mot de passe.

Voyons comment simuler efficacement n'importe quelle donnée (image, texte clair dans une langue donnée, exécutable, document dans un format donné...). Autrement dit, comment faire en sorte que des données chiffrées aient statistiquement l'air de données en clair, tout en conservant la sécurité procurée par le chiffrement.

Tout d'abord, comment caractérise-t-on statistiquement des données chiffrées ? Un simple calcul d'entropie permet de le faire. Soit un alphabet $\mathcal{A}$ donné

comportant n symboles ($n = 2$ si nous considérons des bits). Chaque symbole apparaît dans une donnée codée à l'aide de l'alphabet $\mathcal{A}$, avec une probabilité p_i, $(i = 1, 2, \ldots, n)$. Bien évidemment, $p_1 + p_2 + \ldots + p_n = 1$. Si toutes les probabilités sont égales, nous les notons p et $p = \frac{1}{n}$. Avec ces notations, la quantité d'information contenue dans une donnée D codée sur un alphabet de taille n est donnée par la formule (3.4).

$$H_n(D) = -\sum_{i=1}^{n} p_i \log_2(p_i). \tag{3.4}$$

Cette entropie dépend directement des probabilités p_i de chaque symbole de l'alphabet. Elle est maximale lorsque $p_i = p, \forall i$. Ainsi un texte aléatoire binaire (composé d'un unique bit ; cela s'étend à des données chiffrées de n bits) a une entropie de 1. Pour mieux voir les choses, on peut également définir la notion de redondance (quantité d'information rendondantes) comme la différence entre l'entropie maximale possible et celle effectivement réalisée par une donnée D donnée. Elle est donnée par l'équation (3.5).

$$R_n(D) = -\log_2(\frac{1}{p}) - \left(-\sum_{i=1}^{n} p_i \log_2(p_i)\right) = -\log_2(n) + \sum_{i=1}^{n} p_i \log_2(p_i). \tag{3.5}$$

Pour des données compressées ou chiffrées, l'entropie est maximale et par conséquent la redondance est nulle. Notons que dans le cas du chiffrement, nous avons $H_n(D)$ maximale et $R_n(D) = 0$ quelle que soit la valeur de $n = 2^m$. Cela n'est pas vrai pour du texte compressé.

Le principe du filtrage est alors le suivant. Considérons un filtre $\mathcal{F}$ donné qui bloque les données chiffrées ou compressées. Il calcule donc pour chaque fichier F, la valeur $H_{256}(F)$. Il considère qu'en moyenne $H_{256}(F)$ appartient à un intervalle donné (autrement dit, il construit un test statistique comme décrit dans la section 3.3).

Le principe de simulation est alors le suivant. Soit une donnée D anodine (image ou texte dans une langue donnée). Pour $m = 8$, calculons les valeurs p_i des 256 symboles et l'entropie $H_{2^8}(D)$ correspondante. Il suffit alors d'appliquer la méthode suivante[19] pour un fichier F donné que l'on veut chiffrer et que le filtre $\mathcal{F}$ laissera malgré tout passer. Le principe consiste à rajouter à des endroits aléatoires des données binaires, dites *données de simulation*, dans le texte chiffré, de sorte à globalement réaliser les probabilités p_i de la donnée anodine D, leurrant ainsi le filtre $\mathcal{F}$. Le procédé général de simulation est composé d'un système de chiffrement fort, noté E, et d'un module de simulation

[19] Cette méthode constitue un scénario didactique assez basique mais néanmoins très efficace. Il est assez facile d'en concevoir des variantes très sophistiquées dès lors que le principe de simulabilité statistique est maîtrisé. En particulier, pour contrôler des effets locaux indésirables, il est intéressant de considérer simultanément les valeurs $H_m(F)$ pour $m = 1, 2, \ldots, 8$ avec $n = 2^m$ (en pratique, les filtres ne vont pas au-delà). La dépendance markovienne peut également être considérée pour gérer les filtres plus sophistiqués. Le lecteur pourra lire [125] pour une description d'un procédé de simulabilité utilisant des chaînes de Markov.

de la donnée anodine D, noté S_D. En fait, S_D représente une source d'entropie $H_{2^8}(D)$. Les principales étapes sont alors les suivantes :

1. en entrée, nous considérons un fichier F et une clef K. Le procédé de simulation doit produire un fichier C ;
2. les données sont chiffrées avec le système de chiffrement fort. Nous utiliserons un système de chiffrement par flot (type RC4) [41] et $E_K(F)$ désignera le chiffrement initial proprement dit (avant simulation). Les données de simulation seront notées $S_D(F)$;
3. un tableau T_{sim} de taille 256 est créé et initialisé à zéro ; il contiendra les effectifs d'octets de C ;
4. un tableau P_{sim} de taille 256 est créé et initialisé avec les valeurs théoriques p_i (données anodines) ;
5. un tableau P_D de taille 256 contenant les valeurs théoriques p_i à simuler est créé. Il servira de référence ;
6. les étapes de simulation est alors les suivantes :
   (a) le système de simulation S_D produit une suite initiale $S_D(F)$, de L bits (L est un paramètre). En pratique, $L \geq 1000$. On écrit alors dans le fichier de sortie C ces L bits qui par construction réalisent les probabilités p_i ;
   (b) les tableaux T_{sim} et P_{sim} sont actualisés avec les effectifs réalisés pour chacun des 256 octets possibles ;
   (c) le système de chiffrement E_K produit une suite chiffrante de k bits (en pratique $k \in \{64, 128\}$) et produit k bits de chiffré (addition modulo 2 avec le clair F), lesquels sont écrits dans C ;
   (d) les tableaux T_{sim} et P_{sim} sont alors mis à jour pour tenir compte des modifications provoquées par les données chiffrées ;
   (e) le système de chiffrement E_K produit une suite de 8 bits qui représente la décomposition binaire de la nouvelle valeur de L ;
   (f) le processus est alors itéré jusqu'à ce que tout le fichier F soit chiffré.

Lors de chaque itération, pour chaque valeur $i = 0, 1, \ldots, 255$, si $P_{sim}[i] \geq P_D[i]$, alors le caractère ASCII de code décimal i n'est pas utilisé dans les données de simulation. Au contraire, si $P_{sim}[i] < P_D[i]$, il l'est. Des procédures de recalage permettent de corriger certains écarts en phase finale. Le lecteur montrera que le déchiffrement est bien univoque dès lors que la valeur initiale du paramètre L et que K sont connus.

Au final, le fichier C aura le même profil d'entropie qu'un fichier de type D. Le filtre $\mathcal{F}$ le laissera passer (relativement au test de l'entropie). Notons qu'il est possible d'implémenter ce procédé de manière très rapide. Le déchiffrement des données (incluant le « tri » entre données de simulation et données chiffrées) est direct et plus rapide encore.

Cette technique est également applicable pour simuler du code binaire. Il suffit alors de calculer les valeurs p_i des 256 symboles et l'entropie correspondante puis d'appliquer la méthode précédemment décrite ou celle présentée dans [125].

3.6.5 Application : leurrer la détection virale

Dans la section 3.4, nous avons montré que les techniques de détection virales pouvaient en fait être ramenées à des tests statistiques. Cette vision permet de mieux comprendre l'approche choisie par l'attaquant pour contourner un antivirus. Nous prendrons comme exemple la technique de détection par analyse spectrale, cette dernière représentant l'un des modèles statistiques actuellement les plus élaborés (les autres techniques étant couvertes par des modèles plus triviaux). Bien évidemment, cet exemple est facilement transposable à toute autre technique de détection, dès lors que son modèle statistique est connu.

Rappelons le principe de l'analyse spectrale. Elle consiste à établir la liste de certaines instructions, jugées représentatives d'un programme. Cette liste est appelée le *spectre*. Ces instructions sont choisies en fonction de leur fréquence d'apparition dans les programmes sains (non infectés). Tout compilateur n'utilise en réalité qu'une partie de l'ensemble des instructions théoriquement possibles (le plus souvent afin d'optimiser le code). En revanche, un code malveillant va tenter d'utiliser plus largement ce jeu d'instructions, pour accroître son efficacité. Dans le cas du polymorphisme/métamorphisme, en particulier, un tel code modifie de manière importante son jeu d'instructions dans le but d'interdire toute analyse de forme.

Un exemple célèbre est celui du moteur métamorphe du virus `Win32/Linux.MetaPHOR` [132]. Un extrait des commentaires de son code source donne quelques exemples de transformations par réécriture à l'aide d'instructions équivalentes (le lecteur pourra étudier ce code source qui contient de très nombreuses autres techniques du même genre ; voir également les exercices en fin de chapitre). Ce virus utilise de manière intensive des techniques de réduction de code ou, au contraire, d'expansion de code, lesquelles sont générées de manière pseudo-aléatoire.

```
[...]
;;  Transformations over single instructions:
;;
;;    XOR Reg,-1                --> NOT Reg
;;    XOR Mem,-1                --> NOT Mem
;;    MOV Reg,Reg               --> NOP
;;    SUB Reg,Imm               --> ADD Reg,-Imm
;;    SUB Mem,Imm               --> ADD Mem,-Imm
;;    XOR Reg,0                 --> MOV Reg,0
;;    XOR Mem,0                 --> MOV Mem,0
```

```
;;   ADD Reg,0                 --> NOP
;;   ADD Mem,0                 --> NOP
;;   OR  Reg,0                 --> NOP
;;   OR  Mem,0                 --> NOP
;;   AND Reg,-1                --> NOP
;;   AND Mem,-1                --> NOP
;;   AND Reg,0                 --> MOV Reg,0
;;   AND Mem,0                 --> MOV Mem,0
;;   XOR Reg,Reg               --> MOV Reg,0
;;   SUB Reg,Reg               --> MOV Reg,0
;;   OR  Reg,Reg               --> CMP Reg,0
;;   AND Reg,Reg               --> CMP Reg,0
;;   TEST Reg,Reg              --> CMP Reg,0
;;   LEA Reg,[Imm]             --> MOV Reg,Imm
;;   LEA Reg,[Reg+Imm]         --> ADD Reg,Imm
;;   LEA Reg,[Reg2]            --> MOV Reg,Reg2
;;   LEA Reg,[Reg+Reg2]        --> ADD Reg,Reg2
;;   LEA Reg,[Reg2+Reg2+xxx]   --> LEA Reg,[2*Reg2+xxx]
;;   MOV Reg,Reg               --> NOP
[...]
```

Considérons, pour un type de compilateur donné $\mathcal{C}$, un spectre constitué d'une liste d'instructions $(I_i)_{1\leq i\leq c}$, chacune d'entre elles étant accompagnée d'une fréquence théorique d'apparition n_i caractérisant son occurrence dans des programmes « normaux » générés par le compilateur considéré. La notion de « normalité » correspond en fait à une modélisation en moyenne de la représentation de ces instructions dans des programmes non infectés, et donc à un comportement en moyenne des compilateurs. Le spectre, que nous noterons $spec_i(\mathcal{C})$, s'écrit alors

$$spec_j(\mathcal{C}) = (I_i, n_i)_{1\leq i\leq c}.$$

L'indiciation en j signifie qu'en général plusieurs spectres de référence sont utilisés. Les instructions du spectre peuvent être ou non regroupées en classes, selon différents regroupements possibles. D'un point de vue pratique, plusieurs tests statistiques sont réalisés à la fois sur les échantillons de code analysés mais également sur les résultats de ces premiers tests pour détecter une anomalie sur la première série de résultats[20]. Sans perte de généralité, et pour simplifier le propos, nous considérerons un seul spectre.

Lors de l'analyse d'un programme, pour chacune de ces instructions, l'effectif observé $\hat{n}_i$ est alors calculé, après n observations. L'hypothèse à tester est

[20] Le résultat d'un test statistique peut être considéré sous certaines conditions comme une variable aléatoire. Le comportement de cette variable peut alors être testé comme n'importe quelle autre variable.

l'hypothèse nulle $\mathcal{H}_0$ définie par[21] :

$$\mathcal{H}_0 : \hat{n}_i = n_i \qquad 1 \leq i \leq c.$$

L'hypothèse alternative $\mathcal{H}_1$ consiste à dire qu'il existe au moins une instruction (ou catégorie) k pour laquelle nous avons $\hat{n}_k \neq n_k$ (en fait, par construction, il y en aura au moins deux, du fait que nous avons $\sum_{i=1}^{c} \hat{n}_i = n$).

Si c intructions composent le spectre, l'estimateur

$$D^2 = \sum_{i=1}^{c} \frac{(\hat{n}_i - n_i)^2}{n_i}$$

est calculé. Il s'agit d'un test pour variables catégorielles[22]. En utilisant des approximations gaussiennes, il a été démontré que la statistique de test D^2 admet sous l'hypothèse nulle $\mathcal{H}_0$ une loi asymptotique du chi^2 à $c-1$ degrés de liberté[23].

Pour un seuil de signification fixé α, la procédure de test consiste à comparer l'estimateur D^2 avec le seuil de décision $\chi^2_{(\alpha,c-1)}$ et à décider comme suit :

$$\begin{cases} \mathcal{H}_0 & \text{si } D^2 \leq \chi^2_{(\alpha,c-1)} \\ \mathcal{H}_1 & \text{si } D^2 < \chi^2_{(\alpha,c-1)}. \end{cases}$$

Expliquons maintenant comment simuler faiblement un tel test. Supposons que, lors du processus métamorphique d'un virus, des techniques de réécriture de code (à des fins de mutation, de réduction ou d'expansion de code) modifient les fréquences d'apparition de certaines instructions prises en compte par le spectre d'un ou plusieurs antivirus. Le moteur de métamorphisme va aléatoirement choisir les modifications à opérer. Comment le processus aléatoire doit-il être réalisé, de sorte que le test d'analyse spectrale reste muet ?

Sans perte de généralité, supposons que deux instructions I_{i_1} et I_{i_2} du spectre sont affectées par le processus de mutation du code. Nous allons montrer de quelle manière le moteur métamorphique devra paramétrer ses mutations pour simuler faiblement le test d'analyse spectrale.

Pour ce test, les instructions I_{i_1} et I_{i_2} sont supposées apparaître en moyenne avec une fréquence respectivement estimée à n_{i_1} et n_{i_2}. Pour faciliter la compréhension, considérons tout d'abord le code initial (souche 0). Le programmeur

[21] Ce test, plus connu sous le nom du test du chi-carré (χ^2), est également appelé test d'homogénéité.

[22] Une variable catégorielle est une généralisation de la notion de variable dite de Bernoulli. Alors que cette dernière ne peut recevoir que deux résultats possibles, une variable catégorielle peut en recevoir $c \geq 2$.

[23] La notion de degré de liberté permet de tenir compte des éventuelles dépendances existant entre les fréquences ou de corriger les erreurs résultant de calcul des fréquences théoriques, non pas à partir des paramètres théoriques de la population mais de leur estimation. Ainsi, dans le premier cas, comme $\sum_{i=1}^{c} \hat{n}_i = n$, dès que $c-1$ fréquences théoriques sont connues, la c-ième est fixée (elle est donc dépendante des $c-1$ autres). Dans le deuxième cas, si les fréquences théoriques ne peuvent être calculées qu'en estimant m paramètres de la population, alors le nombre de degrés de liberté sera donnée par $c-1-m$.

l'a conçu de sorte que :

$$D^2 = \sum_{i=1}^{c} \frac{(\hat{n}_i - n_i)^2}{n_i} \leq \chi^2_{(\alpha,c-1)}, \tag{3.6}$$

pour un risque α donné. La valeur initiale, de référence, de D^2 sur la souche 0 sera notée D_0^2 et nous noterons $F = \chi^2_{(\alpha,c-1)} - D^2$.

Le processus de mutation modifie alors les fréquences $\hat{n}_{i_1}$ et $\hat{n}_{i_2}$ comme suit :

$$\hat{n}_{i_1} \rightarrow \hat{n}'_{i_1} \qquad \hat{n}_{i_1} + \delta_1 = \hat{n}'_{i_1} \tag{3.7}$$

$$\hat{n}_{i_2} \rightarrow \hat{n}'_{i_2} \qquad \hat{n}_{i_2} + \delta_2 = \hat{n}'_{i_2} \tag{3.8}$$

Les valeurs δ_i peuvent être négatives ou positives. En première approche, il est évident qu'un bon critère (initial) est de faire en sorte que $E[\delta_1 + \delta_2] = 0$ (l'espérance mathématique des modifications cumulées est nulle). Ce critère est cependant assez restrictif et il est probable que le nombre de possibilités de paramétrage du moteur de mutation sera limité. En particulier, ce critère dépend uniquement de la souche 0 et non pas des fréquences théoriques n_i. Nous allons définir un critère plus général.

Si nous reportons les modifications des fréquences $\hat{n}_{i_1}$ et $\hat{n}_{i_2}$ dans l'expression (3.6), nous pouvons écrire

$$D^2 = \sum_{i=1, i \notin \{i_1, i_2\}}^{c} \frac{(\hat{n}_i - n_i)^2}{n_i} + \frac{(\hat{n}_{i_1} - n_{i_1})^2}{n_{i_1}} + \frac{(\hat{n}_{i_2} - n_{i_2})^2}{n_{i_2}}. \tag{3.9}$$

Cela permet alors de se ramener à la valeur D_0^2, en prenant en compte les équations (3.7) et (3.8) :

$$D^2 = D_0^2 + \frac{2\delta_1(\hat{n}_{i_1} - n_{i_1}) + \delta_1^2}{n_{i_1}} + \frac{2\delta_2(\hat{n}_{i_2} - n_{i_2}) + \delta_2^2}{n_{i_2}}. \tag{3.10}$$

L'équation (3.10) permet alors de donner un second critère :

$$E[\frac{2\delta_1(\hat{n}_{i_1} - n_{i_1}) + \delta_1^2}{n_{i_1}} + \frac{2\delta_2(\hat{n}_{i_2} - n_{i_2}) + \delta_2^2}{n_{i_2}}] = 0. \tag{3.11}$$

ou en simplifiant

$$E[2\delta_1(\hat{n}_{i_1} - n_{i_1})n_{i_2} + \delta_1^2 n_{i_2} + 2\delta_2(\hat{n}_{i_2} - n_{i_2})n_{i_1} + \delta_2^2 n_{i_1}] = 0. \tag{3.12}$$

Une rapide analyse permet de montrer que ce second critère est moins restrictif que celui selon lequel $E[\delta_1 + \delta_2] = 0$ (voir exercices).

Le générateur de mutation (dans le cadre du polymorphisme ou du métamorphisme) devra donc aléatoirement modifier les instructions I_{i_1} et I_{i_2} de sorte que les fréquences $\hat{n}_{i_1}$ et $\hat{n}_{i_2}$ vérifient le critère défini par l'équation (3.12).

Ce schéma de simulabilité faible peut être davantage généralisé. Nous n'avons ici considéré, dans un but didactique, qu'un scénario assez simple, impliquant des outils statistiques basiques. Précisons quelques voies plus sophistiquées et faisons quelques remarques :

- le critère défini par l'équation (3.12), d'un point de vue pratique, doit être mis en œuvre en considérant la variance de la valeur

$$\epsilon = 2\delta_1(\hat{n}_{i_1} - n_{i_1})n_{i_2} + \delta_1^2 n_{i_2} + 2\delta_2(\hat{n}_{i_2} - n_{i_2})n_{i_1} + \delta_2^2 n_{i_1}.$$

  En effet, le critère ne considère que l'espérance mathématique, ce qui n'est pas suffisant. En réalité, la simulabilité faible exige que

$$D_0^2 + \epsilon < \chi^2_{(\alpha,c-1)},$$

  autrement dit que $\epsilon < F$. Si l'on ne considère que la valeur $E[\epsilon]$, avec une probabilité non nulle, la valeur $\chi^2_{(\alpha,c-1)}$ peut être dépassée conduisant au rejet de l'hypothèse $\mathcal{H}_0$. Il est donc nécessaire de prendre en compte le comportement moyen des écarts à la moyenne de ϵ, autrement dit sa variance ;
- il est intéressant d'impliquer un plus grand nombre d'instructions dans la mesure où cela permet de créer des modifications « factices » visant à compenser d'autres modifications, elles indispensables, et ce afin de rester dans la zone d'acceptation du test ;
- la valeur ϵ, généralisée à un plus grand nombre d'instructions, peut également avoir une espérance négative, c'est-à-dire que l'on peut avoir $-D_0^2 \leq \epsilon \leq F$. Cette approche autorise une plus grande latitude de simulabilité faible.

3.6.6 Un modèle statistique du résultat d'indécidabilité de Cohen

Nous avons vu dans la section 3.3, avec la proposition 3.1, que si l'on peut faire tendre la probabilité de non détection vers 0, cette dernière ne sera jamais nulle. Ce résultat constitue, en quelque sorte, une « version statistique » de la preuve de l'indécidabilité de la détection virale. En effet, si une série de tests suffisamment grande (éventuellement $n \to \infty$) produisait une probabilité de non détection exactement nulle, on aboutirait alors à la conclusion contradictoire que le problème de la détection est décidable. Or, il ne l'est pas [26] [38, chapitre 3]. D'autre part, le modèle que nous avons présenté dans cette section, montre que la probabilité de fausse alarme serait logiquement de 1 (voir la figure 3.1). Autrement dit, on détecterait systématiquement tout fichier comme infecté... ce qui est une méthode « peu pertinente » de résolution de la détection virale.

Le notion de simulabilité des tests statistique va nous permettre de donner une illustration statistique encore plus pertinente du résultat d'indécidabilité de Cohen. Dans [26], Fred Cohen a imaginé un virus dit *contradictoire* et noté CV pour illustrer intuitivement ce résultat (si ce dernier est plus intuitif que

la démonstration mathématique, il en reprend l'esprit et de fait cette dernière est tout aussi exacte). Considérons une procédure de décision D permettant de distinguer un virus V de tout autre programme. Le pseudo-code du virus CV est le suivant :

```
CV()
 {
   ......
   main()
    {
     si non D(CV) alors
      {
       infection();
       si condition vraie alors charge_finale();
      }
     fin si
     aller au programme suivant
    }
 }
```

La procédure de décision D détermine si CV est un virus. Dans le cas de CV lui-même, que se passe-t-il ?

- si D décide que CV est un virus, aucune infection ne survient (CV n'est alors pas un virus) ;
- en revanche, si D décide du contraire (CV n'est pas un virus), l'infection survient et CV se révèle bien être un virus.

Cet exemple montre que la procédure D est contradictoire et que toute détection fondée sur D est impossible car il existe au moins un virus, CV, qui ne sera pas détecté.

Selon le modèle de détection statistique et la notion de simulabilité des tests, le virus CV peut alors être formulé comme suit. La procédure D est constituée de n tests statistiques $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$. Le pseudo-code peut alors être réécrit comme suit :

```
CV()

   ......
   simul(D, V)

     produire V' en simulant D à partir de V
     retourner V'


   main()

     si non D(CV) alors
```

```
    infection();
    si condition vraie alors charge_finale();

  sinon

    V' = simul(D, CV)
    exécuter V'

  fin si
  aller au programme suivant
```

Comme précédemment, nous avons :

- Si D décide que CV est un virus, alors CV mute en simulant les tests $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$ de D.
- En revanche, si D décide du contraire (CV n'est pas un virus), l'infection survient et CV se révèle bien être un virus.

Au final, D est tout aussi contradictoire.

3.7 Conclusion

Ce chapitre a permis de décrire d'un point de vue très général, sous l'angle statistique, la détection antivirale. Il nous a permis d'expliquer pourquoi nos antivirus sont par nature d'une efficacité limitée mais également comment ces limitations pouvaient être exploitées par l'attaquant pour retourner ces techniques de défense contre celui qui les met en œuvre.

Le point essentiel qu'il faut conserver à l'esprit – et que l'attaquant exploite en permanence – est que, s'il existe des techniques antivirales relativement efficaces, elles ne sont pratiquement jamais mise en œuvre dans les produits grand public. Ces techniques sont en effet consommatrices de ressources (calcul et/ou mémoire). Elles ne sont donc pas viables d'un point de vue commercial. Cet aspect-là sera exploité plus avant dans le chapitre 8 avec la notion de τ-obfuscation.

Exercices

1. Dans une stratégie de détection, trois tests $\mathcal{T}_1, \mathcal{T}_2$ et $\mathcal{T}_3$ sont appliqués par le détecteur D. Chacun d'entre eux est entâché d'une erreur de première espèce de probabilité α_i et d'une erreur de seconde espèce de probabilité β_i ($i = 1, 2, 3$). Pour chaque fichier soumis à l'analyse antivirale, le détecteur D décide selon le maximum de vraisemblance (principe de la

majorité). Indiquez quelle est la fonctions booléenne de détection globale utilisée dans cette stratégie de détection. Calculez les probabilités d'erreur résiduelle β (probabilité de non détection résiduelle) et d'erreur résiduelle α (probabilité de fausse alarme résiduelle). Concluez sur la qualité de ce détecteur. Trouvez une meilleure fonction booléenne réduisant ces probabilités d'erreurs, différente de la fonction ET.

2. En utilisant les considérations de la section 2.3.2, concernant l'évaluation de la probabilité p_0 utilisée dans la section 3.4.1, montrez que pour un risque α donné, les résultats de la détection en terme de fausses alarmes peuvent être très différents en pratique. Pour quelle valeur de seuil de décision a-t-on $\alpha = p_0$? Concluez.
3. On considère pour le modèle de la section 3.4.1 un seuil empirique de 1. Calculez, quelle est la valeur de α. À partir de quelle probabilité p_0 réelle, ce seuil n'est-il plus suffisant ? Concluez.
4. Décrivez le test statistique modélisant le filtrage de données chiffrées ou compressées, décrit dans la section 3.6.4.
5. Montrez que l'algorithme de simulation d'une donnée anodine pour des données chiffrées, proposé en section 3.6.4, peut être détecté par un test approprié T. Modifiez ensuite l'algorithme de simulation afin de contourner également le test T.
6. Sur un grand nombre d'images au format JPEG, établissez la redondance moyenne, en considérant un alphabet de $n = 256$ symboles. Implémentez la technique décrite dans la section 3.6.4 pour dissimuler à la détection des données chiffrées.
7. Démontrez que le critère $E[\delta_1 + \delta_2] = 0$ défini dans la section 3.6.5 est plus restrictif que le critère défini par l'équation (3.12).
8. Généralisez l'équation (3.12) pour un nombre quelconque d'instructions.
9. Étudiez en détail le code du moteur métamorphe du virus *Win32/Linux.MetaPHOR* [132] et plus particulièrement les techniques de réduction ou d'expansion de code. Une fois le virus compilé, analysez les binaires produits d'une mutation à un autre. Définissez deux spectres $spec_1$ et $spec_2$ tels que d'une mutation à une autre :
   - le test relatif à $spec_1$ conduise à toujours accepter l'hypothèse nulle $\mathcal{H}_0$;
   - le test relatif à $spec_2$ conduise à fréquemment rejeter l'hypothèse nulle $\mathcal{H}_0$.

   Imaginez ensuite comment modifier le code de *Win32/Linux.MetaPHOR* pour simuler le test relatif au spectre $spec_2$.
10. Étudiez le document [125] puis appliquez la méthode de simulabilité présentée pour simuler un code chiffré en code binaire clair.

Chapitre 4

Les virus k-aires ou virus combinés

4.1 Introduction

Dans le chapitre précédent, la vision statistique qui a été présentée permet de modéliser, de décrire et d'analyser les techniques de détection antivirale, et ce, de manière globale. Ainsi, chacune d'entre elles peut être modélisée par un test statistique. Or, l'utilisation des probabilités et des statistiques n'a de sens que si les données d'échantillonnage sont en nombre suffisant. La plupart des résultats théoriques ne sont applicables – et de là les techniques statistiques connues – qu'en vertu des théorèmes centraux limites, issus de la loi dite des grands nombres. Ces résultats établissent le fait que pour telles ou telles conditions, les caractéristiques moyennes d'un grand nombre d'expériences tendent vers certaines constantes.

Ces théorèmes fondamentaux de la théorie des probabilités s'appuient sur deux résultats essentiels[1]. Le premier, connu sous le nom d'*inégalité de Bienaymé-Tchébychev* sert de lemme fondamental pour la démonstration de tous les théorèmes relevant de la loi des grands nombres.

Lemme 4.1 *(Inégalité de Bienaymé-Tchébychev) Soit X une variable aléatoire d'espérance mathématique μ et de variance σ^2. Alors, pour tout nombre réel positif ϵ, la probabilité pour que X s'écarte de son espérance mathématique d'une grandeur non inférieure à ϵ, est telle que :*

$$P[(|X - \mu| \geq \epsilon] \leq \frac{\sigma^2}{\epsilon^2} = \left(\frac{\sigma}{\epsilon}\right)^2.$$

Il est intéressant de noter que cette inégalité ne donne que la limite supérieure de la probabilité de l'écart concerné. En revanche, rien ne permet de modéliser

[1] Nous n'en donnerons pas la démonstration. Le lecteur pourra consulter [137, chapitre 13].

ce qui se passe au-delà de cet écart.

Le second résultat est une version simple du théorème de la loi des grands nombres, dit *théorème de Tchébychev.*

Théorème 4.1 *Pour un nombre d'expériences* **indépendantes suffisamment grand** n*, la moyenne arithmétique des valeurs observées d'une variable aléatoire X converge en probabilité vers son espérance mathématique μ. Autrement dit, quels que soient les nombres réels positifs ϵ et δ, nous avons :*

$$P\left[\left|\frac{\sum_{i=1}^{n} X_i}{n} - \mu\right| < \epsilon\right] > 1 - \delta.$$

Ce théorème fondamental montre que la moyenne arithmétique est une variable aléatoire (donc imprévisible), de variance aussi petite que l'on veut et que si le nombre d'expériences est suffisamment grand, cette moyenne arithmétique se comporte à peu près comme une grandeur non aléatoire. Ce résultat a été généralisé aux cas les plus compliqués où la loi de probabilité de la variable X change d'une expérience à une autre. L'existence de ces résultats fondamentaux permet de bâtir la théorie des probabilités, et à partir de là, de construire des tests de décision. Ces tests reposent en effet au minimum sur la loi de l'hypothèse nulle, laquelle ne peut être établie que si les résultats fondamentaux sont applicables.

Le lecteur notera donc, en vertu de ces résultats, que si l'on dispose de données en grand nombre et si ces dernières sont indépendantes, alors nous pouvons décider, et en particulier, les techniques de détection peuvent être appliquées selon notre modèle, présenté dans le chapitre précédent. Nous allons considérer alors la situation inverse, celle de l'attaquant. S'il parvient à faire en sorte que le nombre d'observations soit suffisamment petit et/ou que les données ne soient pas indépendantes, alors les techniques antivirales utilisées de nos jours ne sont plus valides. Les lois de probabilités qui les sous-tendent n'étant plus vérifiées, les tests qui en découlent ne sont plus valables.

Dans ce chapitre, nous allons présenter une nouvelle catégorie de virus appelés virus[2] k-aires, ou *virus combinés.* Ces virus, au lieu d'être constitués d'un code fait d'une seule pièce, sont au contraire formés de k parties agissant de concert, selon différents modes possibles. Il est essentiel de noter que parmi les k parties constitutives du code, toutes ne sont pas forcément immédiatement exécutables. Il peut s'agir de fichiers inertes, lorsque pris isolément. Il importe qu'au moins une partie soit exécutable. Il n'existe pratiquement pas d'exemple connu pour cette catégorie de virus, si ce n'est pour la forme la plus triviale. Or ces codes représentent assurément une menace que les antivirus actuels sont incapables d'appréhender.

Rappelons la nomenclature qui est donnée dans [38, chapitre 4] et que nous allons détailler dans le présent chapitre. Un virus binaire ($k = 2$) V est composé

[2] La notion de virus k-aires se généralise sans difficulté à tous les codes malveillants qu'ils soient simples ou auto-reproducteurs. Nous nous limiterons, par souci de simplicité, aux codes auto-reproducteurs.

de deux parties V_1 et V_2, chacune ayant une action virale (infection et charge finale) partielle et surtout anodine. Le virus n'est alors véritablement efficace que lors de l'action conjointe des deux virus V_1 et V_2. Deux grandes catégories de virus binaires peuvent être considérées :

- l'action de V_1 et V_2 est séquentielle. Généralement, V_1 active V_2. C'est le cas des virus *Ymun* [38, chapitre 13] et du virus *Perrun* [46]. L'avantage est que l'infection par le virus V_2 peut se faire via un format de fichier normalement considéré comme inerte (fichiers image ou son, texte chiffré...). Cela impose au virus V_1 d'être résident. Ce mode est dénommé mode série ou séquentiel. Il est généralement utilisé dans un but d'évasion de processus[3]. Autrement dit, afin de leurrer l'antivirus, l'exécution vas sauter d'un processus à un autre ;
- l'action de V_1 et V_2 est parallèle, autrement dit, les deux virus sont activés indépendamment l'un de l'autre et doivent par conséquent être tous deux résidents. Les deux virus combinent ensuite leurs actions respectives. Ce mode parallèle est généralement utilisé pour réaliser des étalements de processus[4].

Quel est l'intérêt des virus k-aires ? Et pourquoi sont-ils susceptibles d'échapper à la détection aussi facilement ? En divisant le code viral en plusieurs morceaux, et en répartissant l'action virale au niveau de chacun d'eux, avec une implémentation convenable, les deux principes précédents – taille des données et indépendance – ne sont plus respectés. Chacune des k parties ne contient pas suffisamment de données pour permettre la détection, et la dépendance, pour certains modes, met en défaut les modèles statistiques classiques. Les informations sont non seulement disséminées mais elles sont également fortement corrélées.

4.2 Formalisation théorique

Nous allons maintenant voir comment modéliser la classe des virus k-aires. Le modèle de référence, le modèle de Cohen [38, chapitre 3] ne permet pas de décrire rigoureusement ces virus. Rappelons la définition d'un virus selon Fred Cohen [26].

Définition 4.1 *Un virus est une séquence de symboles qui, interprétée dans un environnement donné (adéquat), modifie d'autres séquences de symboles dans cet environnement, de manière à y inclure une copie de lui-même, cette copie ayant éventuellement évolué.*

Dans le modèle de Cohen, la séquence de symboles est supposé unique. De fait, l'outil mathématique utilisé est une machine de Turing. La bande de calcul

[3] Le terme d'évasion a été choisi par les similitudes qu'il présente avec la technique d'évasion de fréquences en télécommunications.

[4] Le terme d'étalement a été choisi par les similitudes qu'il présente avec la technique d'étalement de spectre en télécommunications.

contient, à l'état initial, un code donné, lequel est ensuite interprété par la fonction de contrôle et la tête de lecture. Mais lorsque plusieurs codes différents interviennent, les machines de Turing simples ne sont plus adaptées. Cependant, le modèle de Turing peut être partiellement généralisé à l'aide de k-machines de Turing (machines de Turing à bandes multiples, au nombre de k). Le modèle résultant serait toutefois plus complexe à décrire et à manipuler. Le théorème suivant permet de définir précisément les limites de cette généralisation. Nous allons considérer les virus comme modèle de calcul. Nous supposons qu'un virus représente un algorithme de résolution d'un problème donné, par exemple rechercher une clef de chiffrement (voir section 8.6.2, par exemple). Un tel virus peut être décrit par une machine de Turing simple (modèle de Cohen).

Théorème 4.2 *Soit une k-machine de Turing travaillant sur une donnée de taille n en temps $f(n)$. Il est alors possible de construire une machine de Turing simple M' travaillant en temps $\mathcal{O}(f(n)^2)$ et telle que, pour toute entrée x, $M(x) = M'(x)$.*

Nous ne donnerons pas la preuve ici. Le lecteur pourra consulter [99, pp. 30-31]. Ce résultat montre que considérer des k-machines de Turing ne permet qu'une généralisation partielle. L'amélioration est limitée par un facteur quadratique. Or, si nous utilisons un grand nombre de virus en parallèle pour résoudre notre calcul, le gain en complexité sera supérieur. Les k-machines de Turing ne sont donc pas adaptées.

Pour décrire de manière formelle les virus k-aires, nous allons considérer des fonctions booléennes et des réseaux d'automates. La notion de familles de circuits booléens a déjà été utilisée pour modéliser la notion de calcul parallèle [99, chapitre 15] mais dans le cas des virus k-aires, cet outil ne convient pas. En effet, chaque circuit est supposé réaliser une tâche particulière. En d'autres termes, chacune des k parties d'un virus k-aire est supposée être de nature immédiatement exécutable, ce qui n'est pas systématiquement le cas. L'utilisation de fonctions booléennes et de réseaux d'automates va se révéler plus pratique.

Donnons tout d'abord une définition plus précise de la notion de virus k-aires.

Définition 4.2 *Un virus* k-aire *est une famille de k fichiers (non nécessairement tous exécutables) dont la réunion constitue un virus et réalise une action finale équivalente à celle d'un virus. Un virus k-aire est dit* de type série *si les k fichiers agissent l'un après l'autre et sont tous de nature exécutable à un moment donné ou bien s'il est formé d'une partie de taille $k' < k$ de fichiers exécutables et de $k - k'$ fichiers non exécutables. Il est dit* de type parallèle *si les k fichiers sont tous exécutables et agissent simultanément.*

Notons qu'il est possible de conjuguer les deux types. Les virus seront alors dits de type *série/parallèle.*

4.2.1 Concepts préliminaires

La plupart des codes auto-reproducteurs actuels sont des vers et, à ce titre, une seule copie de leur code est présente dans une machine donnée. Cela ne correspond bien sûr pas au cas des virus, dont la fonction est d'infecter plusieurs cibles dans le système (et/ou en mémoire). Ils existent donc en plusieurs exemplaires. Afin de traiter tous les cas, tout en restant simple, nous allons considérer que toutes les copies d'un code auto-reproducteur constituent un seul et même fichier, soit le fichier viral. Dans le cas d'un virus k-aire, rappelons que le fichier viral peut être la réunion de k fichiers (mais ce n'est pas toujours le cas).

D'un point de vue formel, cela revient à considérer la relation d'équivalence suivante sur un ensemble S (le système de fichiers).

Définition 4.3 *(Relation d'infection) Soient deux fichiers x_1 et x_2 et soit un virus v donné. On définit alors la relation d'équivalence $\mathcal{R}_v$ par*

$$x_1 \mathcal{R}_v x_2 \text{ si } x_1 \cap x_2 \in \{x_1, x_2, v\}.$$

Le lecteur vérifiera que la relation $\mathcal{R}_v$ est bien une relation d'équivalence. La classe d'équivalence d'un élément x est alors l'ensemble $C(x) = \{y | y \in S \text{ et } x\mathcal{R}_v y\}$. Aucune classe n'est vide car $x \in C(x)$. La classe $C(v)$ contient tous les fichiers infectés par le virus v. Toutes les classes qui sont des singletons contiennent un fichier non infecté par v.

Dans ce qui suit, on considérera alors l'ensemble quotient suivant la relation $\mathcal{R}_v$.

Définition 4.4 *Soit une relation d'équivalence $\mathcal{R}_v$ relativement à un virus v définie sur un ensemble non vide S. Le sous-ensemble de S constitué des classes d'équivalence suivant $\mathcal{R}_v$ s'appelle l'ensemble quotient de S par $\mathcal{R}_v$ et se note $S_{/\mathcal{R}_v}$. Il constitue une partition de l'ensemble S.*

Ainsi, une fois un fichier infecté, il est considéré comme « équivalent » au virus v, ce qui est logique puisqu'il propage lui-même l'infection. Nous ne préciserons plus le passage à l'ensemble quotient, qui sera dès lors implicite, sauf quand cela est strictement nécessaire.

4.2.2 Modélisation par fonctions booléennes

Soit un système (de fichiers) composé de n fichiers. Ce nombre est fini mais arbitrairement grand. Il décrit tous les fichiers existants dans le système à un moment donné. À chaque fichier possible, on attribue une variable booléenne x_i, $i = 1, 2, \ldots, n$. Aucune distinction n'est faite concernant la nature de ces fichiers. Données inertes ou exécutables sont considérées de la même manière. Nous aurons $x_i = 1$ si le fichier i est considéré, sinon $x_i = 0$. En particulier, si le fichier i n'existe pas à un instant donné, alors $x_i = 0$ et en toute rigueur

on devrait considérer un ensemble de $n-1$ variables. Mais il est plus pratique de prendre n arbitrairement grand et d'envisager simultanément tous les fichiers possibles. Quand certains fichiers $i_{j_1}, i_{j_2}, \ldots, j_{i_m}$ sont non existants, les fonctions booléennes qui seront définies sur $f_T : \mathbb{F}_2^n$ seront dégénérées en les variables correspondantes.

Deux fonctions booléennes sont alors définies : la *fonction de transition* et la *fonction d'infection*. Toutes deux décrivent les relations existant entre les fichiers du système à un instant donné : soit en tant que l'une des k parties d'un virus k-aire soit comme un fichier quelconque.

Fonction de transition

La fonction de transition est une fonction booléenne vectorielle, noté $F^t : \mathbb{F}_2^n \to \mathbb{F}_2^n$. Elle décrit les interrelations entre fichiers du système de l'instant $t-1$ à l'instant t. En particulier, dans le cas d'un virus k-aire, impliquant les fichiers $i_{j_1}, i_{j_2}, \ldots, j_{i_k}$, la fonction va décrire les interactions existant entre ces k fichiers d'une part, et celles existant entre ces fichiers et les autres. Sa structure dépend donc du mode d'action des différentes parties du virus entre elles. Notons qu'il est pratique de considérer la restriction F_v^t de F^t au sous-ensemble de S composé des k parties du virus. Cette restriction permet de se concentrer sur les interactions existant entre ces k fichiers, à l'exclusion de celles avec le reste des fichiers du système.

La fonction F^t peut être représentée de plusieurs manières. La première consiste à donner sa table de vérité. Considérons le cas $n = 3$ et les trois variables booléennes associées x_1, x_2, x_3. Alors une table de vérité peut être :

x_3	x_2	x_1	$F_3^t(x_3, x_2, x_1)$	$F_2^t(x_3, x_2, x_1)$	$F_1^t(x_3, x_2, x_1)$
0	0	0	0	0	1
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	0	1
1	1	0	0	1	0
1	1	1	1	0	0

Cette table est la *table de transition globale* du système. Les composantes de la fonction F^t sont les fonctions coordonnées F_1^t, F_2^t, F_3^t. Ainsi $F^t = (F_3^t, F_2^t, F_1^t)$. Il est alors possible par des règles simples de calcul booléen d'établir la forme

disjonctive normale de ces fonctions[5]. Ainsi, nous avons

$$\begin{aligned} F_1^t(x_3, x_2, x_1) &= x_2 x_3 \\ F_2^t(x_3, x_2, x_1) &= x_1 \overline{x_3} \\ F_1^t(x_3, x_2, x_1) &= \overline{x_2} \end{aligned}$$

Cette représentation permet de voir, par exemple, que le fichier 1 n'intervient pas sur le fichier 3. Plus généralement, $F_i^t(x_3, x_2, x_1)$ ne dépend réellement que des variables (fichiers) indexées par j correspondant à des fichiers influençant réellement le fichier i. La fonction est donc la fonction de transition globale du système, la notion de transition correspondant en fait, ici, à une infection. Une dernière représentation est la *matrice d'incidence* $I(F^t)$ associée à la fonction F_t. Elle code d'une manière compacte l'information contenue dans les deux précédentes. C'est une matrice booléenne $(3, 3)$. Ainsi, pour la fonction F^t précédente, nous avons

$$I(F^t) = \begin{vmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{vmatrix}$$

Plus généralement, la matrice d'incidence $I(F^t)$ est une matrice carrée (n, n) définie par :

$$\begin{aligned} a_{ij} &= 1 \text{ si le fichier } i \text{ agit sur le fichier } j, \\ a_{ij} &= 0 \text{ sinon.} \end{aligned}$$

L'information contenue dans la matrice d'incidence $I(F^t)$ peut être représentée graphiquement par le *graphe de connexion* de la fonction F^t.

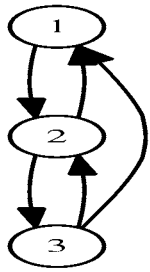


Figure 4.1 – Graphe de transition d'un réseau d'automates

Ce graphe décrit, comme la matrice d'incidence, les interactions entre les différents fichiers mais l'information apportée est assez grossière. Elle est du

[5] Cette forme a été définie dans le chapitre 2. Il est possible de calculer également la forme algébrique normale de chacune de ces fonctions au moyen de la transformée de Möbius [35].

type tout ou rien : on sait seulement si un fichier donné intervient ou pas sur un autre. C'est pourquoi, afin d'avoir quelquefois plus d'informations, on considère une autre façon de décrire le système avec le *graphe d'itération*. Il est obtenu en considérant l'itération discrète en fonction du temps t de la fonction F^t définie de la manière suivante :

$$\begin{cases} x^0 \in \mathbb{F}_2^n \\ x^{t+1} = F^t(x^t) \quad t = 0, 1, 2, \ldots \end{cases}$$

Cela donne le graphe d'itération suivant présenté en figure 4.2, les points étant représentés par la valeur décimale correspondant à chaque triplet (x_3, x_2, x_1).

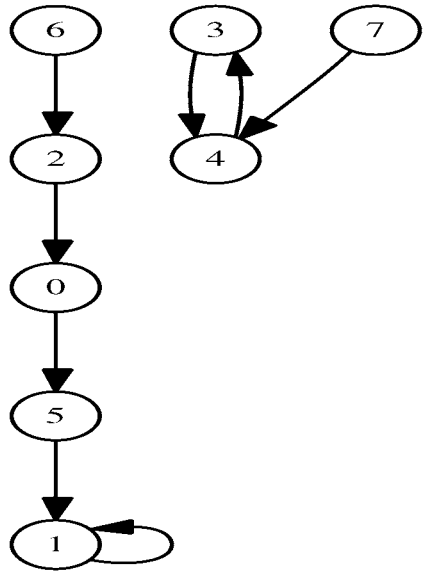


Figure 4.2 – Graphe d'itération d'un réseau d'automates

Le graphe d'itération décrit l'évolution du système S à chaque instant t. Il facilite l'étude de la dynamique du système et les relations dans le temps des différentes parties composant un virus k-aire. En particulier, nous noterons la présence d'un *point fixe* (ou état stable) constitué de l'élément 1 et en partant de l'élément 6 on rejoint l'état stable au bout de $t = 4$ pas. En outre, ce graphe se décompose en deux composantes connexes, chacune étant dotée d'un attracteur (point fixe ou cycle) : point fixe 1 ou cycle $(3 \leftrightarrow 4)$.

Les exemples que nous venons de présenter concernent une évolution en parallèle des différents composants du système. À chaque instant t, chacun de ces éléments évolue en fonction des fonctions coordonnées F_i^t. Avec $x^t =$

(x_1^t, x_2^t, x_3^t), nous avons :

$$\begin{aligned} x_1^{t+1} &= F_1^t(x_1^t, x_2^t, x_3^t) \\ x_2^{t+1} &= F_2^t(x_1^t, x_2^t, x_3^t) \\ x_3^{t+1} &= F_3^t(x_1^t, x_2^t, x_3^t) \end{aligned}$$

Mais si l'on fait évoluer le système en série (les x_i évoluent selon un ordre donné, par exemple x_1 puis x_2 et enfin x_3), nous avons alors le mode d'itération suivant :

$$\begin{aligned} x_1^{t+1} &= F_1^t(x_1^t, x_2^t, x_3^t) \\ x_2^{t+1} &= F_2^t(x_1^{t+1}, x_2^t, x_3^t) \\ x_3^{t+1} &= F_3^t(x_1^{t+1}, x_2^{t+1}, x_3^t) \end{aligned}$$

Nous utiliserons ces différents outils et représentations pour caractériser les principaux types de virus k-aires.

Fonction d'infection

La *fonction d'infection* sert, quant à elle, à décrire la constitution du virus et notamment sa composition en plusieurs parties. Cette fonction correspond en fait à la fonction de détection et indique si le virus est présent ou pas dans le système. Elle permet de décrire également comment est répartie l'information virale dans le virus.

La fonction d'infection est donc définie par $f_v : \mathbb{F}_2^n \to \mathbb{F}_2^n$. Pour $x \in \mathbb{F}_2^n$, le système est infecté par le virus v si $f_v(x) = 1$. Le recours à la fonction d'infection peut sembler artificiel, dans la mesure où cette fonction est le plus souvent inconnue. C'est une connaissance *a posteriori*. Elle se révèle néanmoins pratique pour formaliser les choses.

Composition des deux fonctions

En pratique, il est intéressant de considérer à la fois l'évolution du système à un instant donné t et l'état d'infection réalisé ou non à cet instant. Si dans le cas des virus traditionnels (voir section suivante), l'évolution coïncide avec l'infection, ce n'est pas forcément le cas avec les virus k-aires, la présence d'un sous-ensemble propre des k parties constitutives d'un tel virus ne signifie pas nécessairement que le système S est infecté. Un premier exemple trivial est celui des virus de code source. Ainsi, la présence d'un code source de virus ne constitue pas une infection de S. En revanche, le couple compilateur/code source résulte en l'infection. Le cas des virus compagnon (voir [38, chapitre 9]) peut également être vu comme un cas particulier de codes k-aires.

D'un point de vue mathématique, nous considérerons la composition fonctionnelle des fonctions F^t et f_v, soit $f_v \circ F^t$. Alors, s'il existe $x \in \mathbb{F}_2^n$ tel que $(f_v \circ F^t)(x) = f_v(F^t(x)) = 1$, nous dirons que x a infecté le système S à l'instant t.

Ce modèle peut être étendu aux vers (lesquels ne sont que des virus de réseau [38, chapitres 5 et 9]). Le nombre n décrit alors à la fois les machines dans un réseau et les fichiers présents dans chaque machine.

Nous allons maintenant utiliser ces outils pour étudier les virus traditionnels ($k = 1$) et montrer qu'ils sont parfaitement décrits par notre modèle. Ensuite, nous étudierons les principaux types de virus k-aires identifiés.

Convention 4.1 *Dans un système réel S, les interactions existant naturellement entre les différents fichiers sont très nombreuses : création de fichiers, suppression de fichiers... Décrire toutes ces interactions serait d'une complexité telle qu'il serait impossible de le faire ici. Sans perte de généralité, et pour simplifier le propos, ces interactions, dans ce qui suit, ne seront pas prises en compte et nous nommerons « modèle simplifié » le modèle sans les interactions entre fichiers non viraux. En revanche, chaque fois que cela sera pertinent, l'existence de ces interactions sera évoquée (modèle généralisé), ne serait-ce que pour préciser certains résultats théoriques obtenus à partir du modèle simplifié (sans les interactions entre les fichiers non viraux).*

4.2.3 Cas des virus traditionnels (modèle de Cohen)

Nous considérerons les deux cas principaux : celui d'un virus simple et celui d'un virus polymorphe. Le premier correspond, selon le modèle de Cohen à un ensemble viral de type singleton, alors que le second correspond à la notion de plus grand ensemble viral du modèle de Cohen, dans lequel chaque virus est une forme évoluée d'un autre virus de l'ensemble [38, §3.2.3]. Cependant, chacun d'eux réalise une infection à lui seul (donc $k = 1$).

Cas d'un virus simple

Nous allons raisonner sur un ensemble de trois fichiers : x_1, x_2 et x_3. Le virus v correspondra au fichier x_1, le fichier x_2 sera un exécutable cible pour v et le fichier x_3 sera insensible à l'action de v (fichier texte par exemple). Cela nous permettra de décrire tous les cas d'interactions possibles. Rappelons que nous travaillons sur la structure quotient relativement à la relation d'équivalence $\mathcal{R}_v$, définie dans la section 4.2.1. Autrement dit, nous ne faisons pas de distinction entre v et un fichier infecté par v. Cela revient à considérer la composition fonctionnelle suivante :

$$S \xrightarrow{F^t} S \xrightarrow{\mathcal{R}_v} S_{/\mathcal{R}_v}.$$

L'infection du fichier x_2 par $x_1 = v$ sera alors représentée par la suite

$$(0,1,1) \xmapsto{F^t} (0,1,1) \xmapsto{\mathcal{R}_v} (0,0,1).$$

Les fonctions de transition F^t et d'infection f_{x_1} sont données dans le tableau 4.1.

L'étude du graphe d'itération permet de formuler la proposition suivante.

x_3	x_2	x_1	$F_3^t(x_3,x_2,x_1)$	$F_2^t(x_3,x_2,x_1)$	$F_1^t(x_3,x_2,x_1)$	$f_{x_1}(x_3,x_2,x_1)$
0	0	0	0	0	0	0
0	0	1	0	0	1	1
0	1	0	0	1	0	0
0	1	1	0	0	1	1
1	0	0	1	0	0	0
1	0	1	1	0	1	1
1	1	0	1	1	0	0
1	1	1	1	0	1	1

Table 4.1 – Fonctions de transition et d'infection d'un virus simple

Proposition 4.1 *Soit v un virus simple, décrit par le point e_i ($x_i = 1$ et $x_j = 0$ pour $j \neq i$), et S un système de n fichiers dont m sont non infectables par le virus v. Alors le graphe d'itération de l'infection de S par v comporte :*

- *2^{n-1} composantes connexes, chacune réduite à un point fixe ;*
- *une composante connexe, dénommée composante virale, contenant le point fixe e_i et $2^m - 1$ autres points ayant tous le point e_i comme successeur ;*
- *$n - m - 1$ composantes connexes comprenant chacune un unique point fixe.*

Preuve.
Soit $x \in \mathbb{F}_2^n$. Désignons par $\text{supp}(x)$ l'ensemble $\{i \leq n | x_i = 1\}$. Supposons que $e_j = v$. Alors trois cas sont à considérer :

- si $j \notin \text{supp}(x)$, nous avons $F^t(x) = x$. Il y a, par construction 2^{n-1} points x de ce type ;
- si $j \in \text{supp}(x)$ et les k tels que $k \in \text{supp}(x)$ concernent seulement un fichier infectable, alors nous avons $F^t(x) = e_j$. Il y a, par construction $2^m - 1$ n-uplets de ce type ;
- les autres cas sont ceux pour lesquels nous avons $j \in \text{supp}(x)$ et des k tels que $k \in \text{supp}(x)$ avec x_k désignant un fichier non infectable par v. Soient deux n-uplets x et y différents, décrivant cette situation. On a alors soit $F^{t+1}(x) = y$ si $y \prec x$ (c'est-à-dire $\text{supp}(y) \subsetneq \text{supp}(x)$) ou soit $F^{t+1}(x) = x$ et c'est un élément minimal pour la relation d'ordre $\prec$. Il y a $n - m - 1$ éléments minimaux pour ce troisième groupe de cas. Comme une composante connexe ne peut contenir qu'un seul point fixe [113, pp. 20, proposition 1.1], nous avons le résultat.

■

Le graphe d'itération (figure 4.3) décrit en fait l'évolution du système et des fichiers à chaque instant. Si l'on considère cette fois le graphe de transition et la matrice d'incidence, une vision différente du système est privilégiée. On s'intéresse alors aux diverses influences des variables d'entrée (les fichiers) sur la sortie de la fonction. En d'autres termes, quels fichiers (éventuellement viraux)

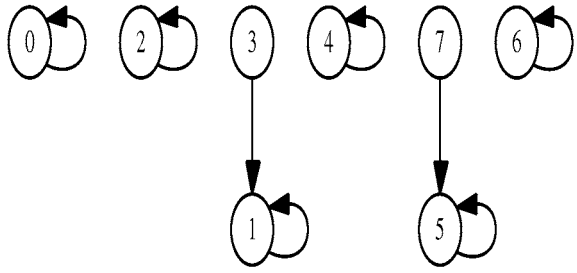


Figure 4.3 – Graphe d'itération pour un virus simple

vont influencer ou agir sur les autres ? Nous avons vu que la matrice d'incidence et le graphe de transition désignent la même chose. Ils sont tous deux établis à partir des fonctions coordonnées F_1^t, F_2^t, F_3^t. Dans le cas de notre virus simple précédent, la matrice d'incidence est alors :

$$I(F^t) = \begin{vmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{vmatrix}$$

ce qui correspond au graphe de transition suivant de la figure 4.4. Nous pouvons

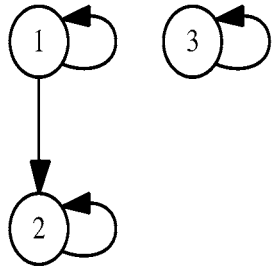


Figure 4.4 – Graphe de transition pour un virus simple

alors établir la proposition suivante.

Proposition 4.2 *Soit v un virus simple, décrit par le point e_i ($x_i = 1$ et $x_j = 0$ pour $j \neq i$), et S un système de n fichiers dont m sont non infectables par le virus v. Alors la matrice d'incidence correspondant à l'infection de S par v comporte :*

- *$m + 1$ lignes de poids 1 (une seule entrée non nulle) ;*
- *$n - m - 1$ lignes de poids 2 ;*
- *$n - 1$ colonnes de poids 1 ;*

- *une colonne de poids $n-m$, dénommée* colonne d'infection, *rassemblant le virus e_i et les cibles x_j pour ce virus.*

Preuve.
La preuve s'établit facilement en considérant que les fonctions coordonnées F_i^t (représentées par les lignes de la matrice) relatives aux fichiers non infectables ne dépendent, par construction, que de ces fichiers. Il y a donc m lignes de poids 1 auxquelles il faut ajouter celle correspondant au virus lui-même. En revanche, les fonctions coordonnées relatives aux fichiers infectables dépendent à la fois du virus et du fichier. Il y en a $n-m-1$.

Pour les colonnes, la preuve est la même en considérant cette fois non plus une relation de dépendance mais l'action que le fichier i a ou n'a pas sur la fonction coordonnée F_j^t. La seule colonne ayant plus d'une entrée non nulle est celle décrivant l'action du virus e_i. ■

La proposition 4.2 est intéressante car elle montre que, selon ce modèle, rechercher un virus simple par ses interactions avec les fichiers du système, en disposant de la matrice d'incidence, a une complexité en $\mathcal{O}(n)$ (évaluation du poids de n colonnes). Malheureusement, si la complexité est linéaire, la taille de la donnée d'entrée est quadratique et peut être très importante. Ainsi si le système S contient $n = 20000$ fichiers, la matrice d'incidence a une taille 4.10^8 octets, soit environ $0,4$ Go. Des techniques d'échantillonnage peuvent certainement permettre de réduire la complexité finale. D'autre part, construire la matrice suppose de tester l'action d'un fichier exécutable sur un autre, ce qui peut être fait par des techniques d'émulation. S'agissant des virus simples, les techniques traditionnelles restent les plus efficaces pour une détection courante par un antivirus du commerce. Toutefois, pour une première identification (cas de virus inconnus), sur un système de banc-test, ce modèle peut fournir une méthode de détection intéressante.

Nous allons maintenant étudier la fonction de transition sous un angle différent qui va nous permettre d'avoir une vision statistique de l'infection. Cette vision peut, entres autres choses, permettre de mettre en œuvre la détection par échantillonnage, évoquée dans le paragraphe précédent. En effet, ce modèle sous sa forme actuelle considère un système S générique. En particulier, on ne fait aucune supposition sur le nombre de fichiers infectables par v ou sur ceux qui ne le sont pas. Il n'est donc pas possible de décrire de manière plus détaillée l'action de v. À moins de connaître en totalité la fonction de transition F^t – ce qui peut ne pas être possible d'un point de vue de la complexité en temps et en mémoire, soit $\mathcal{O}(2^n)$ –, il est nécessaire de disposer d'une autre description concernant l'action de v.

Proposition 4.3 *Soit $F^t(x)$ la fonction de transition pour un virus décrit par le n-uplet e_i. Soient u et v deux éléments de $\mathbb{F}_2^n$. Désignons par c_j les fichiers infectables par e_i et n_k ceux qui ne le sont pas. Alors, dans le cas d'un virus simple, la probabilité $P[< F^t(x), u >=< x, v >]$ vaut :*

- $P[< F^t(x), u >=< x, v >] = 1$ *si* $u = v = e_i$;
- $P[< F^t(x), u >=< x, v >] = 1$ *si* $supp(u) = supp(v)$ *et si ces supports ne contiennent que l'indice* i *et des indices* k *de fichiers non infectables par le virus ;*
- $P[< F^t(x), u >=< x, v >] = \frac{1}{4}$ *si* $v = e_i$ *et si* $supp(u)$ *ne contient que des indices* j *de fichiers infectables par le virus ;*
- $P[< F^t(x), u >=< x, v >] = \frac{3}{4}$ *si* u *et* v *contiennent le virus et/ou les mêmes fichiers* c_j *infectables par le virus (autrement dit les restrictions des supports de* u *et de* v *aux fichiers* c_j *sont identiques) ;*
- $P[< F^t(x), u >=< x, v >] = \frac{1}{2}$ *dans tous les autres cas.*

où $< ., . >$ *désigne le produit scalaire usuel. Enfin, nous avons*

$$P[(f_v \circ F^t)(x) =< x, e_i >] = 1.$$

Preuve.
Les deux premiers cas sont évidents à démontrer. En effet, le virus n'étant pas infectable par lui-même (lutte contre la surinfection) et les fichiers non infectables n'étant pas affectés par l'action du virus, les entrées et les sorties de la fonction sont identiques.
Pour le troisième cas, nous avons :

- soit $i \in \text{supp}(x)$, alors dans ce cas $< F^t(x), u >= 0$. En effet, tous les $< F^t(x), c_j >= 0$ (passage au quotient),
- soit $i \notin \text{supp}(x)$ alors $< F^t(x), u >= 0$ avec une probabilité de $\frac{1}{2}$.

Nous avons alors

$$P[< F^t(x), u >=< x, v >] = P[i \in \text{supp}(x)].1 + P[i \notin \text{supp}(x)].\frac{1}{2} = \frac{1}{4}.$$

Les cas restants se démontrent de manière identique.

Le résultat concernant la probabilité $P[(f_v \circ F^t)(x) =< x, e_i >]$ est évident dans le cas des virus simples. ■

La proposition 4.3 est intéressante car elle montre que les corrélations existant entre des sous-ensembles d'entrée de la fonction F^t et un sous-ensemble de ses fonctions coordonnées sont indépendantes de la composition du système S en terme de fichiers infectables ou non infectables par le virus v. Cela permet donc de caractériser un virus simple, de manière unique, indépendamment du système.

Cas d'un virus polymorphe

Le cas des virus polymorphes/métamorphes (voir chapitre 6) correspond au plus grand ensemble viral de Fred Cohen [26] [38, chapitre 3]. On suppose cet ensemble fini (on peut en fait généraliser au cas fini dénombrable [149], voir également la section 6.1). Dans cet ensemble, chaque élément (virus) est une version évoluée (mutée) d'un autre virus de l'ensemble viral (au moins par fermerture transitive). Donc, pour une machine de Turing donnée, $\forall \{v_i, v_j\} \subset$

$(x_5, x_4, x_3, x_2, x_1)$	$F_5^t(x)$	$F_4^t(x)$	$F_3^t(x)$	$F_2^t(x)$	$F_1^t(x)$	$f_{\{x_1,x_2,x_3,x_4,x_5\}}(x)$
0 0 0 0 0	0	0	0	0	0	0
0 0 0 0 1	0	0	0	0	1	1
0 0 0 1 0	0	0	0	1	0	1
0 0 0 1 1	0	0	0	1	1	1
0 0 1 0 0	0	0	1	0	0	1
0 0 1 0 1	0	0	1	0	1	1
0 0 1 1 0	0	0	1	1	0	1
0 0 1 1 1	0	0	1	1	1	1
0 1 0 0 0	0	1	0	0	0	0
0 1 0 0 1	0	0	0	1	1	1
0 1 0 1 0	0	0	1	1	0	1
0 1 0 1 1	0	0	1	1	1	1
0 1 1 0 0	0	0	1	0	1	1
0 1 1 0 1	0	0	1	1	1	1
0 1 1 1 0	0	0	1	1	1	1
0 1 1 1 1	0	0	1	1	1	1
1 0 0 0 0	1	0	0	0	0	0
1 0 0 0 1	1	0	0	0	1	1
1 0 0 1 0	1	0	0	1	0	1
1 0 0 1 1	1	0	0	1	1	1
1 0 1 0 0	1	0	1	0	0	1
1 0 1 0 1	1	0	1	0	1	1
1 0 1 1 0	1	0	1	1	0	1
1 0 1 1 1	1	0	1	1	1	1
1 1 0 0 0	1	1	0	0	0	0
1 1 0 0 1	1	0	0	1	1	1
1 1 0 1 0	1	0	1	1	0	1
1 1 0 1 1	1	0	1	1	1	1
1 1 1 0 0	1	0	1	0	1	1
1 1 1 0 1	1	0	1	1	1	1
1 1 1 1 0	1	0	1	1	1	1
1 1 1 1 1	1	0	1	1	1	1

Table 4.2 – Fonctions de transition et d'infection d'un virus polymorphe

$PGEV(M)$, $v_i \rightarrow v_j$ ou $v_j \rightarrow v_i$. Si l'ensemble est fini, nous considérerons que les deux relations d'évaluation sont vraies.

La représentation des virus polymorphes selon le modèle booléen précédent devient un peu plus complexe. Pour des raisons de place, nous ne donnerons pas *in extenso* les différentes fonctions (cela est laissé à titre d'exercice) mais seulement les règles pour les construire.

Considérons un virus polymorphe à n formes (voir section 6.1 pour la définition formelle), que nous noterons $(x_{v_1}, x_{v_2}, \ldots, x_{v_i}, \ldots, x_{v_n})$. Considérons également un système S comprenant N fichiers dont m sont infectables par le virus (donc $N-m-n$ sont insensibles au virus). Nous noterons $(x_{f_1}, \ldots, x_{f_j}, \ldots, x_{f_m})$ les fichiers infectables par le virus et $(x_{i_1}, x_{i_2}, \ldots, x_{i_k}, \ldots, x_{i_{N-m-n}})$ ceux qui ne le sont pas.

Définissons alors les règles générales suivantes pour construire la fonction de transition F^t de ce virus, relativement au système S. Pour tout $x \in \mathbb{F}_2^n$, alors,

- si $\text{supp}(x) \cap \{v_i\} = \emptyset$, alors $F^t(x) = x$ (le virus n'est pas actif) ;
- si $\text{supp}(x) \cap \{v_i\} \neq \emptyset$ et $\text{supp}(x) \cap \{i_k\} \neq \emptyset$ et $\text{supp}(x) \cap \{f_j\} = \emptyset$, alors $F^t(x) = x$ (le virus est actif mais aucune cible n'est disponible) ;
- soit $n_1 = |\text{supp}(x) \cap \{v_i\}|$ (nombre de formes mutées représentées dans x) et $n_2 = |\text{supp}(x) \cap \{f_j\}|$ (nombre de fichiers infectables par le virus représentés dans x). Alors, on définit $y = x$, on annule les n_2 positions de y correspondant aux fichiers infectables et si $n_2 > n - n_1$, on met à 1 $n - n_1$ positions nulles de y correspondant à des formes mutées du virus (pour chaque cible une nouvelle forme mutée apparaît, à concurrence du nombre total de ces formes). Finalement on pose $F^t(x) = y$.

Notons que les différentes formes mutées du virus ne s'infectent pas les unes les autres, ce qui est le cas d'un virus bien conçu.

À titre d'illustration, considérons un virus polymorphe à trois formes que nous noterons x_1, x_2 et x_3. Considérons également un système S comprenant $n = 5$ fichiers dont un seul est infectable (noté x_4) par le virus polymorphe $\{x_1, x_2, x_3\}$. Le fichier restant, noté x_5 est non infectable. Donnons les fonctions de transition F^t et d'infection $f_{\{x_1,x_2,x_3\}}$ (tableau 4.2).

Le graphe d'itération est donné en figure 4.5 (les points fixes ont été omis par manque de place). Le graphe de transition est quant à lui donné en figure 4.6 (le calcul de la matrice d'incidence sera laissée à titre d'exercice).

Établissons un premier résultat qui peut être obtenu à partir de ce modèle.

Proposition 4.4 *Soit un système S contenant m fichiers infectables par un virus polymorphe à n formes. Supposons que $n < m$. Alors le graphe de transition correspondant à l'infection par le virus $(x_{v_1}, x_{v_2}, \ldots, x_{v_n})$ contient un sous-graphe dirigé complet*[6] *constitués des points $(e_{v_i})_{1 \leq i \leq n}$.*

[6] On appelle sous-graphe complet, ou *clique*, un sous-ensemble de sommets tous connectés deux à deux par des arêtes ou des arcs.

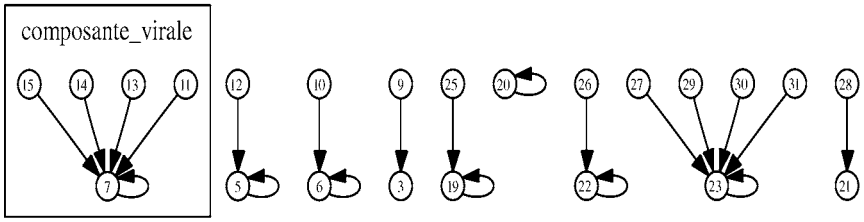


Figure 4.5 – Graphe d'itération (partiel) pour un virus polymorphe à trois formes

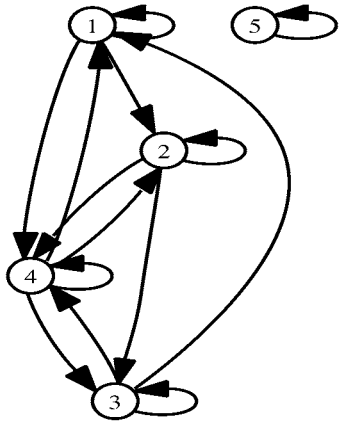


Figure 4.6 – Graphe de transition (partiel) pour un virus polymorphe à trois formes

Preuve.
On supposera, sans perte de généralité, que la mutation se fait selon le schéma $x_{v_i} \to x_{v_{i+1}}$ avec $x_{v_n} \to x_{v_1}$.

Pour démontrer cette proposition, il suffit de montrer que chaque variable x_{v_i} intervient dans la forme normale (disjonctive ou algébrique) des fonctions coordonnées F_j^t, pour j allant de 1 à v_n. Nous utiliserons la forme algébrique normale [35]. Rappelons que la forme algébrique normale (FAN) d'une fonction booléenne f à N variables est décrite par

$$f(x_1, x_2, \ldots, x_N) = \sum_{\alpha \in \mathbb{F}_2^N} a_\alpha x^\alpha \qquad a_\alpha \in \mathbb{F}_2$$

où $x = (x_1, x_2, \ldots, x_N)$ et $x^\alpha = x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_N^{\alpha_N}$ avec $\alpha_i \in \mathbb{F}_2$. Les coefficients

a_α de la FAN peuvent être obtenus grâce à la transformée de Möbius [35] :

$$a_\alpha = g(\alpha) = \sum_{\beta \preceq \alpha} f(\beta_1, \beta_2, \ldots, \beta_n),$$

où $\beta \preceq \alpha$ décrit l'ordre partiel sur le treillis des sous-ensembles de $\mathbb{F}_2^N$. Autrement dit, $\beta \preceq \alpha \Leftrightarrow \beta_i \leq \alpha_i \ \forall i$.

Une fois ces notations fixées, montrons que, quelle que soit la fonction coordonnée F_j^t, pour j allant de 1 à n, toutes les variables $x_{v_i}, 1 \leq i \leq n$ interviennent dans sa forme alébrique normale.

Considérons le n-uplet $\alpha = (\ldots, x_{f_n}, x_{f_{n-1}}, \ldots, x_{f_1}, 0, \ldots, x_{v_i}, 0, \ldots, 0, 0)$ constitué de n positions non nulles correspondant à des fichiers infectables par le virus et à une seule position non nulle correspondant à la forme mutée v_i. Montrons que le coefficient $a_\alpha = 1$ dans la FAN de F_j^t, pour j allant de 1 à n. Par construction de la fonction F^t, le nombre de $\beta \preceq \alpha$ tel que $F_j^t(\beta) = 1$ est donné par

$$\sum_{l=1}^{n} \binom{v_n}{l} = 2^n - 1.$$

Comme ce nombre est impair, nous avons bien $F_j^t(\beta) = 1$, pour tout j de 1 à v_n. Donc, quelle que soit la variable x_{v_i}, elle intervient dans la FAN de toutes les fonctions coordonnées correspondant à une forme mutée du virus. D'où le résultat. ■

Remarque. Pour la plupart des virus polymorphes, le nombre de formes mutées est beaucoup plus grand que le nombre de fichiers potentiellement infectables par ces virus. Donc le sous-graphe complet est rarement réalisé dans le graphe de transition. Néanmoins, son existence potentielle au regard de ce modèle permet d'établir, d'une manière différente de celle de D. Spinellis en 2003 [127] (voir la démonstration dans la section 6.2), la complexité de la détection des virus polymorphes.

Théorème 4.3 *Soit un système S contenant m fichiers infectables par un virus polymorphe à n formes. Supposons que $n < m$. La détection d'un virus polymorphe dans ce système fondée sur les interactions entre les différentes formes mutées est un problème NP-complet.*

Preuve.
La preuve utilise le résultat selon lequel déterminer s'il existe une clique de taille n (sous-graphe complet) dans un graphe (le graphe de transition) est un problème NP-complet [99]. Dans le cadre de notre modèle théorique simplifié (voir convention 4.1), la recherche d'une clique est une instance facile – une unique composante connexe non réduite à un point et contenant un sous-graphe complet de taille n relativement au virus $(x_{v_1}, x_{v_2}, \ldots, x_{v_n})$. Dans le cas réel (existence de très nombreuses interactions entre les différents fichiers infectables et/ou non infectables du système), il existe de nombreuses autres composantes connexes d'autre part et la composante virale n'est plus limitée nécessairement aux seuls fichiers viraux. Autrement dit, dans le cas d'un système réel, le graphe de transition est d'une complexité telle qu'il constitue, dans le cas général, une instance réellement difficile du problème de recherche d'une clique. D'où le résultat. ■

Ce résultat ne considère en aucun cas la complexité intrinsèque de la détection d'un virus polymorphe selon Spinellis, autrement dit décider si une forme mutée est issue d'une autre. Dans le cas présent, la détection concerne la détection de l'ensemble viral lui-même. La complexité globale de la détection d'un virus polymorphe est donc le minimum entre cette complexité intrinsèque et celle relevant du modèle booléen considéré dans ce chapitre.

Considérons à présent le graphe d'itération modélisant un virus polymorphe. La composition fine des différentes composantes connexes devient beaucoup plus difficile à établir, les formules obtenues étant complexes.

Proposition 4.5 *Soit un système S contenant N fichiers dont m fichiers sont infectables par un virus polymorphe à n formes. Alors le graphe d'itération de l'infection de S par ce virus comporte :*

- $2^{N-n} + 2^{N-n-m}.n$ *composantes connexes réduites à un point fixe ;*
- *une composante connexe, dénommée composante virale, contenant le point fixe* $\bigoplus_{i=1}^{n} e_{v_i}$, *successeur de tous les autres points de cette composante, dont le nombre est donné par la formule suivante :*

$$\sum_{i=0}^{n} \sum_{j=0}^{m-n+i} \binom{m}{m-j} \binom{n}{n-i},$$

- $(2^n - n - 1).2^{N-n-m} - 1$ *composantes connexes comprenant chacune un unique point fixe, dont le support contient : un sous-ensemble propre de* $v_1, v_2, \ldots, v_n$.

Preuve.
Pour les composantes connexes réduites à un point, elles sont constituées des points $x \in \mathbb{F}_2^N$ dont le support contient

- soit des positions représentant des fichiers infectables et/ou des fichiers non infectables, à l'exclusion de toute forme mutée du virus. Il y en a exactement 2^{N-n} ;

- soit des positions représentant des fichiers non infectables uniquement et une forme mutée exactement. Il y en a exactement $2^{N-n-m}.n$.

Pour le second point de la proposition, cette composante connexe contient tous les points dont l'image par la fonction F^t est le point référençant toutes les formes mutées du virus. Ces points contiennent i positions non nulles correspondant à des formes mutées du virus ($i > 0$) et de $n - i$ à m positions correspondant à des fichiers infectables par le virus. D'où la formule.

Pour le troisième point, les autres composantes connexes non réduites à un point fixe rassemblent tous les éléments $x \in \mathbb{F}_2^N$ dont l'image $y = F^t(x)$ contient dans son support un sous-ensemble propre de positions représentant des formes mutées du virus. Cette image est telle que $F^t(x) \neq x$ (puisque les composantes ne sont pas réduites à un point fixe). Il y en a exactement $2^n - n - 1$; le terme $n - 1$ prend en compte les cas où $F^t(x) = x$ avec x ne contenant uniquement au plus une forme mutée.

Pour chacune d'elles, on peut considérer un sous-ensemble quelconque de positions correspondant à des fichiers non infectables. Il y en a exactement 2^{N-n-m}. On enlève finalement la composante connexe décrite par le second point de la proposition. ■

Le lecteur remarquera que cette proposition n'explicite pas, contrairement à la proposition 4.1, le nombre de points des composantes connexes non réduites à un point fixe et différentes de la composante virale. Outre la complexité des formules les dénombrant, cela n'a qu'un intérêt très marginal. Le lecteur pourra les établir après quelques efforts.

En effet, dans la réalité, comme $n < m$, le graphe d'itération sera très incomplet et en général le nombre ν de formes mutées effectivement réalisées est égal à n. C'est la raison pour laquelle la vision statistique de la fonction $F^t(x)$ pourrait être considérée comme plus intéressante par rapport à celle, strictement combinatoire, du graphe d'itération. Malheureusement, dans le cas des virus polymorphes, les résultats de la proposition 4.3 pour les virus simples sont difficiles à établir. Différentes simulations ont été conduites et leurs résultats ont suggéré les conjectures suivantes.

Conjecture 4.1 *Soit $F^t(x)$ la fonction de transition décrivant l'infection par un virus polymorphe à n formes d'un système S contenant m fichiers infectables par ce virus. Soient u et v deux éléments de $\mathbb{F}_2^{n+m}$. Alors :*

- $P[< F^t(x), u >=< x, v >] = \frac{1}{2} + \epsilon(m)$ *où* $lim\ \epsilon(m) = 0$ *quand* $m \to \infty$;
- *si* $n < m$ *alors* $P[< F^t(x), u >=< x, v >] = \frac{1}{2} + \epsilon'(m)$ *où* $lim\ \epsilon'(m) = 0$ *quand* $m \to \infty$ *et* $\epsilon(m) - \epsilon'(m) > 0$ *quelle que soit la valeur* m ;

Dans le premier cas, les simulations ont montré que $\epsilon(m) = 0,00000095$ pour $m = 10$ alors que $\epsilon'(m) = 0,0003052$ pour $m = 10$ et $n = 5$. Dans les deux cas, pour $m > 20$, la probabilité $P[< F^t(x), u >=< x, v >]$ est pratiquement égale à $\frac{1}{2} - 10^{-9}$.

Ce résultat, vérifié et validé expérimentalement, montre tout l'intérêt des virus polymorphes par rapport aux virus simples. La proposition suivante est

cependant plus explicite si l'on considère la fonction d'infection f_v, construite de la manière suivante : pour tout x, si $\text{supp}(x) \cap \{v_i\} \neq \emptyset$ alors $f_v(x) = 1$ (la fonction vaut 1 si le support de x contient au moins une position non nulle correspondant à une forme mutée du virus).

Proposition 4.6 *Soit $F^t(x)$ la fonction de transition décrivant l'infection par un virus polymorphe à n formes d'un système S contenant m fichiers infectables par ce virus. Soit la fonction d'infection f_v. Alors, pour tout $u \in \mathbb{F}_2^{n+m}$,*

$$P[(f_v \circ F^t)(x) =< x, u >] = \frac{1}{2}.$$

Preuve.
Deux cas sont à considérer pour le calcul de cette probabilité.

Soit x est tel que $\text{supp}(x) \cap \{v_i\} = \emptyset$, alors : $< x, u >= 0$ dans un cas sur deux, $(f_v \circ F^t)(x) = 0$ et donc $P[(f_v \circ F^t)(x) =< x, u > |\text{supp}(x) \cap \{v_i\} = \emptyset] = \frac{1}{2}$.

L'autre cas est celui des x tels que $\text{supp}(x) \cap \{v_i\} \neq \emptyset$ et pour lequel $< x, u >= 0$ dans un cas sur deux, $(f_v \circ F^t)(x) = 1$ et donc $P[(f_v \circ F^t)(x) =< x, u > |\text{supp}(x) \cap \{v_i\} \neq \emptyset] = \frac{1}{2}$.

Comme le premier cas intervient avec une probabilité de $\frac{1}{2^n}$ et que le second intervient avec une probabilité de $1 - \frac{1}{2^n}$, nous avons

$$P[(f_v \circ F^t)(x) =< x, u >] = \frac{1}{2^n}.\frac{1}{2} + (1 - \frac{1}{2^n}).\frac{1}{2} = \frac{1}{2}.$$

■

Dans le cas des virus polymorphes/métamorphes, « l'information » virale n'est plus concentrée sur un seul code mais sur toutes les formes mutées de ce code. Il n'y a pas redondance au regard de la relation d'équivalence utilisée pour le quotientement. En revanche, si l'une d'entre elles est analysée, le processus de mutation est alors connu et toutes les autres formes mutées sont également connues (à la complexité calculatoire près). Nous sommes à nouveau dans la situation d'un virus simple (voir exercice). Le fait que chaque forme mutée porte la totalité de l'information constitue une faiblesse.

Ce que montre également le modèle booléen, c'est une certaine équivalence entre les virus simples et les virus polymorphes/métamorphes (voir également les exercices). Si l'on compare leur graphe d'itération respectif, il n'existe aucune chaîne de longueur supérieure à 2 (si l'on exclut les points fixes). C'est d'ailleurs la raison pour laquelle nous n'avons pas considéré véritablement les itérations de la fonction F^t. La seule différence tient au poids de Hamming des points fixes dans chaque composante connexe non réduite à un point fixe, et notamment pour la composante virale. Égale à 1 dans le cas des virus simple, elle vaut n dans le cas d'un virus polymorphe à n formes (ou m si $m < n$).

L'intérêt des virus k-aires est précisément de pouvoir scinder l'information virale et ainsi de définir des classes de virus qui ne peuvent se ramener au cas des virus traditionnels, simples ou mutants. Deux classes principales de code k-aires sont à considérer :

- *codes de classe I.* Les codes agissent de manière séquentielle (l'un après l'autre). Trois sous-classes sont alors à considérer :
  - *sous-classe A (codes séquentiels dépendants).* Chaque code fait référence ou contient une référence aux autres. C'est la classe la plus faible dans la mesure où toute détection de l'un permet la détection facile des autres ;
  - *sous-classe B (codes séquentiels indépendants).* Aucun code ne fait référence aux autres. La détection de l'un ne met pas en danger les autres qui peuvent rester actifs. Un code de remplacement peut alors être substitué à celui qui a été détecté ;
  - *sous-classe C (codes séquentiels faiblement dépendants).* La dépendance des codes n'existe que dans un seul sens.
- *codes de classe II.* Les codes agissent de manière parallèle (en même temps). Les trois sous-classes précédentes sont aussi à considérer. Les k codes doivent donc figurer simultanément dans la machine pour pouvoir agir. Cela peut constituer une faiblesse.

4.3 Codes k-aires séquentiels

Ce type de codes est constitué d'un ensemble de k codes dont chacun ne réalise qu'une partie des actions du programme malveillant, selon un mode séquentiel. Le cas optimal étant que chacune de ces k parties, prise isolément, ne représente qu'une action anodine et normale. La nature virale des actions menées ne doit alors pouvoir être établie qu'en considérant simultanément un sous-ensemble important de ces k sous-codes. Il existe très probablement un très grand nombre de configurations possibles.

L'utilisation de codes agissant en série, en vue d'une action finale, n'est pas nouvelle, même si les cas connus ne sont pas très élaborés et correspondent à des scenarii assez frustres dont la détection est aisée (en général $k = 2$). Il s'agit en général des codes 2-aires appartenant aux sous-classe A (codes séquentiels dépendants) ou C (codes séquentiels faiblement dépendants). En outre, pour les cas connus, chacune des deux parties composant le code malveillant est elle-même un code malveillant. La détection de ces codes ne pose en général aucun problème car les codes, pris isolément, sont de nature virale. C'est, par exemple, le cas du virus *Perrun* [46] ou du couple *Scob/Padodor* [42]. Le lecteur pourra également consulter la référence [30], pour un exemple de codes des sous-classes A et C, à base de macro-virus, sous OpenOffice (virus[7] *Final_ Touch*).

Nous allons ici présenter en détail un exemple de code de la classe B (codes séquentiels indépendants). À la connaissance de l'auteur, il n'existe pas d'attaque connue appartenant à cette classe. Pour évaluer et illustrer le potentiel de ces codes, un programme de type preuve de concept, dénommé POC_SERIAL

[7] Plusieurs versions de ce virus ont été développées et testées sous OpenOffice. À titre d'exemple, une partie P_1 modifie certains menus et boutons. Une seconde partie P_2 installe ensuite une ou plusieurs macros malicieuses.

a été conçu et testé. Nous allons le présenter succintement dans un premier temps puis le modèle théorique général décrivant cette catégorie de codes sera explicité.

4.3.1 Le ver POC_SERIAL

Le ver sera décrit d'une manière générique car d'une part de nombreuses variations sont possibles et il est préférable de présenter l'algorithmique générale – ce qui suffit amplement à la compréhension de la technologie présentée. D'autre part, et pour des raisons légales, il n'est pas possible de détailler les ressorts d'implémentation d'un ver pratiquement impossible à détecter.

Le ver POC_SERIAL est constituée de k parties. La valeur de k est configurable mais en pratique une valeur de $4 \leq k \leq 8$ est suffisante. Chacune de ces parties effectue une tâche bien particulière, en elle-même anodine – en d'autres termes, elle ne se distingue en rien, du point de vue fonctionnel, d'une application légitime. La charge finale de POC_SERIAL consiste à dérober un document et à le faire sortir du poste cible. D'un point de vue pratique, ce ver peut appartenir à l'une quelconque des trois classes possibles (ver simple, ver d'e-mail ou macro-ver ; voir [38, chapitre 4]).

Notons $(P_i)_{1 \leq i \leq k}$ les parties constituant le ver POC_SERIAL (noté W_k par souci de concision). On suppose que les parties de W_k agissent selon l'ordre croissant des indices : $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_k \rightarrow P_1$. Les principales caractéristiques de ce ver sont alors les suivantes :

- à un instant t, une seule partie P_i est présente dans le système. Si, pour une raison ou une autre, plusieurs parties $P_{i_1}, P_{i_2}, \ldots, P_{i_j}$ sont présentes dans le système, alors ces parties disparaissent selon la règle suivante[8] :
  1. si P_i et P_{i+1} sont simultanément présentes dans le système, alors P_i « s'autodétruit » ;
  2. si P_i est présente mais pas P_{i+1} alors P_{i+1} est créé et P_i « s'autodétruit »,

  avec la convention que $k + 1 = 1$. Cette règle est itérée dans le sens croissant des indices et jusqu'à converger vers un k-uplet de poids 1. Pour cette contrainte, quelle que soit la nature de la règle choisie, le point fondamental réside dans le fait que les parties ne doivent pas coexister dans le système ;
- pour tout $1 \leq i \leq k$, le code de P_i ne contient aucune référence au code d'un P_j ($1 \leq i \leq k$ et $i \neq j$). En d'autres terme, l'analyse de P_i ne fournit aucune information sur la nature d'une autre partie P_j et ne permet donc pas de l'identifier[9] ;

[8] Pour ce point, d'autres règles peuvent être définies. Par exemple, si deux parties P_i et P_j ($i \neq j$) ou plus sont présentes simultanément dans la machine, les différentes parties s'autodétruisent. Toutefois ce cas correspond à une instance faible relativement au problème de leur détection (voir exercices).

[9] C'est là un point essentiel pour interdire à l'analyste toute vision d'ensemble du virus. Cette contrainte peut sembler contradictoire avec le point précédent. En effet, il faut que chaque

- d'un point de vue pratique, un délai minimal $\Delta t > 0$ entre l'action de deux parties P_i et P_{i+1} doit être observé, ce afin de garantir qu'aucun antivirus ne risque de considérer simultanément ces deux parties ;
- chaque partie P_i met en œuvre une ou plusieurs techniques de *leurrage*. Ces techniques consistent à créer ou à supprimer des fichiers[10]. Nous verrons, dans le cadre des théorèmes 4.4 et 4.5, l'intérêt du leurrage.

Selon le scénario développé et la version du ver, soit l'infection est propagée de poste en poste de manière centralisée par une machine transformée en serveur (local ou global) pour l'infection, soit chaque machine est capable de propager elle-même l'infection.

x_4	x_3	x_2	x_1	$F_4^t(x)$	$F_3^t(x)$	$F_2^t(x)$	$F_1^t(x)$	$f_{\{x_1,x_2,x_3,x_4\}}(x)$
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	0	0
0	0	1	1	0	1	0	0	0
0	1	0	0	1	0	0	0	0
0	1	0	1	1	0	0	0	0
0	1	1	0	1	0	0	0	0
0	1	1	1	1	0	0	0	0
1	0	0	0	0	0	0	1	0
1	0	0	1	0	0	1	0	0
1	0	1	0	0	1	0	0	0
1	0	1	1	0	1	0	0	0
1	1	0	0	0	0	0	1	0
1	1	0	1	0	0	1	0	0
1	1	1	0	0	0	0	1	0
1	1	1	1	0	0	0	1	1

Table 4.3 – Fonctions de transition et d'infection d'un code k-aire séquentiel ($k = 4$)

4.3.2 Modèle théorique

Pour modéliser ce type de virus, nous allons considérer un ver de type POC_SERIAL à quatre parties soit $W_4 = (P_1, P_2, P_3, P_4)$. Par souci de clarté et

partie puisse détecter la présence des autres sans les connaître et sans y faire référence. C'est là une contrainte qui peut recevoir une multitude de réponses techniques opérationnelles. Une voie possible, parmi de nombreuses autres, consiste à utiliser les techniques présentées dans [109]

[10] Les fichiers supprimés par une partie P_i peuvent avoir été créés par une partie P_j avec $j < i$. Dans tous les cas, cette supression doit concerner des fichiers « non critiques » afin de ne pas consituer un facteur d'alerte au niveau des antivirus. En pratique, toutefois, la création de fichiers est préférable.

sans perte de généralité, nous ne représenterons à l'aide de variables booléennes que les quatre fichiers correspondant aux différentes parties du ver. D'un point de vue général, il suffit de considérer un plongement de $\mathbb{F}_2^4$ dans $\mathbb{F}_2^n$ pour prendre en compte tous les fichiers d'un système S. Nous nous placerons dans un premier temps dans le cadre du modèle simplifié (voir convention 4.1). Avec

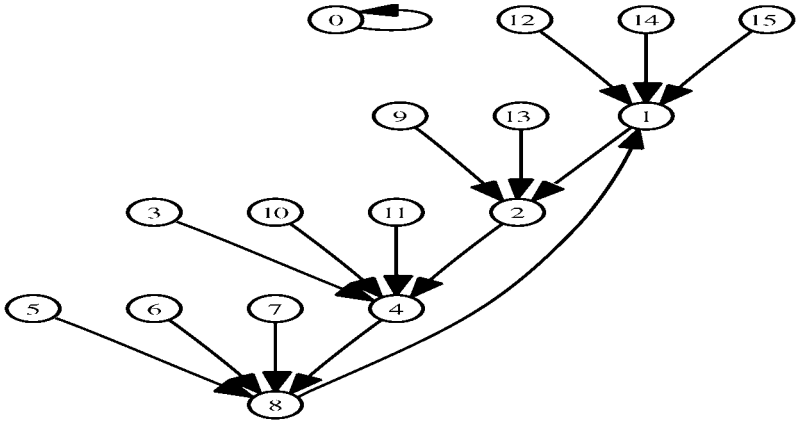


Figure 4.7 – Graphe d'itération pour le ver W_4

les contraintes imposées dans la section précédente, les fonctions de transition F^t et d'infection f_{W_4} du ver W_4 sont données dans le tableau 4.3. Les parties P_i seront représentées par les variables x_i.

Le graphe d'itération est donné en figure 4.7. Le graphe de transition est quant à lui donné par la figure 4.8 (le calcul de la matrice d'incidence sera laissée à titre d'exercice).

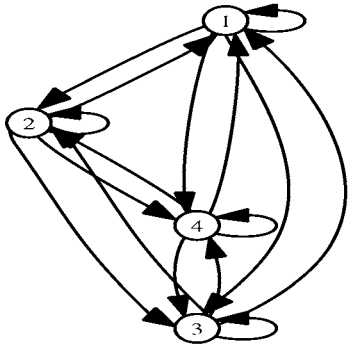


Figure 4.8 – Graphe de transition pour le ver W_4

Établissons un premier résultat qui peut être obtenu à partir de ce modèle.

Nous nous placerons dans le cas d'un ver de type POC_SERIAL pour lequel il sera considéré que le nombre de fichiers infectables est nul.

Proposition 4.7 *Soit un système S contenant N fichiers (avant infection) et soit un code W_k k-aire de type séquentiel. Alors, le graphe d'itération décrivant l'infection de S par le code W_k au cours du temps (modèle simplifié), contient :*
- 2^N *composantes réduites chacune à un point fixe ;*
- 2^N *composantes connexes comprenant* $2^k - 1$ *points, chacune comprenant un cycle de longueur k.*

Preuve.
Notons $W_k = (x_k, x_{k-1}, \ldots, x_1)$. Sans perte de généralité, nous décrirons les fichiers du système S, autres que les parties consituant W_k par le N-uplet $(x_{N+k}, x_{N+k-1}, \ldots, x_{k+1})$. Tout point de $x \in \mathbb{F}_2^{N+k}$ sera représenté par le couple $(y, \overline{y})$ avec $\overline{y} \in \mathbb{F}_2^k$ (partie « virale » de x).

Pour les composantes connexes réduites à un point, elles sont constituées des points $x \in \mathbb{F}_2^{N+k}$ dont le support ne contient aucun fichier correspondant à une des parties P_i du code W_k. Il y en a exactement 2^N.

Pour le second point de la proposition, chacune des composantes connexes contient tous les points dont l'image par la fonction F^t est l'un des k points e_i (rappel : ce point désigne le point dont toutes les coordonnées sont nulles sauf la i-ième). En outre, pour chaque point e_i, nous avons $F^t(e_i) = e_{i+1}$ avec la convention que $k + 1 = 1$. Les fichiers non infectables n'ayant aucune interaction avec une quelconque partie du code W_k, pour tout $x = (y, \overline{y})$, nous avons $F^t(x) = (y, F^t(y'))$.

En outre, pour $x = (y, \overline{y})$ et $x' = (y', \overline{x'})$ tels que $y \neq y'$, alors nous avons $F^t(x) \neq x'$ et $F^t(x') \neq x$. Comme, il y a 2^N y possibles, nous obtenons le résultat. ■

Ce résultat nous permet d'établir la complexité de la détection d'un code de type W_k (séquentiel k-aire) à partir du graphe d'itération (autrement dit par analyse fonctionnelle).

Théorème 4.4 *Soit un système S infecté par un code k-aire de type séquentiel. La détection d'un tel code dans ce système, fondée sur l'analyse de l'évolution fonctionnelle du système, est un problème NP-complet.*

Preuve.
Nous nous placerons dans le cadre du modèle booléen généralisé. Autrement dit, toutes les interactions possibles entre les N fichiers du système S sont prises en compte. Cela implique que les 2^N composantes connexes ne sont plus isolées et que des arcs unissent un grand nombre d'entre elles. Le « raccordement » des composantes connexes peut être affiné dans le sens d'une plus grande complexité du graphe, par les techniques de leurrage mises en œuvre par chacune des parties P_i (voir section 4.3.1).

Détecter le code W_k implique, par définition, de détecter chacune de ses parties. Pour chaque $x \in \mathbb{F}_2^{N+k}$, il faut rechercher un cycle de longueur k de type hamiltonien. Or, ce problème est NP-complet, dans le cas général [99]. D'où le résultat. ■

Ce résultat montre que la détection des codes de type k-aires en mode séquentiel est un problème difficile au sens de la théorie de la complexité, autrement dit impossible à résoudre en pratique pour un antivirus. Il n'est pas nécessaire que la valeur de k soit très importante. En effet, un choix optimal de la valeur Δt du délai séparant l'action consécutive de deux parties P_i et P_{i+1} permet de se limiter opérationnellement à de faibles valeurs de k. Rappelons que les différentes parties P_i doivent être conçues de sorte à être le moins possible différenciables fonctionnellement d'un programme légitime.

Si l'on considère le graphe de transition décrivant un code de type W_k, nous avons alors la proposition suivante.

Proposition 4.8 *Soit un système S infecté par un code W_k, k-aire de type séquentiel. Alors, le graphe de transition décrivant l'infection de S par le code $W_k = (x_k, x_{k-1}, \ldots, x_1)$ (modèle simplifié) contient un sous-graphe dirigé complet constitués des points $(e_i)_{1 \leq i \leq k}$.*

Preuve.
Nous laisserons le lecteur établir la preuve à titre d'exercice. Elle s'établit de manière similaire à celle de la proposition 4.4. ■

Cette proposition montre alors que lorsque l'on considère le modèle booléen généralisé, la détection fondée sur l'étude du graphe de transition (modélisant les interactions entre les fichiers), la détection des codes de type k-aire en mode séquentiel est un problème difficile.

Théorème 4.5 *Soit un système S infecté par un code k-aire de type séquentiel. La détection d'un tel code dans ce système, fondée sur les interactions entre les fichiers du système S est un problème NP-complet.*

Preuve.
La preuve de ce théorème est similaire à celle du théorème 4.3. ■

La détection des codes de type W_k (k-aires séquentiels) est donc un problème impossible à résoudre en pratique, du fait de sa complexité. Mais contrairement aux virus polymorphes, la connaissance d'une partie P_i n'est pas suffisante pour analyser le code : il est nécessaire de disposer des k parties. Chaque partie ne contient qu'une part limitée de l'information. Cela montre tout l'intérêt des codes k-aires séquentiels de la sous-classe B par rapport aux sous-classes A et C (voir exercices). Nous pouvons résumer cela dans la proposition suivante.

Proposition 4.9 *Soit $F^t(x)$ la fonction de transition décrivant l'infection par un code k-aire séquentiel de type W_k, d'un système contenant N fichiers (avant infection). Soit la fonction d'infection f_{W_k}. Alors, pour tout $u \in \mathbb{F}_2^k$, tel que $supp(u) \subseteq supp(W_k)$,*

$$P[(f_v \circ F^t)(x) =< x, u >] = \frac{1}{2} - s(u)\frac{1}{2^k},$$

avec $s(u) = (-1)^{wt(supp(u))}$.

Preuve.
Rappelons que supp(W_k) désigne le $(N+k)$-uplet dont seules les k positions correspondant aux parties constituant W_k valent 1. Le résultat se démontre alors aisément en considérant le fait que $< x, u >= 0$ avec une probabilité exactement égale à $\frac{1}{2}$ et que la fonction $(f_v \circ F^t)(x) = 1$ pour $x \cap \text{supp}(W_k) = \text{supp}(W_k)$. Alors pour ce dernier cas, soit supp(u) est de poids impair, alors $< x, u >= 1$ et $s(u) = -1$, sinon $< x, u >= 0$ et $s(u) = 1$. D'où le résultat. ■

Ce résultat montre que la détection statistique par sondage sur le graphe de transition devient quasi-impossible dès lors que k croît.

4.4 Codes k-aires en mode parallèle

Les codes k-aires de ce type agissent simultanément en mémoire. Cela constitue potentiellement une faiblesse dans la mesure où l'analyste dispose, en théorie, de la totalité de l'information relative au code malveillant, contrairement aux codes k-aires de type séquentiel. Mais en pratique, la mise en œuvre de codes k-aires de type parallèle peut se révéler efficace et représenter une véritable menace. Deux raisons essentielles, parmi certainement d'autres, sont à considérer :

- ces codes peuvent être mis en œuvre comme processus furtifs, utilisant les techniques de type *rootkit* les plus récentes (technologies *BluePill* et *SubVirt*; voir chapitre 7). L'utilisation conjointe de ces technologies avec des codes k-aires en mode parallèle permet d'accroître le niveau de sophistication de l'attaque;
- le problème de la détection n'est pas le seul à prendre en compte. Celui de l'éradication l'est tout autant. Nous verrons comment un code k-aire de type parallèle des sous-classes A ou C peut agir malgré des faiblesses apparentes et mettre en difficulté un grand nombre de systèmes.

4.4.1 Le virus PARALLÈLE_4

Ce code malveillant a été rencontré en 2004 sur un système réel mais il n'est pas officiellement documenté. Détecté d'une manière générique par quelques antivirus seulement, c'est-à-dire sans identification particulière, ce code survit à toute tentative de désinfection (mise en quarantaine), alors que des fichiers

viraux sont bien identifiés et isolés. Ce code a été baptisé PARALLÈLE_4, en vertu de ses mécanismes de fonctionnement. Il s'agit d'un code 4-aire de type parallèle, de la sous-classe A. Nous allons présenter une version évoluée et améliorée de ce code, qui s'inspire fortement de la forme rencontrée sur un système réel.

Les caractéristiques du virus PARALLÈLE_4 et ses mécanismes d'action sont les suivants :

- chacune des quatre parties du code contient une partie de l'information (actions réalisées par le virus) ;
- les quatre parties sont lancées simultanément et sont résidentes ;
- chacune des parties est capable de régénérer les trois autres sous une forme mutée, la mutation incluant le nom même de ces parties ;
- outre les actions propres aux virus, chacune pour la part qui lui est dévolue, chaque partie surveille les trois autres et vérifie en permanence qu'elles sont bien résidentes. Si l'une d'entre elles ne l'est plus (mort du processus correspondant), l'une des trois autres, aléatoirement, la relance (remise en résidence). Un mécanisme de lutte contre la surinfection mémoire permet d'interdire que plusieurs processus correspondant à une même partie ne soient simultanément actifs. Le nom de chaque processus étant différent à chaque fois, chacune des parties est capable d'identifier en aveugle les trois autres, et ce en permanence ;
- chaque partie, à intervalles réguliers, s'autorafraîchit : elle se duplique en mémoire et le processus « parent » s'autodétruit. Le nom du processus change à chaque fois et chaque partie est capable, indépendamment, de repérer la mutation de nom.

Lorsque l'antivirus – quand ce dernier est capable de déterminer la présence d'un processus douteux – supprime l'une des quatre parties composant ce code, les trois autres sont en mesure de réactiver le processus, sous un nom différent. Le résultat est que l'action antivirale se limite à la détection. L'éradication n'est pas possible.

L'action du code PARALLÈLE_4 est possible sans que cela soit perceptible au niveau de l'activité du système : aucun ralentissement notable n'est observé. Toutefois, dans le cas d'une action antivirale contre le code malveillant, un fort ralentissement peut être noté voire provoqué volontairement par ce dernier. Cela permet une rétorsion en cas d'action de l'antivirus et déboucher sur un véritable déni de service contre le système, voire un blocage de ce dernier lorsqu'une action antivirale est suspectée par le code malveillant.

4.4.2 Modèle théorique

La modélisation des codes k-aires de type parallèle, quelle que soit la sous-classe, n'est conceptuellement pas différente des codes k-aires séquentiels. L'exécution des processus étant par essence elle-même séquentielle (modèle de Turing), les premiers peuvent se ramener aux seconds. Les différentes parties d'un code k-aire parallèle doivent nécessairement s'exécuter selon un certain ordre.

Dans le cas d'un véritable parallélisme, le modèle est en revanche plus complexe et constitue une voie de recherche actuelle (voir section 4.5).

Le principal résultat que l'on peut donner, à partir du graphe de transition, est le suivant.

Proposition 4.10 *Soit un système S infecté par un code k-aire de type parallèle, noté P_k. Alors, le graphe de transition décrivant l'infection de S par le code $P_k = (x_k, x_{k-1}, \ldots, x_1)$ (modèle simplifié) contient un sous-graphe dirigé complet constitués des points $(e_i)_{1 \leq i \leq k}$.*

Ce résultat se démontre de la même manière que pour les codes k-aires de type séquentiel [54].

En considérant la propriété de leurrage et le modèle booléen généralisé, cette proposition permet de montrer que la détection des codes de type k-aires en mode parallèle, fondée sur l'étude du graphe de transition (modélisant les interactions entre les fichiers), est un problème difficile.

Théorème 4.6 *Soit un système S infecté par un code k-aire de type parallèle. La détection d'un tel code dans ce système, fondée sur les interactions entre les fichiers du système S est un problème NP-complet.*

La preuve est laissée à titre d'exercice ; voir également [54]. D'un point de vue général, la plupart des résultats théoriques établis pour les codes k-aires de type séquentiels sont valables pour les codes parallèles. La différence notable tient au fait que l'attaque peut être réalisée beaucoup plus rapidement (les codes sont tous actifs, simultanément en mémoire). En revanche, surtout pour les sous-classes A et C, la présence simultanée de tous les codes en mémoire constitue une faiblesse, sauf dans le cas d'une utilisation conjointe avec une technologie *Rootkit* puissante. En revanche, la sous-classe B est elle plus puissante puisque la connaissance d'une des k parties (ou d'un sous-ensemble de ces parties) ne permet ni d'identifier ni d'analyser les autres. Mais cette sous-classe est plus délicate à réaliser.

4.5 Conclusion

Les codes k-aires constituent une classe puissante de codes. Outre de nombreuses applications bénéfiques potentielles (protection de programme dans un cadre de protection du droit d'auteur, par exemple), dans un contexte malveillant, ils constituent une nouvelle menace dont la perception est encore balbutiante. Les premières études et analyses démontrent clairement l'énorme potentiel de ces codes, et de là le risque très important pour la protection des systèmes. La complexité en pire cas de la détection de ces codes ne permet pas aux antivirus d'agir efficacement. La complexité en cas moyen, laquelle décrit mieux les contraintes auxquelles est confronté en pratique un antivirus, montre que ces codes sont de nature à défaire ces derniers, et ce pour longtemps.

Mais les quelques aspects théoriques présentés dans ce chapitre ne décrivent qu'une faible part des codes k-aires possibles. Les fonctions booléennes vectorielles (fonctions de transition) permettent de décrire les choses de manière simplifiée mais néanmoins puissante. Les règles présidant à la construction des fonctions de transition présentées dans ce chapitre ne sont que quelques exemples d'un ensemble beaucoup plus large. Nous pourrions par exemple définir des règles de sorte à, par exemple, avoir un graphe hamiltonien à la fois dans le graphe d'itération et dans le graphe de transition.

Toutefois, avec les fonctions booléennes, par leur nature déterministe, seule une faible partie des codes k-aires peut être envisagée. En effet, pour $x \in \mathbb{F}_2\ n$, la valeur $F^t(x)$ est alors fixée : l'évolution du code, modélisée au moyen de cette fonction, est alors déterministe. Or, il est plus intéressant, si l'on veut rendre encore plus complexe la détection des codes, de considérer des fonctions F^t pour lesquelles x peut avoir plusieurs images possibles. Il est alors nécessaire de considérer des objets mathématiques beaucoup plus complexes, comme les automates cellulaires non déterministes (voir chapitre 6). Les premières études (voir section 6.4.3) montrent que l'on peut déboucher sur des problèmes appartenant à des classes de complexité beaucoup plus fortes que la simple classe NP, voire sur des problèmes indécidables. L'utilisation d'automates cellulaires permet également de considérer des fonctions de transition non définies en un nombre fini de points.

Exercices

Sauf indication particulière, les exercices font référence au modèle booléen simplifié, tel que défini par la convention 4.1.

1. Établissez la fonction de transition F^t dans le cas d'un virus simple avec $n = 6$ et $m = 2$. Dessinez ensuite le graphe d'itération associé. Que pouvez-vous dire au sujet des composantes connexes relevant du troisième cas de la proposition 4.1 ? Démontrez que l'une d'entre elles possède dans le cas général toujours 2^m points. Quelle propriété particulière ont ces derniers ?
2. Établissez les fonctions booléennes coordonnées des différents exemples de fonctions F^t de la section 4.2.3.
3. Soit un plus grand ensemble viral $PGEV(M)$, pour une machine de Turing donnée M, composé de trois virus. Soit un système S composé de $n = 7$ fichiers dont deux sont non infectables par $PGEV(M)$. Établissez les fonctions de transition et d'infection de $PGEV(M)$, la matrice de transition, et dessinez les graphes de transition et d'itération. Un programme en langage C peut être réalisé et produire des fichiers de sortie pour le logiciel *GraphViz* (disponible sur `http://www.graphviz.org`).
4. Montrez, en considérant une relation d'équivalence et un quotientement différents de ceux définis dans la section 4.2.1, que pour le modèle booléen

utilisé dans ce chapitre, les cas des virus simples et des virus polymorphes sont identiques. Montrez ensuite que cette équivalence ne s'étend pas aux virus métamorphes (consultez la section 6.1 pour une définition précise de ces concepts). Définissez alors une nouvelle relation d'équivalence permettant de ramener le cas des virus métamorphes au cas des virus simples. Que concluez-vous en terme de lutte antivirale ?

5. Avec le théorème 4.3, il a été montré que la détection des virus polymorphes reste de complexité NP-complet avec le modèle booléen. Toutefois, la mutation du virus dans le système se matérialise par l'existence dans le graphe de transition de chaînes, dont la longueur correspond exactement au nombre k de formes mutées effectivement générées moins une unité, ces chaînes unissant les variables $x_{v_1}, x_{v_1}, \ldots, x_{v_k}$. La recherche de chaînes d'une longueur fixée étant un problème de complexité polynomiale, expliquez pourquoi, malgré tout, la complexité générale de la détection des virus polymorphes est au mimimum de type NP.
6. Établissez les formules donnant le nombre de points contenus dans les composantes connexes de la proposition 4.5, non réduites à un point fixe et différentes de la composante virale, en fonction du poids de Hamming (nombre de positions non nulles dans l'écriture binaire) du point fixe de cette composante.
7. Montrez à l'aide du modèle booléen défini dans ce chapitre, que si l'on connaît la fonction de mutation μ d'un virus polymorphe (ou métamorphe), ce dernier peut être ramené au cas d'un virus simple.
8. Démontrez qu'en considérant non plus le modèle booléen simplifié défini par la convention 4.1 mais le modèle booléen généralisé (prenant en compte toutes les interactions entre les fichiers d'un système), les résultats de la proposition 4.3 restent sensiblement les mêmes.
9. En vous inspirant de la modélisation présentée en section 4.3.2 pour les virus séquentiels de la sous-classe B, établissez le modèle correspondant pour les codes séquentiels des classes A et C. Montrez ensuite que dans le cas général la détection de ces codes possède la même complexité que la détection d'une quelconque de leurs k parties.
10. Considérons un code k-aire $W_k = (P_1, P_2, \ldots, P_k)$ de type séquentiel appliquant la règle suivante : si deux parties P_i et P_j $(i \neq j)$ ou plus sont présentes simultanément dans la machine, les différentes parties s'autodétruisent. Montrez alors que pour un système S contenant N fichiers (avant infection), le graphe d'itération décrivant l'infection de S par le code W_k contient :
    - 2^N points fixes qui ont chacun $2^k - k - 1$ prédécesseurs ;
    - 2^N composantes connexes, chacune constituée d'un cycle de longueur k.

Montrez que ce résultat permet d'affirmer que le code W_k, à partir du graphe d'itération, est une instance « facile » du problème de détection. Donnez un algorithme en temps polynomial permettant de détecter W_k.

11. Trouvez pour quelles règles de construction de la fonction de transition (cas d'un code k-aire de type séquentiel), les graphes d'itération et de transition sont de type hamiltonien.

Le cycle d'une attaque virale

Chapitre 5

Introduction

Dans cette seconde partie, nous allons considérer les différentes phases d'une attaque à l'aide d'un code malveillant – auto-reproducteur ou non –, ou plus exactement, les différentes techniques de lutte anti-antivirale mises en œuvre pendant ces phases :

- un code malveillant s'exécute à des fins d'installation dans le système et/ou de propagation vers d'autres systèmes (dans le cas d'un ver, par exemple). Des techniques de camouflage, ou plus généralement de furtivité, sont alors utilisées pour dissimuler les données du code et ses actions ;
- le code active une ou plusieurs charges finales (la ou les fonctions offensives du code). Là encore, la furtivité sert à cacher ces actions. Mais le blindage de code, destiné également à retarder (blindage léger) ou à interdire (blindage total) l'analyse du code malveillant et la mise à jour des antivirus et autres logiciels de sécurité, sera également utilisé afin que la nature de ces actions restent secrètes ;
- le code s'enterre dans le système, entre en dormance, en résidence et met en œuvre des mécanismes de persistance. Conjuguée à l'utilisation des techniques de furtivité, là encore, la survie du programme et des actions, pendant cette phase critique, est assurée en faisant également muter le code grâce à des techniques de polymorphisme et de métamorphisme ;
- enfin, dans l'éventualité où le code serait finalement capturé, le blindage compliquera ou interdira son analyse.

Toutes ces techniques de lutte antivirale, qui illustrent avec force le duel incessant entre la « cuirasse et l'épée », lorsqu'elles sont couplées aux techniques présentées dans la partie précédente (notamment la technologie des codes k-aires), constituent un potentiel assez inquiétant. Tout cela permet d'imaginer ce qu'une attaque ciblée vraiment agressive pourrait provoquer comme dégâts tout en contournant de manière certaine toute forme de protection et toute ligne de défense.

Comme dans la partie précédente, le parti pris adopté est celui de la modélisation. L'étude de ces modèles a permis d'en identifier les instances les plus

complexes du point de vue du défenseur. Si les problèmes ouverts sont encore très nombreux, la voie ainsi défrichée ne manque pas de révéler des techniques et des approches particulièrement puissantes pour l'action des codes malveillants.

Cette approche par la modélisation permet également de montrer que si les attaquants considérent encore des instances assez faibles d'un modèle général, identifiées le plus souvent par une approche pragmatique quelquefois de génie, ils ne sont plus très éloignés des instances les plus critiques pour le défenseur. Le cas du métamorphisme, avec un moteur comme *MetaPHOR* que nous décrirons dans le chapitre 6, est certainement le plus parlant.

Et nous ne devons pas oublier que ce constat repose uniquement sur la partie « émergée » de la virologie informatique : les codes restés indétectés et les résultats de recherche restés secrets manquent à l'appel pour avoir une idée véritablement précise du panorama réel de la virologie informatique. Cette seconde partie va tenter de montrer quelques-uns de ces aspects qui peuvent rendre une attaque indétectable ou terriblement difficile à gérer.

Elle va également tenter de montrer qu'en faisant voler en éclat les concepts actuels de la virologie informatique, concepts que l'on peut considérer comme vieillots, une science plus mûre, incroyablement plus efficace est en train d'émerger, si ce n'est pas déjà fait. L'apport de la science informatique théorique est indéniable. La formalisation du polymorphisme et du métamorphisme à l'aide de grammaires formelles est, par exemple, sans contexte une voie extrêmement prometteuse. Celle de la cryptologie, dans le cadre du blindage total, autre exemple, l'est tout autant.

Ainsi, le passage d'une algorithmique pragmatique – qui, ces dernières années, a déjà produit quelques codes *preuves-de-concept* puissants et élégants[1] – à une algorithmique raisonnée, s'appuyant sur les résultats les plus achevés de l'informatique théorique constitue une évolution majeure. Les trois prochains chapitres vont tenter de faire pressentir au lecteur ce que nous réserve l'avenir et quelles sont les voies que l'attaquant ne tardera pas à exploiter pour son plus grand bénéfice.

Mais au-delà des simples aspects offensifs et des désirs de nuisances des acteurs malveillants de la virologie informatique, cette évolution fait parvenir la virologie informatique à la maturité qu'elle mérite. Elle passe ainsi du statut d'une activité marquée du sceau de l'opprobre et de l'infamie à celui d'une science aboutie et belle, excitante pour l'esprit du chercheur, d'où tout esprit de malveillance est banni. L'espoir est que ce statut retrouvé de science véritable permettra enfin de considérer et concrétiser l'idée que la virologie informatique peut être source d'applications positives et bénéfiques, maîtrisées et servant l'Homme.

[1] Les codes produits par le célèbre groupe *29A* en sont probablement les meilleurs des exemples.

Le seul exemple de *rootkits* indéctables, surveillant un système et le protégeant des attaques, est en soi un excellent exemple. D'autres sont assurément à imaginer. Qu'elle ne sera pas la puissance de codes utilisant les techniques présentées ci-après et agissant contre les attaquants eux-mêmes mais sous le contrôle des hommes : la réponse du berger à la bergère, en quelque sorte.

Chapitre 6

Résister à la détection : la mutation de code

6.1 Introduction

La notion de polymorphisme viral est implicitement contenue dans la définition du *Plus Grand Ensemble Viral* relativement à une machine de Turing M, noté $PGEV(M)$ [26] et [38, chapitre 3]. Dans cet ensemble, deux virus sont une forme évoluée l'un de l'autre (en appliquant la fermeture transitive).

Toutefois, au-delà de la simple préoccupation formaliste de Fred Cohen, la « naissance » du polymorphisme est initiée par le besoin de lutter contre les premières techniques de détection utilisées, à savoir la recherche de signatures. Ces dernières représentant un élément fixe, l'idée de limiter la présence de ces invariants de code s'est naturellement et immédiatement imposée. En faisant varier le code, le plus possible, de réplication en réplication, les signatures ne sont plus utilisables et d'autres techniques plus complexes doivent être utilisées pour identifier un code viral, ou plus généralement malveillant. Cette variabilité de code peut être obtenue par chiffrement du code – ce dernier changeant à chaque clef – ou par des techniques de réécriture – une même action pouvant être réalisée par des codes différents (voir le théorème de Kleene [38, chapitre 2].).

Toutefois, les analystes d'antivirus ont très vite trouvé la parade. Les techniques de polymorphisme requièrent des procédures de chiffrement/déchiffrement ou des procédures similaires pour que le code puisse s'exécuter. Or ces procédures, elles, ne changent pas et constituent elles-mêmes des éléments invariants pouvant être utilisés comme signatures. En réaction, les auteurs de virus ont travaillé à faire varier la totalité du code : le *métamorphisme* est alors né.

D'une manière plus formelle, définissons[1] ce que sont polymoprhisme et

[1] Nous ne redéfinissons pas les notations utilisées ici. Le lecteur pourra consulter [38, chapitres 2 et 3].

métamorphisme. Nous reprenons le formalisme utilisé par Zhou et Zuo [149, 150].

Définition 6.1 *(Virus polymorphe à deux formes [149]) Soit v et v' deux fonctions récursives totales différentes. Le couple (v, v') est appelé virus polymorphe à deux formes si, pour tout x, le couple (v, v') v satisfait :*

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

et

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

où les prédicats récursifs $T(d,p)$, $I(d,p)$ désignent respectivement la condition de déclenchement de charge finale et la condition d'infection. La fonction $S(p)$ est une fonction récursive dite de sélection.

Dans cette définition, quand le prédicat $T(d,p)$ est vérifié, alors la charge finale est lancée (représentée par la fonction $D(d,p)$). Si le prédicat $I(d,p)$ est vérifié, alors le virus choisit un programme à l'aide de la fonction de sélection $S(p)$, l'infecte d'abord et exécute ensuite le programme original x (transfert de contrôle à la partie hôte). Notons que l'ensemble des prédicats $T(d,p)$ et $I(d,p)$, ainsi que des fonctions $D(d,p)$ et $S(p)$, constituent le *noyau* du virus. Selon ce formalisme, la fonction $S(p)$ désigne en fait la fonction réalisant la « mutation » de code (par chiffrement ou réécriture). Cette définition correspond en fait à celle d'un Plus Grand Ensemble Viral composé de deux éléments seulement. Pour des ensembles plus grands, et donc les virus polymorphes réels, la définition doit être étendue à un n-uplet $(v_1, v_2, \ldots, v_n)$ de fonctions récursives totales.

Il est également possible de considérer des virus polymorphes ayant un nombre infini (mais néanmoins dénombrable) de formes.

Définition 6.2 *(Virus polymorphe à nombre infini de formes [149]) Une fonction récursive totale $v(m,x)$ est un virus polymorphe à nombre infini de formes si, pour tout m et tout x, alors $v(m,x)$ satisfait :*

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

et si, pour tout $m \neq n$, $v(m,x) \neq v(n,x)$.

À part sous des formes triviales [149], aucun autre type de virus polymorphes ayant un nombre infini de formes n'est connu. Il s'agit là d'un problème ouvert [51].

Donnons à présent la définition des virus métamorphes.

Définition 6.3 *(Virus métamorphe [149]) Soit v et v' deux fonctions récursives totales différentes. Le couple (v, v') est appelé virus métamorphe si pour tout x, alors le couple (v, v') satisfait :*

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

et

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

où $T(d,p)$ - respectivement $I(d,p)$, $D(d,p)$, $S(p)$ - est différent de $T'(d,p)$ - respectivement $I'(d,p)$, $D'(d,p)$, $S'(p)$.

La définition se généralise à un n-uplet quelconque de fonctions récursives totales. En fait, les virus métamorphes sont similaires aux virus polymorphes, excepté que les fonctions de sélection $S(p)$ et $S'(p)$ sont différentes. Alors que les différentes formes mutées d'un virus polymorphe partagent le même noyau (en particulier la fonction de mutation), celles d'un virus métamorphes ont chacune un noyau différent.

6.2 Complexité de la détection des virus polymorphes

La détection virale la plus utilisée est la recherche de motifs fixes appelés signatures (voir section 2.3.1 pour la définition de ce terme). Plus généralement, les techniques d'analyse de forme recherchent directement ou indirectement des caractéristiques fixes représentables par des chaînes d'octets, à la structure plus ou moins complexe selon la technique utilisée. Cette recherche utilise généralement l'algorithme de Boyer-Moore [14] dont la complexité nécessite $N + S$ étapes pour une signature de taille S dans un fichier de taille N (en octets).

Dès le début des années 1990, les programmeurs de virus ont alors mis en œuvre des techniques de polymorphisme, c'est-à-dire des techniques visant à limiter le plus possible la présence et l'utilisation de ces séquences d'octets fixes (voir chapitre 2). C'est ainsi qu'en 1991 est apparu le premier moteur polymorphe, le *Dark Avenger Mutation Engine*. Si les premières techniques de polymorphisme se sont révélées assez faibles, elles ont vite évolué vers des techniques de plus en plus complexes qui parviennent encore à dérouter assez facilement les antivirus actuels. La détection des virus polymorphes est par conséquent un problème dont il est essentiel de déterminer la complexité générale. Chaque nouvelle instance de ce problème contrarie chaque fois un peu plus les éditeurs d'antivirus. On peut alors supposer que le problème spécifique consistant à détecter des virus polymorphes est un problème difficile. Mais que doit-on entendre par difficile ? Ce problème a été étudié par D. Spinellis en

2003 [127] en réduisant le problème de la satisfaisabilité des formules logiques (dénommé problème SAT ; c'est un problème NP-complet) au problème de la détection des virus polymorphes. L'année suivante, Z. Zuo et M. Zhou [149] ont généralisé le résultat de Spinellis, en utilisant la notion de fonctions récursives (voir section 6.2.3).

6.2.1 Le problème SAT

Le problème SAT (pour *satisfaisabilité*) consiste à déterminer si une formule booléenne est satisfaisable ou non. Définissons tout d'abord ce qu'est une formule booléenne.

Définition 6.4 *Une formule booléenne ϕ est une expression composée de variables booléennes $x_1, x_2, \ldots$ et de connecteurs booléens $\vee$ (OU), $\wedge$ (ET), $\neg$ (NON). Une telle formule sera donc soit une simple variable booléenne, soit une expression de la forme $\neg\phi$, soit une expression de la forme $\phi_1 \vee \phi_2$, soit encore une expression de la forme $\phi_1 \wedge \phi_2$, où ϕ, ϕ_1 et ϕ_2 sont des formules booléennes.*

Les variables booléennes valent soit 0 soit 1, ou encore FAUX ou VRAI. Nous noterons indifféremment $\neg x_i$ ou $\overline{x_i}$ pour désigner la négation de la variable x_i.

Définition 6.5 *On appelle* interprétation *d'une formule booléenne ϕ, un ensemble valeurs $(v_0, v_1, \ldots)$ pour les variables de cette formule avec $v_i \in \{0, 1\}$. Une interprétation est dite* satisfaisante *si la formule ϕ vaut 1 (est vraie) lorsque $x_i = v_i$.*

Donnons un exemple pour illustrer les deux définitions.

Exemple 6.1 *Soit la formule ϕ à trois variables :*

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \neg x_3).$$

L'interprétation $(x_1 = 1, x_2 = 1, x_3 = 0)$ est satisfaisante pour ϕ. En revanche, l'interprétation $(x_1 = 0, x_2 = 1, \forall x_3)$ ne l'est pas.

Une formule booléenne existe sous deux formes équivalentes : une forme dite *normale conjonctive* dont la structure générale est $\phi = \bigwedge_{i=0}^{m} C_i$ où les C_i désignent des clauses logiques constituées de la disjonction d'une ou plusieurs variables booléennes notées $(x_{i_1} \vee x_{i_2} \vee \ldots x_{i_k})$; une forme dite *normale disjonctive* dont la structure générale est $\phi = \bigwedge_{i=0}^{m} D_i$ où les D_i désignent des clauses logiques constituées de la conjonction d'une ou plusieurs variables booléennes notées $(x_{i_1} \wedge x_{i_2} \wedge \ldots x_{i_k})$. Il est à noter que le passage d'une forme à une autre possède une complexité mémoire exponentielle (en pire cas ; voir [99, pp. 75-76] pour la démonstration).

Définition 6.6 *Le problème* SAT *consiste à déterminer si une formule booléenne ϕ, sous sa forme normale conjonctive, est satisfaisable ou non.*

Notons que le choix de la forme normale conjonctive est motivé par le fait que c'est la représentation qui décrit le mieux le problème SAT : pour que ϕ soit satisfaisable, il suffit que chaque clause C_i soit vraie. Le problème est alors de déterminer si une formule booléenne est satisfaisable ou non. En pratique, lorsque le nombre de variables est limité, il est toujours possible de répondre à la question. Pour une formule booléenne à n variables, il suffit de parcourir les 2^n interprétations et d'en exhiber une pour laquelle la formule est satisfaite. Mais dès que le nombre de variables devient trop important, cette approche n'est plus possible. En existe-il une autre permettant de faire mieux que l'approche exploratoire ? La réponse est malheureusement négative, comme le prouve le théorème suivant.

Théorème 6.1 *Le problème* SAT *est NP-complet.*

Le lecteur pourra consulter [99, chapitre 9] pour une démonstration de ce célèbre résultat). Cela signifie que parmi tous les problèmes de NP[2], le problème SAT figure parmi les plus difficiles.

6.2.2 Le résultat de Spinellis

D. Spinellis s'est intéressé au problème de la détection des virus de taille finie subissant une mutation au cours du processus de duplication – voir la définition de Fred Cohen dans [38, chapitre 3]. Il s'agit donc d'un cas particulier de polymorphisme (la taille étant finie). L'approche de Spinellis a été de montrer qu'un détecteur $\mathcal{D}$ pour un virus mutant donné V peut être utilisé pour résoudre le problème SAT. Or, dans la section précédente, il a été montré que ce problème est NP-complet.

Théorème 6.2 *Le problème de la détection d'un virus polymorphe de taille finie est un problème NP-complet.*

Démontrons ce théorème – nous reprenons ici à quelques différences de notation près, la preuve originale donnée dans [127].

Preuve.
Supposons que $\mathcal{D}$ soit capable de déterminer de manière sûre et en temps polynomial si un programme $\mathcal{P}$ est une version mutée du virus V. Le but est de

[2] Rappelons que le terme NP *(Non deterministic Polynomial)* désigne les problèmes effectivement calculables par une machine de Turing non déterministe. Cela signifie que la complexité du problème est potentiellement exponentielle.

considérer $\mathcal{D}$ comme un oracle[3] pour déterminer la satisfaisabilité d'une formule booléenne F à n variables. Rappelons que F possède la forme suivante :

$$F = (x_{a_{1,1}} \vee x_{a_{1,2}} \vee x_{a_{1,3}}) \wedge (x_{a_{2,1}} \vee \overline{x_{a_{2,2}}}) \wedge \ldots (x_{a_{i,j}} \vee \ldots) \wedge \ldots$$

avec $0 \leq a_{i,j} < n$. Par conséquent, $\mathcal{D}$ constituerait – s'il existait – une solution en temps polynomial du problème SAT.

Une formule de type de celle de F est tout d'abord utilisée pour créer un virus archétype A et un phénotype de satisfaisabilité P caractérisant une instance possible mutée de A. En considèrant qu'un virus peut être décrit par un triplet (f, s, c) avec les notations suivantes :

- f est la fonction de duplication et de calcul (déterminer si F est satisfaite ou non avec c) ;
- s est une valeur booléenne (*Vrai* ou *Faux*) indiquant si une instance du virus a calculé une solution pour F ;
- c est un entier décrivant (codant) les valeurs possibles en entrée de F (autrement dit $0 \leq c < 2^n$ et $c = (x_0, x_1, \ldots, x_{n-1})$).

La fonction f associe à tout triplet (f, s, c) un triplet (f, s', c') de la manière suivante[4] :

$$\lambda(f, s, c).(f, s \vee F, \text{si } c = w^n \text{alors } c \text{ sinon } c + 1).$$

La $(i+1)$-ième génération du virus est produite en appliquant la fonction f à la i-ième génération. Autrement dit, les étapes suivantes sont successivement réalisées :

1. évaluation de $F(c)$ avec $c = (x_0, x_1, \ldots, x_{n-1})$;
2. incrémentation c (avec $c < 2^n$) ;
3. calcul de $s \vee F(c)$.

Maintenant, $\mathcal{D}$ doit décider si le virus archétype A défini par $(f, \text{Faux}, 0)$, par l'application successive de la fonction f, produira jamais la mutation P définie par le triplet $(f, \text{Vrai}, 2^n)$. Il s'agit donc pour $\mathcal{D}$ de déterminer si une des mutations de A satisfera la formule F (notons que cela peut se produire prématurément pour un $c' < 2^n$ mais comme la valeur booléenne `Vrai` est un élément absorbant pour l'opération logique $\vee$, la notation précédente est utilisée).

[3] Un oracle, plus exactement appelé *machine de Turing avec oracle*, est une machine de Turing M dotée d'une bande de calcul supplémentaire, appelée bande oracle, d'un état $q_?$ et de deux états q_{non} et q_{oui}. Soit maintenant A tout langage sur un alphabet Σ. Chaque fois que M rencontre l'état $q_?$ pour une chaîne quelconque $z \in \Sigma^*$ sur la bande oracle, alors M passe dans l'état q_{oui} si $z \in A$ ou dans l'état q_{non} si $z \notin A$. En d'autres termes, il s'agit d'une modélisation de la notion traditionnelle d'oracle.

[4] La démonstration initiale utilise la notation lambda de Church pour définir des fonctions partielles. Ainsi $\lambda x.f(x)$ désigne la fonction qui à x associe $f(x)$.

Nous avons ainsi montré que si un tel détecteur $\mathcal{D}$ existait (en d'autres termes, était capable d'identifier qu'un virus est une forme mutée d'une autre et ce, de manière systématique), alors il pourrait être utilisé pour résoudre le probème SAT en temps polynomial. D'où le résultat. ■

À titre d'illustration [127], considérons la formule $F = (x_0 \vee x_1) \wedge \overline{x_0}$. La fonction f est alors définie par

$$\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \overline{x_0}, c + 1).$$

Le virus archétype A est donné par

$$(\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \overline{x_0}, c + 1), \text{Faux}, 0),$$

tandis que le phénotype P caractérisant la satisfaisabilité (et donc la forme mutée) est défini par

$$(\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \overline{x_0}, c + 1), \text{Vrai}, 4).$$

Le lecteur vérifiera que A génère une mutation satisfaisant P après quatre générations.

Le lecteur trouvera dans [127, Annexe II] un exemple écrit en langage C, de programme illustrant la démonstration précédente. Bien évidemment, le programme proposé n'est pas réellement un virus mais simule en partie et trivialement la variabilité du code par les différentes « mutations » du code global (illustrées par les différentes valeurs de c remplissant le tableau $x[N]$). Dans un contexte viral réel, l'entier c peut être décrit par un entier de Gödel e_P (voir [38, chapitre 2]) dont la décomposition binaire satisfait la formule F. La formule F est elle-même calculée à partir d'un entier de Gödel e_A ne satisfaisant pas F.

Le résultat de Spinellis est intéressant et important car il prouve la complexité générale du problème de la détection des virus polymorphes. Cependant, il est nécessaire de faire quelques remarques :

- Spinellis ne considère qu'un cas restreint du polymorphisme, quand la souche initiale A (et donc la fonction de duplication f) est connue. Nous verrons dans la section 6.2.3 comment se généralise ce résultat ;
- d'autres programmes que P peuvent « satisfaire » la formule F sans pour autant provenir par mutation de A : le poids de F, c'est-à-dire le nombre de valeurs c telle que $F(c)$ soit vraie, n'est pas nécessairement égal à 1. Cela illustre la notion de fausse alarme (ou faux positifs).

6.2.3 Résultats généraux

Les résultats de Spinellis sont d'autant plus intéressants qu'il n'existe que peu de résultats théoriques concernant le problème de la détection virale [38, chapitre 3]. Un grand nombre de problèmes ouverts subsistent encore [51] et beaucoup n'ont probablement pas encore été identifés.

Concernant spécifiquement les virus polymorphes, le seul autre résultat connu à ce jour concerne les virus polymorphes ayant un nombre infini de formes. Il a été établi par Z. Zuo et M. Zhou en 2004. Notons D_i l'ensemble des virus polymorphes ayant un nombre infini de formes et $D_i^{\text{fixé}}$, l'ensemble des virus polymorphes ayant un même noyau donné (voir section 6.1).

Théorème 6.3 *[149] L'ensemble $D_i^{fixé}$ est Π_2-complet et l'ensemble D_i est Σ_3-complet.*

Rappelons les correspondances existant entre les différentes notations de la théorie de la complexité. La classe $\Pi_0 = \Sigma_0$ est en fait la classe P des problèmes polynomiaux et nous avons $NP = \Sigma_1$, $coNP = \Pi_1$, $\Sigma_2 = NP^{NP}$ et $\Pi_2 = coNP^{NP}$. Plus généralement, $\Sigma_{i+1} = NP^{\Sigma_i P}$ et $\Pi_{i+1} = coNP^{\Sigma_i P}$ (voir [99, chapitre 17]). En terme de complexité, les problèmes que l'on peut

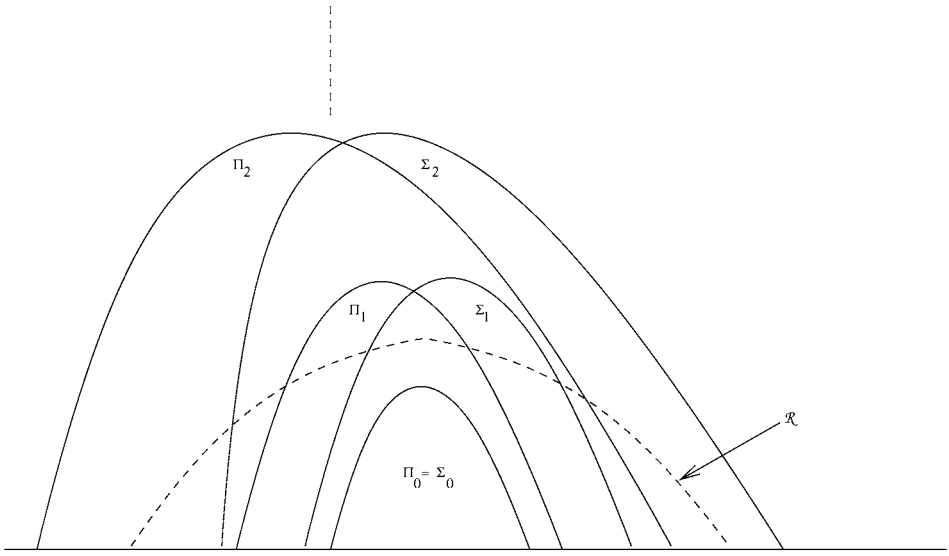


Figure 6.1 – Classes Π_n et Σ_n et leur hiérarchie

calculatoirement résoudre (ensemble $\mathcal{R}$) se trouvent sous la courbe en pointillé de la figure 6.1. Le théorème précédent constitue une généralisation du théorème 6.2. Il montre que la détection des virus polymorphes dans le cas général est un problème appartenant à une classe de complexité supérieure à la classe NP.

En revanche, en ce qui concerne les virus métamorphes, aucun résultat de complexité n'est connu à ce jour. S'il est évident que leur détection est au minimum celle des virus polymorphes, en revanche rien ne permet pour le moment d'affirmer que la détection de ces codes n'appartient pas à une classe de

complexité supérieure. Il s'agit là également d'un problème ouvert. Cependant, dans la section 6.4, nous apporterons un début de réponse en modélisant les techniques métamorphes à l'aide de grammaires formelles.

6.3 Les techniques de mutation de code

La mutation d'un code a pour but, comme nous l'avons déjà évoqué, de contrarier les techniques traditionnelles d'analyse de forme. D'abord polymorphes, les codes malveillants ont dû s'adapter et devenir plus complexes pour contourner les évolutions des antivirus : le métamorphisme est né même si peu de cas sont actuellement connus (ce sont d'ailleurs essentiellement des codes preuve-de-concept). Mais si cette évolution peut sembler banale car s'inscrivant dans le contexte normal du sempiternel duel « bouclier contre épée » – discours tenu par les concepteurs et éditeurs d'antivirus –, il n'en est rien. Le passage aux techniques de mutation a constitué un saut conceptuel important et un défi permanent qui va bien au-delà de ce simple duel. Et l'on peut affirmer que les programmeurs de codes malveillants ont gagné ce duel. Expliquons pourquoi.

Les éditeurs sont contraints de suivre le rythme et les évolutions imposés par la communauté des programmeurs de codes malveillants. Pour chaque nouvelle algorithmique virale, les antivirus doivent ou devraient être modifiés et ce de plus en plus en profondeur. Tous les codes, connus ou inconnus, pour peu qu'ils utilisent cette nouvelle algorithmique, sont en théorie détectés[5]. Dès qu'un nouveau concept viral fait son apparition – en terme de programmation et/ou de concept –, les antivirus doivent alors être complètement repensés et modifiés. Cela a pour conséquence, au fur et à mesure des évolutions virales, d'une part de retarder la lutte toujours un peu plus, de la rendre plus complexe et d'exercer une pression croissante sur la communauté antivirale, mais également de diminuer globalement l'efficacité des antivirus.

En effet, la détection de chaque nouvelle technologie virale réclame des ressources croissantes (mémoire et temps de calcul), lesquelles ne peuvent pas être consacrées de la même manière à tous les virus. Ainsi, on découvre que certaines techniques de mutation, pourtant anciennes, ne sont plus prises en compte par les antivirus. Ces derniers ne sont généralement capables de gérer efficacement les virus que dans une « fenêtre temporelle » de plus en plus limitée et pour les codes les plus répandus : comme en médecine, seules les grandes « maladies virales » font l'objet d'attention, les « maladies virales orphelines » ou anciennes sont délaisées voire ignorées[6]. Le problème est que leur nombre augmente. Au final, si les codes finissent toujours par être détectés à un moment donné, pendant combien de temps le sont-ils ?

L'autre aspect concerne l'éradiction des codes malveillants. Le marketing se concentre sur la détection mais il est singulièrement muet concernant la désinfection. Et là, les techniques polymorphes/métamorphes posent quelquefois de

[5] En théorie, car la réalité est tout autre, comme il a été montré dans les chapitres précédents.
[6] Cela concerne notamment les attaques ciblées qui ne sont pratiquement jamais détectées.

sérieux problèmes, notamment lorsqu'elles sont combinées avec la technologie des codes k-aires (voir chapitre 4), qui ne sont jamais évoqués. Par expérience, il existe des codes détectés mais que les antivirus sont incapables de traiter (voir également la section 4.4). Ils constatent, démarrage après démarrage, la présence de ces virus mais le mal est toujours là. Les sites des éditeurs ne les mentionnent que très rarement. C'est une sorte de cynisme viral : être détecté n'est plus un problème pour un code si les antivirus ne parviennent pas à le désinfecter. C'est également en cela que les techniques de mutation de codes ont démontré leur puissance.

6.3.1 Les techniques de polymorphisme

Les techniques polymorphes semblent très nombreuses mais en réalité, d'un point de vue conceptuel, le polymorphisme se résume à quelques grands principes, lesquels s'expriment de nombreuses manières au niveau de leur implémentation, des possibilités offertes par le système d'exploitation ou le processeur et l'ingéniosité d'implémentation du programmeur. Elles sont de deux sortes et sont utilisées de manière complémentaire. La structure générique d'un moteur polymorphe est la suivante (résumée dans la figure 6.2) :

- une routine de chiffrement/déchiffrement, qui a pour fonction de chiffrer la plus grande partie du code malveillant. Le code est ainsi différent à chaque clef ;
- une routine de polymorphisme (ou *polymorpheur*) dont la fonction est de faire varier la routine de chiffrement/déchiffrement. Les premiers virus chiffrés utilisaient un décrypteur fixe, lequel a très vite constitué une signature. L'idée a été alors de faire muter le chiffrement utilisé. Cela impose de gérer simultanément le chiffrement et le déchiffrement. Selon la nature des algorithmes utilisés, la gestion de cette mutation est plus ou moins complexe.

Le polymorpheur : polymorphisme par réécriture

Les techniques de réécriture consistent à transformer des instructions en d'autres instructions équivalentes. Ces systèmes de réécriture sont assez complexes à concevoir et à écrire si on veut qu'ils soient efficaces. C'est probablement la raison pour laquelle on les rencontre peu, si ce n'est de manière triviale dans la plupart des virus. Ces techniques sont d'un usage plus fréquent et conséquent dans certains virus métamorphes comme le virus *Win32/Linux.MetaPHOR*, présenté dans la section 6.3.2. Ces techniques sont généralement gérées par les antivirus, une fois qu'elles sont connues (voir section 6.4). Nous montrerons dans la section 6.4.4 comment formaliser le problème de la réécriture et comment utiliser des techniques de réécriture complexes pour produire des virus extrêmement difficiles à détecter. Ces techniques de réécriture sont de plusieurs ordres :

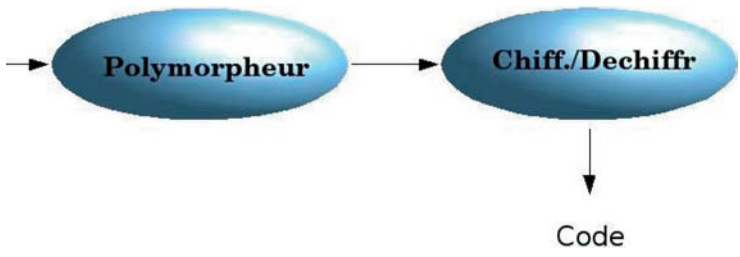


Figure 6.2 – Structure générale d'un moteur polymorphe

- *lexicales.* Les instructions sont remplacées par des instructions équivalentes. Les substitutions peuvent être relativement triviales : remplacer par exemple un `DEC EAX` par un `SUB EAX, 1` ou un `XOR EAX, EAX` par un `MOV EAX, 0`. Des substitutions moins évidentes sont également possibles en utilisant des opcodes équivalents : ainsi les deux instructions hexadécimales `8BF8H` et `89C7H` se décodent toutes deux en `MOV EDI, EAX` – voir [85] pour d'autres exemples utilisant le format d'instruction d'Intel. Ces transformations sont faciles à détecter, ne serait-ce que par émulation de code. Tout comme le sont les techniques consistant à remplacer une instruction par plusieurs autres : par exemple l'instruction `MOV EAX, EBX` sera remplacée par le bloc `PUSH EBX / POP EAX` ;
- *syntaxiques.* L'ordre des instructions est modifié mais l'action du code reste inchangée. Le principe de base est d'utiliser l'indépendance des opérations. Ainsi la séquence `MOV ESI, S / XOR EAX, EAX` est équivalente à la séquence `XOR EAX, EAX / MOV ESI, S`. Toutefois, l'action du code est le plus souvent dépendante de l'ordre des instructions. Si on veut les inverser, il est alors nécessaire de conserver, lors de la protection du code, des éléments suffisants pour restaurer l'ordre requis des instructions lors de la phase de déprotection. La suite d'instructions `XOR EAX, EAX / POP EAX` est différente de `POP EAX / XOR EAX, EAX` ;
- *morphologiques.* Il s'agit de changer l'apparence globale du code tout en conservant sa structure. Le cas le plus fréquent consiste à insérer des instructions nulles ou sans effet (*garbage code*) : `NOP`, `XOR Reg, 0`, `DEC EAX / INC EAX`...

Toutes ces techniques sont en général combinées pour générer des modules de chiffrement/déchiffrement différents à chaque mutation. Ainsi, le code suivant :

```
     MOV AX, 3D02
     INT 21H
```

peut être transformé en

```
     SUB AX, AX
     ADD AX, 3000
     CMP AX, 78AF
     JE  PAS_DE_SAUT
     XOR AX, 0D02
PAS_DE_SAUT:
     INT21
```

Le code suivant d'un décrypteur générique

```
       MOV EDI, OFFSET START_ENCRYPT  ; EDI = offset du
                                      ; corps viral
       ADD EDI, EBP
       MOV ECX, 0A6BH                 ; taille du code viral
       MOV AL, SS:Key[EBP]            ; la clef (un octet)
DECRYPT_LOOP:
       XOR [EDI], AL                  ; chiff./dechiff. par
                                      ; xor constant
       INC EDI                        ; compteur++
       LOOP DECRYPT_LOOP
       JMP SHORT START_ENCRYPT        ; saut au debut
                                      ; du code
```

peut ainsi être transformé en le code suivant :

```
       MOV  EAX, OFFSET START_ENCRYPT + 666
       PUSH EAX
       POP  EDI
       ADD  EDI, -666
       AND  EDI, EDI
       JZ   PAS_DE_SAUT
       XOR  ECX, ECX
       ADD  ECX, BDFFH
       XOR  ECX, B795H
       MOV  AL, SS:Key[EBP]
DECRYPT_LOOP:
       MOV  BL, AL
       MOV  BH, AL
       NEG  BL
       MOV  DL, [EDI]
       AND  [EDI], BL
       NEG  DL
```

```
        AND  DL, BH
        OR   [EDI], DL
        ADD  EDI, 1
        LOOP DECRYPT_LOOP
        JMP SHORT START_ENCRYPT
PAS_DE_SAUT:
        .....
```

Les instructions de sauts conditionnels et non conditionnels permettent également de combiner ces techniques tout en modifiant la structure du code. À titre d'illustration, l'exemple suivant, en langage C permet de s'en convaincre :

```
if(C == val)                    if(C != val)
 {                               {
  bloc_instr. 1                   bloc_instr. 2
 }                    <------>   }
else                            else
 {                               {
  bloc_instr. 2                    bloc_instr. 1
 }                               }
```

Les différents types d'instructions de saut en assembleur autorisent un grand nombre de variations. Cependant, un émulateur efficace saura gérer ces techniques de polymorphisme dans les cas les plus fréquents, car l'exploitation des sauts à des fins de polymorphisme est assez frustre dans la plupart des virus connus. Aussi, deux techniques ont été imaginées[7] pour défaire les émulateurs actuels (en particulier lorsqu'elles sont combinées :

- la technologie PRIDE (*Pseudo-Random Index DEcryption*) que nous présenterons dans la section 6.3.2. Elle met en œuvre un mimétisme avec les applications normales, au niveau des accès mémoire. Lors d'un chiffrement/déchiffrement, les émulateurs détectent les accès séquentiels à une zone mémoire donnée. La technique consiste alors à accéder aléatoirement à la zone contenant les données à chiffrer ou à déchiffrer ;
- la technologie d'évasion de saut de code (ou *Branching technology*). Là aussi, cette technique consiste à modifier un code de sorte à ce qu'il se comporte comme une application légitime, en terme de sauts. En effet, les programmes « normaux » possèdent de nombreux sauts conditionnels. Cela a pour conséquence essentielle que des portions de code sont fréquemment non exécutées lorsque la condition régissant le saut n'est pas réalisée. Or, dans un décrypteur classique, cela n'est pas le cas : le code de déchiffrement est linéaire et toujours exécuté.

La technologie d'évasion de saut de code [131] L'objectif est de produire un décrypteur dont le comportement est identique à celui d'un programme

[7] Ces techniques sont dues à un des membres du groupe 29A, connu sous le nom de *The Mental Driller* [131].

normal : la linéarité de l'exécution doit être brisée. L'idée est de produire un code de décrypteur tel que l'accès aux différentes instructions – lesquelles existent sous différentes formes équivalentes – se fait par des sauts aléatoires. La contrainte est que le flot d'exécution finale respecte bien l'algorithme de chiffrement/déchiffrement. Cette technologie utilise la notion de récursivité. Un arbre d'exécution est construit et non seulement chaque parcours du sommet à une feuille constitue une version du décrypteur, mais ce dernier peut également être construit en sélectionnant aléatoirement une branche à chaque niveau dans l'arbre. Cette sélection est réalisée au moyen d'instructions de saut conditionnels. D'où le terme dévasion de saut de code. Cela offre une richesse combinatoire d'autant plus grande que la profondeur de l'arbre (le niveau maximal de récursivité) est élevée. La fonction récursive *FaireBranche()* est décrite dans le tableau 6.1. Cette procédure va coder l'arbre lui-même.

Les principales étapes de codage du décrypteur sont les suivantes :

1. coder les instructions d'initialisation (registres de travail). Générer du code mort *(garbage code)* et appeler la procédure récursive *FaireBranche()* ;
2. la procédure *FaireBranche()* prend le contrôle et ne le rend que lorsque la totalité de l'arbre est codée. La fonction doit en permanence savoir à quel niveau de récursivité (compris entre 0 et $N-1$) elle travaille. Une variable `RECURS` est dont utilisée. Elle est incrémentée à l'entrée de la procédure et décrémentée en sortie ;
3. si le niveau de récursivité autorisé n'est pas atteint, alors une instruction du décrypteur est codée, et son adresse est mémorisée dans le tableau `TableDeSaut`. Du code mort aléatoire (en nature et en taille) est ensuite produit et codé dans le décrypteur ;
4. générer une instruction de saut conditionnel aléatoire du type `CMP Reg1, Reg2 / JA @xxx` ou `TEST REG, 2^i / J(N)Z @xxx` (la valeur i est aléatoire). L'idéal est de faire en sorte, quitte à utiliser le code mort généré à chaque étape, que la condition de saut soit réalisée avec une probabilité la plus proche possible de $\frac{1}{2}$. Comme la destination de cette instruction de saut n'est pas encore connue (l'arbre n'est pas terminé), son adresse est stockée sur la pile. La procédure *FaireBranche()* est alors appelée, le niveau maximal de récursivité n'étant pas atteint (création d'une nouvelle branche). Au retour de cette procédure, une branche complète a été codée. L'index d'insertion d'instruction pointe alors vers l'endroit où se trouvera la nouvelle branche. L'adresse de l'instruction de saut générée à ce niveau est récupérée de la pile (fonction `POP`). La distance entre cette adresse et l'index d'insertion courant est calculée et l'instruction de saut conditionnel dont l'adresse a été dépilée est complétée avec cette distance. La procédure *FaireBranche()* est appelée à nouveau (création d'une nouvelle branche dans l'arbre) ;
5. lorsque le niveau maximal de récursivité autorisé est atteint $(N-1)$, l'algorithme de chiffrement/déchiffrement proprement dit est codé (instruc-

TableDeSaut[NT] : table contenant les instructions du décrypteur (hors chiffrement) vers lesquelles le code peut sauter.
TableDeSautAResoudre[NR] : table contenant les sauts d'itération de chiffrement.
nt = 0 ; nr = 0 ;
$RECURS\_MAX = N - 1$; $RECURS = 0$;
Procédure FaireBranche( )

$RECURS++$;
Si ($RECURS! = RECURS\_MAX$) {
  Coder instruction du décrypteur ;
  TabledeSaut[NT] = @instruction ;
  $NT++$;
  Générer du code mort aléatoire ;
  Générer un saut conditionnel aléatoire ;
  $PUSH@SAUT$;
  FaireBranche( ) ;
  $RECURS--$;
  $POP@SAUT$;
  Calculer Distance($NT, @SAUT$) ;
  Compléter instruction saut avec Distance($NT, @SAUT$) ;
  FaireBranche( ) ;
  $RECURS--$;
  Retour ; }
Sinon {
    Coder une version de l'algo de chiffrement ;
    TableDeSautAResoudre[$NR++$] = $@SAUTFINBOUCLE$;
    Générer du code mort aléatoire ;
    Coder instruction de saut final de boucle ;
    Retour ; }
$RECURS--$;
}

Table 6.1 – Fonction récursive pour la technologie d'évasion de saut de code

tion de chiffrement, gestion du compteur de boucle...) en insérant du code-mort aléatoire.

Le point essentiel réside dans l'obligation de coder cet algorithme de manière différente pour chacune des branches (soit au total $2^{\text{RECURS\_MAX}}$; voir les exercices en fin de chapitre). Dans le cas contraire, du code identique dans plusieurs branches constituerait une signature exploitable par un antivirus. Le code de chiffrement se termine par un test et un saut (par exemple `DEC ECX / JNZ DECRYPT_VIRUS`). Mais le codage de l'arbre n'étant pas achevé, il n'est pas encore possible de renseigner la destination du saut (boucle de chiffrement/déchiffrement) si l'on veut exploiter toutes les versions de l'algorithme (une par branche). Dans ce but, on mémorise dans le tableau `TableDeSautAResoudre` l'adresse du saut de fin de boucle de chiffrement. Du code mort aléatoire est ensuite généré et l'instruction de saut est codée puis insérée. On sort finalement de la procédure *FaireBranche()* ;

6. la phase finale, une fois l'arbre totalement codé, consiste à compléter les instructions de saut conditionnel de fin de boucle de chiffrement/déchiffrement (tableau `TableDeSautAResoudre`). Elle est essentielle pour assurer un comportement dynamique du décrypteur comparable à celui d'un programme normal. Pour ce faire, on associe à chaque entrée dans le tableau `TableDeSautAResoudre` une entrée aléatoirement choisie dans le tableau `TableDeSaut`. Les sauts en fin de boucle de chiffrement pointent donc vers une destination aléatoire de l'arbre.

L'expérience montre que la technologie d'évasion de saut de code parvient à leurrer efficacement les antivirus actuels, dès lors que le niveau de récursivité maximal autorisé est élevé. En effet, tout émulateur devra analyser l'arbre avec une complexité en $\mathcal{O}(2^{\text{RECURS\_MAX}})$. Au-delà d'un certain temps, les émulateurs abandonnent (cela rejoint la technique de blindage par τ-obfuscation présentée dans les sections 8.2.3 et 8.6). En revanche, un niveau de récursivité maximal autorisé élevé implique de considérer des algorithmes de chiffrement/déchiffrement plus complexes (pour disposer d'un nombre d'instructions plus élevé). Les algorithmes cryptologiques modernes (par flot ou par bloc) conviennent parfaitement.

En conclusion, outre cette technologie d'évasion de saut de code, un grand nombre de techniques évoluées de réécriture existent pour concevoir un polymorpheur. Nous avons présenté leur philosophie générale. Le lecteur trouvera dans [131] la description de l'implémentation de certaines de ces techniques : récursivité généralisée de code (extension de la technologie d'évasion de saut de code à tout le programme), génération efficace de code mort... Ces techniques sont illustrées dans le code source du virus *W32.Tuareg*[8].

[8] Le code source de ce virus est disponible sur `http://vx.netlux.org`

Le décrypteur : polymorphisme par chiffrement

Ce type de polymorphisme consiste à chiffrer une partie du code. Chaque changement de clef produira, à partir d'un même code en clair, un cryptogramme différent et donc une forme différente. Cependant, la gestion du déchiffrement impose de conserver des éléments fixes, utilisables par les antivirus (en particulier, les informations directes ou indirectes liées à la clef utilisée ; le virus doit en effet pouvoir se déchiffrer lui-même). Ces contraintes font que les techniques polymorphes par chiffrement, pour la plupart, sont très mal utilisées ou ont une portée très limitée. En conséquence, les antivirus parviennent très souvent à traiter beaucoup de codes polymorphes par chiffrement connus[9]. Cela tient essentiellement aux raisons suivantes :

- les algorithmes de chiffrement généralement utilisés n'offrent qu'une très faible sécurité cryptologique. Ils utilisent des fonctions arithmétiques triviales (`XOR, ADD, SUB...`) utilisant des arguments constants. La cryptanalyse de tels systèmes est triviale – quand la clef n'est pas disponible ; voir par exemple [25]). Plus récemment, des virus utilisant des systèmes de chiffrement offrant un très haut niveau de sécurité (PGP, RC5, RSA, IDEA...) ont fait leur apparition. Cependant, la gestion de la clef rend leur analyse puis leur détection possible, au final ;
- la gestion de la clef est inexistante (la clef est dans le code, camouflée ou non) ou, quand elle est absente, son entropie est limitée ce qui autorise une recherche exhaustive directe ou indirecte (après une phase de réduction de l'espace des clefs fournie par une analyse préliminaire ; le lecteur pourra consulter [78] pour une telle analyse dans un contexte non viral) ;
- la présence de données chiffrées dans un code, même si elle ne permet pas l'analyse (voir chapitre 8) ne manquera pas d'être détectée du fait de sa forte entropie (voir section 3.6.4). Une politique de sécurité peut consister à filtrer et analyser systématiquement de tels contenus à forte entropie. La solution est alors d'abaisser fortement cette entropie par des techniques de simulabilité. Mais il est également possible de considérer des aspects TRANSEC (voir section 7.4).

Initialement, les premiers virus chiffrés comportaient un décrypteur fixe, lequel a été immédiatement utilisé par les antivirus comme signature. L'évolution naturelle a été de faire varier le décrypteur par l'adjonction d'un polymorpheur (voir section précédente). Mais même dans ce cas-là, beaucoup de codes polymorphes restent encore détectables. Nous allons voir dans la section suivante pourquoi.

Le lecteur pourra consulter [15] pour une présentation didactique d'un moteur polymorphique basique, mais néanmoins efficace, en assembleur.

[9] Rappelons au lecteur que les antivirus ne détectent pas tous les virus et que la connaissance des éditeurs se limite aux cas identifiés. Des codes malveillants identifiés par l'auteur, « à la main », sur des systèmes réels ne sont toujours pas détectés plus d'un an après par la totalité des antivirus !

6.3.2 Les techniques de métamorphisme

Les techniques de métamorphisme datent de l'année 2000. Évolution logique du polymorphisme face aux évolutions techniques antivirales, le métamorphisme représente un saut technologique majeur dans le domaine de l'algorithmique virale. Ce sont actuellement les techniques virales les plus complexes à imaginer et à implémenter. Elles réclament une véritable réflexion avant toute mise en œuvre. C'est la raison pour laquelle il existe peu de virus métamorphes élaborés et efficaces (moins d'une quinzaine). La plupart sont heureusement des preuves de concept.

Mais le terme de « métamorphisme » est souvent mal utilisé et mal compris. Beaucoup n'y voient que des techniques, certes plus complexes, destinées à faire muter l'ensemble du code. Autrement dit, pour reprendre le formalisme présenté dans la section 6.1, les différentes formes mutées d'un virus métamorphes doivent avoir des noyaux rigoureusement différents. Cependant ce processus de mutation ne doit pas se limiter au seul code – la partie exclusivement formelle du programme – mais doit s'étendre à sa structure entière de sorte à changer également le flot d'exécution. D'une certaine manière, le métamorphisme consiste à changer simultanément de mots et de grammaire, là où le polymorphisme ne changeait que les mots.

Il est important de conserver à l'esprit que les techniques métamorphes n'ont pas pour but d'interdire l'analyse de code. C'est là le rôle des techniques de blindage, présentées dans le chapitre 8. Leur objectif, comme le polymorphisme, est de sinon rendre impossible, du moins de compliquer à l'extrême les techniques d'analyse automatique (par un moteur antiviral, par exemple). De ce point de vue, ces techniques sont proches de celles de la τ-obfuscation définie dans la section 8.2.3. En revanche, les techniques métamorphes, pour la majorité de celles actuellement connues, ne résisteront jamais, contrairement aux techniques de blindage, à l'analyse humaine. Il existera toujours des programmeurs et analystes de codes compétents capables – certes avec quelquefois du temps, voire beaucoup de temps – de percer à jour et de comprendre ces techniques.

Le point essentiel concernant un moteur de métamorphisme est que le code à la génération t construit et génère le code de la génération $t + 1$. Cela impose de gérer en parallèle deux codages différents, et bien sûr de faire en sorte qu'il n'y ait pas d'interférences entre les deux. En pratique, dans un code métamorphe, près de 90 à 95 % du code sont consacrés au moteur de polymorphisme lui-même. Nous allons présenter les techniques métamorphes en illustrant les principaux concepts utilisés avec le virus *Win32/Linux.MetaPHOR*[10], qui est probablement le code métamorphe le plus sophistiqué et le plus élégant connu à ce jour.

[10] Ce virus à été conçu par un programmeur du nom de *The mental driller*, à titre de preuve de concept. Étant donné la taille du code et sa complexité, nous n'en donnerons que les extraits les plus pertinents. Le lecteur pourra analyser le code source disponible sur `http://vx.netlux.org`.

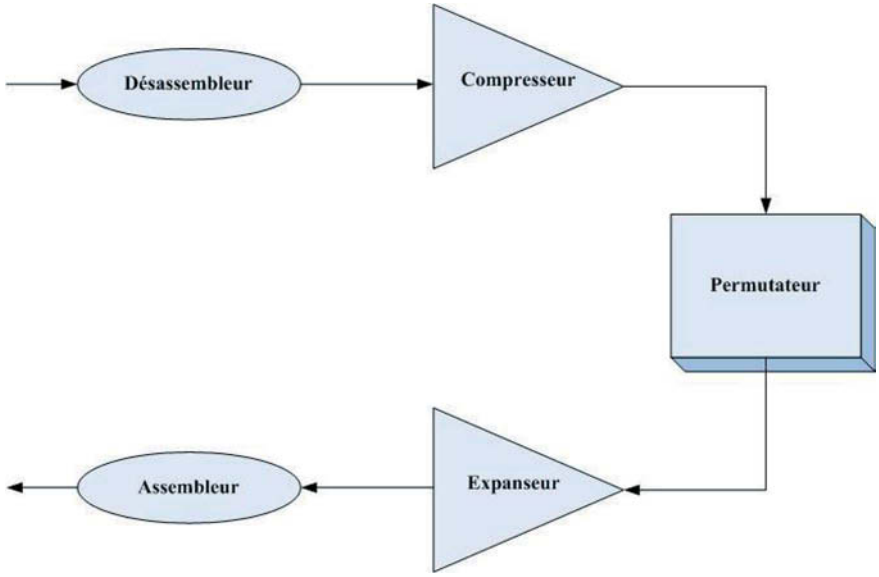


Figure 6.3 – Structure générale d'un moteur métamorphe

Structure d'un moteur métamorphe

Un tel moteur comprend plusieurs parties (voir figure 6.3), lesquelles peuvent être optionnelles. Tout dépendra des degrés de sophistication et de complexité souhaités. Ces parties sont dans l'ordre :

1. un module de désassemblage interne ou *désassembleur*. L'exécutable étant présent sous forme codée, selon des conventions[11] propres à chaque moteur, pour ne pas en révéler sa nature par une simple lecture, il est nécessaire de le décoder au préalable. Cette étape est essentielle ne serait-ce que pour déterminer la longueur du programme lui-même et toutes les informations qui le caractérisent. Ce désassembleur est capable de décoder tous les types d'instructions. En outre, il doit également dépermuter le code qui a été préalablement permuté. Cela consiste essentiellement à supprimer un grand nombre de fonctions de saut ou d'appel ;
2. un module de compression ou *compresseur*. Le code, une fois désassemblé doit être réduit. Cela a pour but de contrôler et de limiter la taille du virus, de mutation en mutation. En effet, beaucoup de techniques provoquent un allongement non désiré de la taille du code. La plupart des virus métamorphes n'intègrent pas cette phase de compression, ce qui provoque non seulement une explosion de la taille du code, mais également génère des effets de bords néfastes pour le code, en termes de

[11] Ces conventions ou codages peuvent elles-mêmes être variables d'une mutation à une autre.

détection. La compression se fait au niveau des instructions. Deux ou plusieurs instructions sont remplacées par une seule. Il s'agit en quelque sorte d'une optimisation de code. Une fonctionnalité d'émulation de code peut également être intégrée. Elle permet de supprimer les instructions redondantes ou inutiles ;

3. un module de permutation ou *permutateur*. Les instructions sont permutées pour empêcher toute analyse de forme (voir section 2.3.1). Cette opération est essentiellement mise en œuvre à l'aide de fonctions de saut ;
4. un module d'expansion ou *expanseur*. Contrepoint du module de compression, ce module effectue le travail inverse. Autrement dit, il recode chaque instruction en plusieurs instructions, pour une même action. Le couple compresseur/expanseur doit être pensé et implémenté avec soin pour éviter une augmentation incontrôlée du code. Des instructions redondantes, vides ou inutiles *(garbage code)* sont insérées ;
5. un module d'assemblage ou *assembleur* interne. Les informations sont recodées (si le code utilisé est le même) ou codées (si le code utilisé lui-même change) tout en gérant les fonctions dynamiques par relogeabilité et recalcul d'adresses (instructions de saut, d'appels...), en modifiant la longueur des instructions, les registres utilisés...

Le langage d'assemblage

L'auteur du code *Win32/Linux.MetaPHOR* a tout d'abord conçu son propre langage de pseudo-assembleur. Ce langage reprend la philosophie des opcodes x86. Si ce langage est toujours le même d'une mutation à une autre, une évolution naturelle de *MetaPHOR* serait de considérer une mutation du langage lui-même[12].

L'intérêt de passer par un langage de pseudo-assemblage pendant le processus de mutation, est de contrarier fortement l'analyse du code par les antivirus. Le code n'existe en mémoire que sous ce langague, inconnu par ces derniers. Le principe fondamental dans la construction des opcodes de ce nouveau langage est de ne produire aucune ambiguïté lors du décodage/assemblage. Autrement dit, il doit y avoir une correspondance univoque entre chaque instruction du pseudo-assembleur et chaque instruction assembleur du processeur cible. L'auteur a donc défini un vocabulaire pour ce langage (les nouveaux opcodes) et une grammaire (la façon d'utiliser et de combiner les opcodes). Ainsi, par exemple, avec le codage suivant pour certaines instructions de base :

```
00: ADD, 08: OR, 20: AND, 28: SUB, 30: XOR,
38: CMP, 40: MOV, 48: TEST
```

et les règles suivantes, consistant à rajouter des octets

[12] Il serait également intéressant de coupler cette mutation du langage assembleur avec les techniques de simulabilité présentées dans la section 3.6.4 exploitant les caractéristiques statistiques du langage assembleur x86.

```
+00: Reg,Imm
+01: Reg,Reg
+02: Reg,Mem
+03: Mem,Reg
+04: Mem,Imm
+80: operandes sur 8 bits
```

l'opcode $83H$ désigne l'instruction `ADD Mem, Reg` dont les opérandes sont de huit bits, tandis que l'opcode $A1H$ désigne, toujours sous huit bits, l'instruction `OR Reg, Reg`. Le lecteur pourra constater (voir le code source du moteur du virus *Win32/Linux.MetaPHOR*) que ces opcodes ne peuvent pas désigner d'autres instructions : le codage est univoque[13]. Les autres opcodes de base sont les suivants (extraits) :

```
50 : PUSH Reg
51 : PUSH Mem
58 : POP Reg
59 : POP Mem
68 : PUSH Imm
E0 : NOT Reg
E1 : NOT Mem
E2 : NOT Reg8
E3 : NOT Mem8
E4 : NEG Reg
E5 : NEG Mem
E6 : NEG Reg8
E7 : NEG Mem8
E8 : CALL label
E9 : JMP label
F0 : SHIFT Reg,Imm
F1 : SHIFT Mem,Imm
F2 : SHIFT Reg8,Imm
F3 : SHIFT Mem8,Imm
```

```
70-7F : saut conditionnels
EA : CALL Mem (appels d'API)
EB : JMP Mem (obfuscation des appels d'API)
EC : CALL Reg (obfuscation des appels d'API)
ED : JMP Reg (obfuscation des appels d'API)
.........
```

La structure générale d'une instruction pour ce langage pseudo-assembleur est la suivante :

```
  XX     XX XX XX XX XX XX XX XX XX XX  XX  XX XX XX XX
Opcodes  <-- arguments instruction -->  LM  <-pointeur->
```

La valeur `LM` *(Label Mark)* vaut 1 si une autre instruction pointe sur cette instruction. Cela sert notamment lors du processus de compression : deux instructions ne peuvent être remplacées par une seule si une troisième instruction pointe sur la deuxième. Enfin, le pointeur indique la position vers laquelle pointait cette instruction (valeur du registre `EIP` dans la codification originale). Cela

[13] Dans le contexte des langages formels, la grammaire est dite non ambiguë.

permet de mémoriser le statut antérieur de cette instruction et facilite ainsi, entre autres choses, la gestion des instructions de saut, d'appels....

En résumé, le virus crée un langage (toujours le même) et génère lors de chaque mutation un assembleur/désassembleur différent pour la prochaine forme mutée. La difficulté tient au fait que deux versions différentes de ce assembleur/désassembleur coexistent en permanence, celui de la génération t qui génère celui de la génération $t+1$.

Le module de désassemblage/dépermutation

Le code, dont le point d'entrée est dans le registre `ESI`, doit dans un premier temps être désassemblé et nettoyé. Une zone mémoire est réservée (variable `PathMarks`), de la taille du code. Elle sert de zone de gestion du code sous sa forme désassemblée. Deux tables sont également créées, chacune indexée par un compteur qui lui est propre :

- la table `LabelTable` dont l'objectif est de pouvoir référencer simultanément les instructions du code sous leur forme assemblée et désassemblée. Elle contient donc des éléments de 2 DWORD (8 octets). Le premier concerne l'EIP *Extended Instruction Pointer* actuel, alors que le second référence le code désassemblé. Lorsqu'une instruction est traitée, elle est immédiatement référencée dans et vers cette table selon son statut. Ainsi toutes les références d'instruction vers d'autres instructions (fonctions de saut ou d'appel, par exemple) sont automatiquement actualisées lors du processus de désassemblage ;
- la table `FutureLabelTable`. Elle sert à mémoriser les cibles de fonctions de saut ou d'appels, dans le code désassemblé, faisant référence à des portions de codes qui ne le sont pas encore. À chaque traitement d'une instruction, le code vérifie si cette adresse n'est pas référencée dans cette table, et le cas échéant les informations concernant cette adresse sont mémorisées.

Ces deux tables sont donc judicieusement conçues et utilisées pour gérer simultanément le code avant et après désassemblage, en particulier pour mémoriser en permanence les références entre les deux versions du code (instructions de saut ou d'appel). Le principe de base est d'interchanger les labels d'instructions avec les pointeurs vers les entrées de la table `LabelTable` selon que l'instruction est un saut non conditionnel ou un saut conditionnel.

Au final, le désassemblage a permis :

- de décoder la totalité du code dans le langage de pseudo-assembleur ;
- d'éliminer la totalité du code qui ne peut être atteint ;
- d'éliminer les permutations d'instructions et les instructions de saut utilisées pour permuter le code ;
- de substituer aux labels d'instructions des pointeurs vers les entrées de la table `LabelTable`.

L'algorithme étant plutôt complexe à décrire, illustrons la partie consacrée à la dépermutation par un exemple parlant issu de [132].

```
    CODE              PASSES                CODE FINAL DÉSASSEMBLÉ
   ------            ------                ----------------------
    xxx1          1) décodé en xxx1                xxx1
    xxx2          2) décodé xxx2                   xxx2
    xxx3          3) décodé xxx3                   xxx3
    jmp @A        4) change EIP en @A
    yyy1          5) décodé xxx7                   xxx7
    yyy2          6) décodé xxx8                   xxx8
@B: xxx4          7) décodé xxx9                   xxx9
    xxx5          8) change EIP en @B
    xxx6          9) décodé xxx4                   xxx4
    jmp @C       10) décodé xxx5                   xxx5
    yyy3         11) décodé xxx6                   xxx6
    yyy4         12) change EIP en @C
@A: xxx7         13) décodé xxx10                  xxx10
    xxx8         14) décodé xxx11                  xxx11
    xxx9         15) décodé JZ et                  JZ @D
                     stockage du label
                     @D dans la table
                     FutureLabelTable
    jmp @B       16) decode xxx12                  xxx12
@D: xxx13        17) decode RET, obtient           RET
                     @D de la table
                     FutureLabelTable et
                     actualise l'instruction
                     JZ (passe 15 ; @D = EIP
                     en cours)
    xxx14
    RET          18) décodé xxx13            @D: xxx13
    yyy5         19) décodé xxx14                 xxx14
@C: xxx10        20) décodé RET (la table         RET
                     FutureLabelTable est
                     vide, fin du traitement).
    xxx11
    jz @D
    xxx12
    RET
```

Le lecteur peut constater que toutes les instructions de code inutile ou mort (instructions de type yyy*) ont été naturellement éliminées et que des blocs de code ont été permutés. Mais la structure du code (le squelette) reste peu ou prou modifiée et est donc susceptible de constituer un élément invariant utilisable pour la détection (par exemple voir [130, §11.2.2]). L'auteur a donc imaginé un procédé élégant pour que la mutation affecte réellement la structure globale du programme. Pour cela, tout le processus de mutation repose sur la

gestion simultanée du code courant (obtenu par désassemblage à la génération t), de celui de la version mutée produite à la génération t et du code qui sera désassemblé lors de la génération $t+1$.

À titre d'exemple, une des méthodes permettant de réaliser cette opération consiste à transformer (par le module de compression) des instructions de saut conditionnels en sauts inconditionnels ou l'inverse. Mais ces derniers n'existeront que dans la version désassemblée du code de la génération suivante. Ainsi, reprenons l'exemple précédent (sans le code mort) dans lequel la première instruction de saut (inconditionnel) sera remplacée par un saut conditionnel.

```
    CODE          CODE DÉSASSEMBLÉ
  -------        -----------------
    xxx1             xxx1
    xxx2             xxx2
    xxx3             xxx3
    CMP X,X          CMP X,X
    JZ @A            JZ @A
@B: xxx4       @B:   xxx4
    xxx5             xxx5
    xxx6             xxx6
    jmp @C           xxx10
@A: xxx7             xxx11
    xxx8             jz @D
    xxx9             xxx12
    jmp @B           RET
@D: xxx13      @A:   xxx7
    xxx14            xxx8
    RET              xxx9
@C: xxx10            jmp @B
    xxx11      @D:   xxx13
    jz @D            xxx14
    xxx12            RET
    RET
```

Le couple d'instruction `CMP X,X ; JZ @A` sera remplacé par une instruction `JMP @A`, lors de la phase de compression à la génération t, et sera donc naturellement éliminée lors du désassemblage à la génération $t+1$. Le lecteur, en comparant les deux versions, constatera que la structure générale a été profondément modifiée. Il suffit de savoir qu'un grand nombre de tels changements peuvent avoir lieu de manière décalée, ainsi le retour à une structure déjà rencontrée dans une génération antérieure n'est possible qu'avec une très faible probabilité. Néanmoins, le code étant fini, au bout d'un certain nombre de générations, il est inévitable qu'une structure réapparaise. Le tout est que cela se produise après un nombre suffisamment grand de générations.

À noter que lors de cette phase de désassemblage/permutation, les instructions peuvent également être remplacées par des formes équivalentes. Ainsi, la

procédure suivante :

```
;INC Reg

  @@Op_INC:   and     eax, 7          ; registre de INC
              mov     [edi+1], eax    ;
              xor     eax, eax        ; pseudo-opcode de ADD
              jmp   @@Op_GenINCDEC

;DEC Reg

  @@Op_DEC:   and     eax, 7          ; registre de DEC
              mov     [edi+1], eax    ;
              mov     eax, 28h        ; pseudo-opcode de SUB

@@Op_GenINCDEC:
              mov     edx, [edi]      ; Met le pseudo-opcode
                                      ; (ADD ou SUB)
              and     edx, 0FFFFFF00h
              and     eax, 0FFh
              add     eax, edx
              mov     [edi], eax
              mov     eax, 1          ; Met l'opérande
                                      ; d'addition/subtraction
              mov     [edi+7], eax
              add     esi, 1          ; EIP suivant
             jmp   @@NextInstruction
```

remplace l'instruction `INC Reg` (respectivement `DEC Reg`) par l'instruction `ADD Reg, 1` (respectivement `SUB Reg, 1`).

Il est intéressant de noter que ce désassemblage ne se fait pas de manière linéaire, ce qui pourrait être utilisé par certaines techniques antivirales. Il est au contraire aléatoire.

Le module de compression/émulation

L'objectif est de supprimer tous les effets d'obfuscation. Le principe est ici simple. Il s'agit de remplacer une ou plusieurs instructions par une seule. Le code est ainsi compressé. C'est ce module qui très certainement constitue la partie la plus faible du moteur du virus *Win32/Linux.MetaPHOR* (voir section 6.3.2). Ces transformations sont de trois types :

- remplacement d'une instruction unique par une instruction équivalente. Voici quelques exemples :

```
XOR Reg,-1              --> NOT Reg
MOV Reg,Reg             --> NOP
SUB Mem,Imm             --> ADD Mem,-Imm
```

```
XOR Reg,0                --> MOV Reg,0
ADD Mem,0                --> NOP
OR  Reg,0                --> NOP
AND Mem,-1               --> NOP
AND Reg,0                --> MOV Reg,0
XOR Reg,Reg              --> MOV Reg,0
SUB Reg,Reg              --> MOV Reg,0
OR  Reg,Reg              --> CMP Reg,0
AND Reg,Reg              --> CMP Reg,0
TEST Reg,Reg             --> CMP Reg,0
LEA Reg,[Imm]            --> MOV Reg,Imm
LEA Reg,[Reg+Imm]        --> ADD Reg,Imm
LEA Reg,[Reg2]           --> MOV Reg,Reg2
LEA Reg,[Reg+Reg2]       --> ADD Reg,Reg2
LEA Reg,[Reg2+Reg2+xxx]  --> LEA Reg,[2*Reg2+xxx]
MOV Reg,Reg              --> NOP
```

– remplacement de paires d'instructions par une instruction équivalente. Voici quelques exemples :

```
PUSH Imm / POP Reg        --> MOV Reg,Imm
PUSH Reg / POP Reg2       --> MOV Reg2,Reg
PUSH Reg / POP Mem        --> MOV Mem,Reg
PUSH Mem / POP Reg        --> MOV Reg,Mem
MOV Mem,Reg/PUSH Mem      --> PUSH Reg
POP Mem / MOV Reg,Mem     --> POP Reg
POP Mem2 / MOV Mem,Mem2   --> POP Mem
MOV Mem,Imm / PUSH Mem    --> PUSH Imm
MOV Mem,Imm / OP Reg,Mem  --> OP Reg,Imm
POP Mem / PUSH Mem        --> NOP
MOV Mem,Reg / CALL Mem    --> CALL Reg
MOV Mem,Reg / JMP Mem     --> JMP Reg
MOV Mem2,Mem / JMP Mem2   --> JMP Mem
OP Reg,xxx / MOV Reg,yyy  --> MOV Reg,yyy
NOT Reg / NEG Reg         --> ADD Reg,1
NOT Reg / ADD Reg,1       --> NEG Reg
NOT Mem / NEG Mem         --> ADD Mem,1
NOT Mem / ADD Mem,1       --> NEG Mem
NEG Reg / NOT Reg         --> ADD Reg,-1
NEG Reg / ADD Reg,-1      --> NOT Reg
NEG Mem / NOT Mem         --> ADD Mem,-1
NEG Mem / ADD Mem,-1      --> NOT Mem
TEST X,Y / != Jcc         --> NOP
POP Mem / JMP Mem         --> RET
PUSH Reg / RET            --> JMP Reg
MOV Reg,Mem / CALL Reg    --> CALL Mem
```

La seconde instruction supprimée (par compression) est remplacée par une instruction NOP ;

- remplacement de triplets d'instructions par une instruction équivalente. Voici quelques exemples :

```
MOV Mem,Reg
OP Mem,Reg2
MOV Reg,Mem                --> OP Reg,Reg2

MOV Mem,Reg
OP Mem,Imm
MOV Reg,Mem                --> OP Reg,Imm

MOV Mem2,Mem
OP Mem2,Reg
MOV Mem,Mem2               --> OP Mem,Reg

MOV Mem2,Mem
OP Mem2,Imm
MOV Mem,Mem2               --> OP Mem,Imm

CMP Reg,Reg
JO/JB/JNZ/JA/JS/JNP/JL/JG @xxx
!= Jcc                     --> NOP

CMP Reg,Reg
JNO/JAE/JZ/JBE/JNS/JP/JGE/JLE @xxx
!= Jcc                     --> JMP @xxx

MOV Mem,Imm
CMP/TEST Reg,Mem           --> CMP/TEST Reg,Imm
Jcc @xxx                       Jcc @xxx

MOV Mem,Reg
SUB/CMP Mem,Reg2           --> CMP Reg,Reg2
Jcc @xxx                       Jcc @xxx
```

Les deux dernières instructions supprimées (par compression) sont remplacées par deux instructions NOP.

Les instructions nulles produites (NOP) sont conservées car elles serviront lors de la phase d'expansion de code. En effet, une telle instruction peut être retransformée dans le sens inverse en une instruction différente.

Pour illustrer l'action du module de compression, considérons le code (avant compression) suivant.

```
MOV [Var1], ESI   PUSH ESI        MOV EAX,ESI
PUSH [Var1]       nop             nop
POP EAX           POP EAX         nop
PUSH EBX          PUSH EBX        PUSH EBX
POP [Var2]        POP [Var2]      POP [Var2]
ADD EAX,[Var2]    ADD EAX,[Var2]  ADD EAX,[Var2]
```

Nous obtenons alors le code compressé suivant (utilisation d'une émulation partielle en fin de code) :

```
MOV EAX,ESI       MOV EAX,ESI   LEA EAX,[ESI+EBX]
nop               nop           nop
nop               nop           nop
MOV [Var2],EBX    ADD EAX,EBX   nop
nop               nop           nop
ADD EAX,[Var2]    nop           nop
```

Le module de permutation

L'objectif est ici de brouiller le code tout en réorganisant ses interdépendances (saut, test, appels...). L'auteur de *Win32/Linux.MetaPHOR* effectue ce brouillage par blocs : le code est découpé en blocs puis les blocs sont permutés. Si le registre `ESI` contient l'offset de la première instruction et si le registre `EDI` contient celui de la dernière instruction, considérons le pseudo-code suivant, pour le découpage du code en différents blocs (on notera *Binf_ i* et *Bsup_ i* les limites inférieure et supérieure du bloc *i*) :

```
i = 1;
Tant que (ESI < EDI) alors
  Binf_i = ESI;
  /* Ajout d'un octet aléatoire */
  ESI = ESI + Random(16);
  Bsup_i = ESI;
  i = i + 1;
  Si ((ESI + 0FH) > EDI)
   Binf_i = ESI;
   Bsup_i = ESI;
   break;
  Fin si
Fin Tant que
```

Voici un exemple de code (à gauche, et sa version permutée par blocs à droite) :

```
ESI = 00000000h et EDI = 00000060h

DD   00000000h,0000000Ah          DD   00000032h,0000003Dh
DD   0000000Ah,00000017h          DD   00000023h,00000032h
```

```
DD   00000017h,00000023h permutation  DD   0000000Ah,00000017h
DD   00000023h,00000032h ----------> DD   00000000h,0000000Ah
DD   00000032h,0000003Dh              DD   00000017h,00000023h
DD   0000003Dh,00000049h              DD   00000052h,00000060h
DD   00000049h,00000052h              DD   0000003Dh,00000049h
DD   00000052h,00000060h              DD   00000049h,00000052h
```

Une première instruction de saut vers le bloc contenant le point d'entrée du code est installée, puis d'autres pour aller de bloc en bloc. Ainsi, par exemple, après l'instruction à l'offset `0000003DH`, le programme saute vers l'instruction située à l'offset `00000023h` et ainsi de suite. Toutes les informations de sauts (conditionnels ou non) sont mémorisées dans une table avant permutation, puis réutilisées pour transposer l'information à la version permutée du code.

Le module d'expansion

Ce module effectue le travail inverse de celui du module de compression. De façon aléatoire, il remplace chaque instruction par une ou plusieurs autres instructions selon les modèles dont quelques exemples ont été présentés dans la section consacrée au compresseur. Les transformations sont faites de manière récursive, c'est-à-dire que chaque instruction impliquée dans le résultat de l'expansion est elle-même expansée jusqu'à un certain niveau de récursivité, dont la valeur est contenue dans la variable `SizeOfExpansion`, et comprise entre 0 et 3.

Ainsi l'instruction `PUSH Reg` peut une première fois être expansée en

```
MOV Mem, Reg
PUSH Mem
```

mais la première instruction peut ensuite être à son tour expansée, ce qui finalement produit

```
PUSH Reg
POP Mem
PUSH Mem
```

et ainsi de suite jusqu'à atteindre la valeur limite de récursivité. À chaque fois, la décision de poursuivre récursivement l'expansion est aléatoire.

Ce module effectue également d'autres opérations, afin d'accroître le plus possible la variabilité du code, parmi lesquelles :

- la translation de registres. Le but est que les opérations et accès mémoire utilisent des registres à chaque fois différents. Ne sont pas concernés les registres `EAX`, `ECX` or `EDX` (utilisés pour les appels à API) et le registre `ESP` (utilisé pour la gestion de la pile) ;
- la variation des adresses mémoire. Là aussi, il s'agit de faire varier les adresses utilisées en mémoire, pour une grande variabilité. Les instructions utilisant des adresses mémoire sont recensées et référencées dans une

table des variables. Cette table est permutée et les adresses des variables sont réattribuées.

Le lecteur trouvera dans le code quelques autres techniques d'expansion.

Le module d'assemblage

Dernier module, mais essentiel, il est quasiment couplé au précédent dans la mesure où les instructions produites par l'expanseur sont codifiées directement pour pouvoir ensuite être directement assemblées. Par exemple, si l'expanseur doit traiter l'instruction `CMP EAX, 0`, il produira directement une instruction existant dans le langage de pseudo-assemblage (voir plus haut) soit `OR EAX, EAX` soit `TEST EAX, EAX`. Le réassembleur traduira en opcodes (`21H` ou `49H`), sans se soucier de la signification des instructions. Le code ainsi produit est prêt pour une copie directe dans l'hôte en cours d'infection.

Toutes les instructions de type saut (conditionnels ou non) sont également gérées par utilisation d'une table contenant toutes ces instructions qui devront ensuite être complétées lorsque l'assemblage sera terminé, pour prendre en compte leurs destinations. Deux cas se présentent :

- les sauts concernent un retour un arrière dans le code. Ce cas est facilement géré car la longueur du saut est connu (différence d'adresses), la destination ayant déjà été assemblée ;
- les sauts concernent un code en aval. Comme il s'agit d'instructions non encore assemblées, le réassembleur ne peut savoir par avance s'il s'agit d'un saut long ou court. Il regarde alors si l'instruction pointe au-delà de 128 octets ou non :

```
@@Assemble_Jump_Fowards:
   mov  ebx, eax   ; Calcul de la distance
   sub  ebx, esi
   cmp  ebx, 0B0h  ; 11 * 0Bh = 121.
                   ; Il faut que ce soit < 128
                   ; pour un saut court
   jbe   @@Assemble_JmpFwd_Short
```

Si cela est le cas, l'instruction de saut est codée aléatoirement court ou long (long avec une probabilité de $\frac{1}{8}$) :

```
@@Assemble_JmpFwd_Short:
   call Random     ; produit un nombre aléatoire
   and  eax, 7     ;
   or   eax, eax   ; vaut 0 dans 1 cas sur 8
   jz   @@Assemble_JmpFwd_Long_Set00
   mov  eax, [esi] ; Opcode
   and  eax, 0FFh
   cmp  eax, 0E8h  ; est-ce un CALL ?
   jz   @@Assemble_JmpFwd_Long_Set  ; (pour plus
                                    ;  de sureté)
```

```
cmp  eax, 0E9h                    ; est-ce un JMP ?
jz   @@Assemble_JmpFwd_Short_JMP ; alors court
```

Le moteur polymorphe *Win32/Linux.MetaPHOR*

Le virus inclut un module de chiffrement/déchiffrement (décrypteur) du code assez classique du moins dans sa philosophie. Le décrypteur est quasiment construit en totalité par les modules d'expansion et d'assemblage. Il varie en taille et en position (effet du métamorphisme). Il réalise le chiffrement du virus avec une probabilité de $\frac{15}{16}$. Sinon le code n'est pas chiffré. Le chiffrement utilisé est assez basique (choix aléatoire entre les fonctions `ADD`, `XOR`, `SUB`). La clef est générée aléatoirement et stockée dans le code.

Le déchiffrement du code, à l'exécution, est fortement irrégulier. Le module de chiffrement/déchiffrement utilise la technique PRIDE (*Pseudo-Random Index DEcryption*) [131,143]. Cette technique permet d'accéder de manière aléatoire à la mémoire, afin de tromper les techniques antivirales par émulation de code qui ainsi concluront à des accès mémoire d'une application légitime. Cette technique, mise au point par l'auteur du virus *Win32/Linux.MetaPHOR*, se résume par le pseudo-code donné dans le tableau 6.2.

Aléa(N) : produit un nombre aléatoire compris entre 0 et $N-1$ (N entier).
Entrée : M, la taille des données chiffrées (doit être de la forme 2^m), IV = Aléa(M) et D adresse de début des données chiffrées.

Registre1 = Alea(M)
Registre2 = IV
Tant que Registre2 $\neq IV$ faire
Dechiffrer [(Register1 $\oplus$ Register2) + D]
Registre1+ = (Alea(M) & -2) ;
Registre1 = Registre1 modulo M ;
Registre2 + + ;
Registre2 = Registre2 modulo M ;
Fin Tant que
Aller en D

Table 6.2 – Algorithme PRIDE pour le déchiffrement aléatoire en mémoire

Dans cet algorithme, le registre 1 modifie le registre 2, lequel est réellement aléatoire. Le registre 2 sert en fait de simple compteur. La seule contrainte est que sa valeur reste confinée entre 0 et D (la taille de données chiffrées). L'intérêt

de cet algorithme tient au fait qu'il utilise de manière optimale les fonctions modulo 2^n, lesquelles ont de très intéressantes propriétés cryptographiques. En effet, les itérations successives assurent avec ces fonctions, qu'avant de produire à nouveau la valeur IV, toutes les valeurs comprises entre 0 et 2^m auront été générées une fois et une seule (voir exercice).

En outre, le décrypteur varie en taille (effet du métamorphisme) et en position (effet du métamorphisme).

Avantages et force du métamorphisme

Nous avons présenté les principaux ressorts algorithmiques du moteur du virus *Win32/Linux.MetaPHOR*. Il y en a beaucoup d'autres, notamment au niveau de l'implémentation (par exemple la gestion des appels aux API ou la gestion mémoire). Mais la philosophie essentielle a été ici résumée.

Le premier intérêt de ce moteur est sa capacité à produire des codes capables d'infecter aussi bien sous Windows que sous Linux, en parcourant les disques montés dans un système ou les unités de type réseau (sous Windows) :

- pour Windows, les exécutables de type PE, selon deux modes possibles : entrelacement ou en fin de code (type *appender*) ;
- sous Linux, les exécutables de type ELF, par ajout d'une section supplémentaire dans le segment de données. L'infection intervient avec une probabilité de 0,5 dans le répertoire courant, et remonte jusqu'à trois niveaux au-dessus.

Un désassembleur est capable de s'adapter à n'importe quel processeur et système d'exploitation et ainsi de produire des codes multi-plateformes.

Si le moteur *MetaPHOR* est puissant et produit des codes réellement métamorphes – en vertu notamment d'une utilisation massive du non-déterminisme –, il est encore possible d'améliorer sensiblement ses capacités dans ce domaine. L'analyse de ce code, comme celui d'autres codes métamorphes, montre que conceptuellement il subsiste néanmoins un certain nombre d'aspects invariants ou variant peu :

- le langage de pseudo-assemblage est toujours le même. Une évolution puissante consiste à générer un nouveau langage pour chaque nouvelle génération. Il est alors nécessaire de disposer d'un module dédié à cette mutation, qui, par exemple, pourrait intervenir juste après le module de désassemblage. La mutation de ce pseudo-langage peut consister en une simple permutation des opcodes. Certes, la complexité du code augmentera en conséquence mais il en résultera une meilleure efficacité en terme de lutte anti-antivirale ;
- un grand nombre de choix dans le code de *MetaPHOR* sont aléatoirement faits selon une probabilité de $\frac{1}{8}$ alors que d'autres le sont avec une probabilité égale à $\frac{1}{2}$. La première probabilité, si elle est facile à programmer, n'est pas optimale notamment en ce qui concerne les choix dans la nature des sauts. En effet, elle laisse la place à une invariabilité encore importante. Une probabilité rigoureusement égale à $\frac{1}{2}$ est de loin préférable.

Mais les techniques métamorphes présentées ici, comme toutes celles actuellement connues, possèdent un point faible. Il est résumé dans le fait technique suivant, issu de [81] : *« un virus métamorphe doit être capable de se désassembler et de faire de l'auto-rétroconception. »* Cela signifie que pour pouvoir agir, il est contraint de n'utiliser que des techniques, notamment d'obfuscation, faciles à inverser. Autrement dit, selon ces auteurs, le virus est contraint aux mêmes règles que les antivirus. Ce postulat n'est pas immuable comme nous le verrons avec la τ-obfuscation présentée dans les sections 8.2.3 et 8.6. Cette technique, utilisée dans un virus comme *Win32/Linux.MetaPHOR*, peut lourdement contrarier la lutte antivirale.

6.4 Polymorphisme, grammaires formelles et automates finis

Les techniques polymorphes finissent par être toujours, sinon détectées, du moins potentiellement détectables. La nuance est fondamentale car elle signifie que si les spécialistes de la lutte antivirale sont capables de mettre au point des algorithmes de détection efficaces, une fois les techniques polymorphes analysées et comprises, ces algorithmes ne sont pas tous forcément implémentés dans les logiciels antivirus ou, s'ils le sont, ils sont limités au traitement des instances faciles du problème général[14]. Un antivirus est avant tout un produit commercial et ce produit ne doit pas constituer une gêne pour l'utilisateur. C'est cette constatation qui a conduit à imaginer certaines techniques de blindage (voir le chapitre 8).

Le fait que la plupart des techniques de polymorphisme connues finissent – ne serait-ce qu'en théorie[15] – par être gérées au niveau antiviral tient à leur nature profonde. Elles correspondent à des instances faciles – au sens de la théorie de la complexité – d'un problème concernant la reconnaissance de langages formels. Nous allons présenter la mutation de code (polymorphisme/métamorphisme) et sa détection sous cet angle. Les résultats établis permettront ensuite de montrer dans quels cas un virus peut ou ne peut pas être détecté efficacement.

6.4.1 Grammaires formelles

Définissons quelques concepts préliminaires.

Définition 6.7 *(Alphabet et chaînes de symboles) Soit un ensemble non vide fini* Σ *appelé* alphabet *dont les éléments sont dénommés* symboles. *Soit* $\Sigma =$

[14] Le meilleur exemple est probablement la technologie *Armadillo* assurant la protection d'exécutables. Alors qu'elle met en œuvre des techniques polymorphes relativement classiques, la manière dont elle les utilise met régulièrement en défaut la plupart des antivirus.
[15] Rappelons que les techniques qui mettent en échec les antivirus restent finalement inconnues.

$\{a_1, a_2, \ldots, a_n\}$. *Une* chaîne *sur* Σ *est une séquence de symboles de* Σ, *soit* $b_1 b_2 b_3 \ldots b_m$ *avec* $b_i \in \Sigma$ *et* $m \geq 0$. *La chaîne vide* ϵ *est l'unique chaîne telle que* $m = 0$.

Soient deux chaînes $x = b_1 b_2 \ldots b_m$ et $y = c_1 c_2 \ldots c_n$. Les deux chaînes sont dites égales si et seulement si $m = n$ et $b_i = c_i$ pour tout $i \in \{1, \ldots, m\}$. La concaténation des chaînes x et y se note $xy = b_1 b_2 \ldots b_m c_1 c_2 \ldots c_n$. Si A et B sont deux ensembles de chaînes définies sur Σ, alors les ensembles suivants sont définis :

$$\begin{aligned} AB &= \{xy | x \in A, y \in B\}, \\ A^* &= \{x_1 x_2 \ldots x_n | n \geq 0, x_1, x_2, \ldots, x_n \in A\}, \\ A^+ &= \{x_1 x_2 \ldots x_n | n \geq 1, x_1, x_2, \ldots, x_n \in A\}. \end{aligned}$$

Ces notations et définitions de base nous permettent à présent de définir la notion fondamentale de *grammaire formelle.*

Définition 6.8 *(Grammaire formelle) Une* grammaire formelle *G est le quadruplet $G = (N, T, S, R)$ où :*

- *N est un ensemble de symboles non terminaux ;*
- *T est un alphabet de symboles dits terminaux, avec $N \cap T = \emptyset$;*
- *$S \in N$ est le symbole de début ;*
- *R, un système de réécriture, c'est-à-dire un ensemble fini de règles, $R \subseteq (T \cup N) * \times (T \cup N)*$, tel que $(u, v) \in R \Rightarrow u \not\in nT^*$ (on ne peut réécrire une chaîne ne contenant que des symboles terminaux).*

Le système de réécriture de chaînes (encore appelé *système semi-Thue*) sur un alphabet Σ est en fait un sous-ensemble fini de $\Sigma^* \times \Sigma^*$, autrement dit l'ensemble fini $R = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}$ de couples de chaînes de symboles de l'alphabet Σ. Une paire $(u, v) \in R$ est appelée *règle de réécriture* ou *production* et on note $u ::= v$ au lieu de $(u, v) \in R$.

Exemple 6.2 *Soit la grammaire G définie ainsi :*

- $N = \{X\}$;
- $T = \{x, y, z\}$;
- $R = \{(X, xXx), (X, yXy), (X, z), (X, xzx)\}$.
- $S = X$.

On synthétise l'ensemble R avec la notation suivante :

$$X ::= xXx | yXy | z | xzx,$$

où le signe | désigne le OU *logique (X est réécrit par l'une des valeurs possibles du membre droit de chaque production).*

Une grammaire permet ainsi de construire à partir du symbole de début S, des chaînes de symboles plus complexes, en substituant aux symboles non

terminaux (ici limités à X) les membres droits d'une règle ou production. Ainsi, sur l'alphabet $\Sigma = T$, on peut générer les chaînes suivantes :

$$xxzxx, xxxyzyxxx, yyxxzxxyy, yxxzxxy...$$

Il est possible de considérer des cas plus complexes avec un plus grand nombre de symboles terminaux, comme dans l'exemple suivant.

Exemple 6.3 *Considérons l'alphabet binaire $T = \Sigma = \{0,1\} \cup \{\epsilon\}$, l'ensemble $N = \{S, X, Y\}$ et le symbole de début S. Considérons le système de réécriture composé des productions suivantes :*

$$\begin{aligned} S &::= 1S|0S|X \\ X &::= 0Y \\ Y &::= 1Y|0Y|Z \\ Z &::= \epsilon \end{aligned}$$

Cette grammaire génère en fait les chaînes binaires contenant au moins un 0.

À partir de la définition 6.8, le système de réécriture R induit la relation suivante : soient x et y deux éléments de $(\Sigma \cup N)^*$, on dit que $x \Rightarrow y$ si et seulement si $x = x_1x_2x_3$ et $y = x_1y_2x_3$ pour des chaînes x_1, x_2, x_3, y_2 et la production (x_2, y_2). En d'autres termes, y est produit à partir d'un jeu de substitutions de chaînes dans x. On définit alors, à partir de cette relation, la fermeture réflexive et transitive de la relation $\Rightarrow$, que l'on notera $\Rightarrow^*$. Elle est définie de la manière suivante :

- $x \Rightarrow^0 y$ si et seulement si $x = y$;
- pour $k > 0$, alors $x \Rightarrow^k y$ si et seulement si pour un certain y', on a $x \Rightarrow^{k-1} y'$ et $y' \Rightarrow y$;
- enfin $x \Rightarrow^* y$ si et seulement si il existe un $k \geq 0$ tel que $x \Rightarrow^k y$.

Cette dernière relation permet de définir la notion de *langage formel.*

Définition 6.9 *Soit $G = (N, T, S, R)$ une grammaire formelle. Le langage formel généré par G est l'ensemble $L(G) = \{x \in \Sigma^* | S \Rightarrow^* x\}$.*

Les notions de grammaires formelles et de langages formels sont en fait une abstraction des notions traditionnelles de grammaire et de langue. Une grammaire est l'ensemble des règles permettant de construire un langage. Cela concerne les langues naturelles mais également les langages informatiques.

Très vite la théorie des grammaires formelles s'est attachée à classer les différents types de grammaires (et donc de langages). Cela a donné lieu à la célèbre classification de Noam Chomsky [21,22]. Cette classification détaille les catégories standards dans lesquelles sont réparties la plupart des grammaires formelles. Nous ne détaillerons pas les propriétés spécifiques à chacune d'entre elles – cela dépasserait le cadre de cet ouvrage – mais nous mentionnerons leurs caractéristiques essentielles – le lecteur pourra consulter [66, 68, 84] pour plus de détail. Cette classification comprend quatre types de grammaires.

- les grammaires de type 0 ou *grammaires dites sans restriction.* Ces grammaires regroupent toutes les grammaires formelles dont les productions peuvent être construites librement (sans restriction, d'où le terme). Elles sont de la forme $x ::= y$ où y est une chaîne quelconque formée de symboles de $N \cup T$. Ces grammaires génèrent tous les langages pouvant être reconnus par une machine de Turing (voir section 6.4.2), autrement dit les langages récursivement énumérables (voir [38, chapitre 2]) ;
- les grammaires de type 1 ou *grammaires contextuelles.* La seule contrainte pour les productions est que la taille des mots ne peut décroître. Par conséquent, les seules productions autorisées sont de la forme $x ::= y$ tant que $|y| \geq |x|$ (la taille de x reste inférieure ou égale a celle de y). Le terme de « contextuel » tient au fait que les productions sont de la forme (xXy, xzy) (X symbole terminal unique alors que x et y sont des éléments de $(N \cup T)^*$ et $z \in (N \cup T)^+$). Le fait que X soit remplacé par z ou non est déterminé par x et y (le contexte de X). Cette classe comprend tous les langages naturels ;
- les grammaires de type 2 ou *grammaire non contextuelles.* Les productions sont, pour ce type de grammaires, de la forme $X ::= y$ où X est un symbole terminal unique et y est un élément de $(N \cup T)^*$. Le terme X peut être réécrit indépendamment de son contexte, contrairement aux grammaires de type 1. Ces grammaires décrivent la plupart des langages de programmation ;
- les grammaires de type 3 ou *grammaires régulières.* Ce sont les grammaires dont les productions sont de la forme $X ::= x$ ou $X ::= xY$ avec $(X, Y) \in N^2$ et $x \in T^*$.

6.4.2 Polymorphisme et langages formels

La notion de grammaire et de langage formels offre une modélisation puissante de la notion de polymorphisme. Cette approche a été introduite pour la première fois dans [106]. Malheureusement, l'auteur de cette étude n'a fait qu'effleurer les relations entre polymorphisme et grammaire formelle. En partant des prémices présentées dans l'article original, nous allons développer ces relations et traiter sous cet angle de la détection des techniques polymorphes. La réalité est plus complexe et moins en faveur des antivirus que ne laisse penser l'auteur de [106]. Selon ce dernier, toutes les techniques polymorphes sont inévitablement condamnées à être gérées avec succès par les antivirus. Nous allons montrer qu'il faut malheureusement fortement relativiser cette affirmation.

Considérons l'ensemble des instructions x86 comme alphabet. Il est certes de taille importante mais néanmoins finie. Ces instructions peuvent être combinées selon des règles propres au compilateur. L'ensemble de ces règles est assimilable à une grammaire formelle de type 2 et le langage assembleur est le langage engendré par cette grammaire. La programmation d'un moteur polymorphe, et en particulier de sa partie la plus importante c'est-à-dire le générateur de code mort *(garbage generator)*, consiste en fait à générer un langage, que nous

nommerons langage polymorphe, avec sa grammaire propre.

Soit un moteur polymorphe décrit par la grammaire

$$G = \{\{A, B\}, \{a, b, c, d, x, y\}, S, R\}.$$

Les instructions a, b, c et d représentent des instructions de code mort *(garbage code)* tandis que les instructions x et y représentent les instructions réelles du décrypteur[16]. Le système de réécriture R peut alors être de la forme suivante :

$$\begin{aligned} S &::= aS|bS|cS|xA \\ A &::= aA|bA|cA|dA|yB \\ B &::= aB|bB|cB|dB|\epsilon \end{aligned}$$

Le langage polymorphe généré par cette grammaire est constitué des mots de la forme :

$$\{a, b, c, d\}^* x \{a, b, c, d\}^* z \{a, b, c, d\}^*.$$

Ces mots représentent les différentes versions mutées du décrypteur. Il est alors facile de voir, pour cette grammaire, que le mot `abcddxd` n'appartient pas à ce langage contrairement au mot `adcbxaddbydab`.

Tout le problème des antivirus consiste à disposer d'un algorithme permettant d'identifier de manière générique tout mot du langage polymorphe. Mais que devient cette « facilité » dans le cas de moteurs polymorphes réels et plus complexes. Là est précisément tout l'intérêt de la formalisation des moteurs polymorphes à l'aide des grammaires formelles. Car selon la nature de la grammaire utilisée par ce moteur, le problème de l'appartenance d'un mot (entendons d'une version mutée) à un langage (le langage polymorphe) est plus ou moins complexe à résoudre.

6.4.3 Détection et reconnaissance de langages

Donnons une première définition pour formaliser les choses.

Définition 6.10 *Soit une grammaire $G = (N, T, S, R)$ et soit une chaîne $x \in T^*$. Le* problème d'appartenance *relativement à G consiste à déterminer si $x \in L(G)$ ou non. Le* problème de complétude *de G consiste à déterminer si $L(G) = T^*$ ou non.*

Le premier problème modélise directement le problème de la détection d'un moteur polymorphe, une fois ce dernier connu (autrement dit une fois la grammaire associée connue). Le second permet de définir autrement la notion de non détection et de fausse alarme (voir exercices).

Afin de traiter le premier problème, présentons les outils formels et algorithmiques permettant de la résoudre. Cela permettra ensuite de donner les

[16] Par exemple $x =$ `XOR [EDI], AL` et $y =$ `INC EDI`.

principaux résultats concernant la complexité générale de ce problème et d'expliquer ce que l'on peut en déduire finalement en matière de détection. Nous allons considérer la notion d'*automate* comme outil générique de résolution du problème d'appartenance.

Définition 6.11 *(Automate fini déterministe) Un* automate fini déterministe *est défini par le 5-uplet* $(Q, \Sigma, \tau, q_0, F)$ *où :*

- Q *est l'ensemble des états possibles de l'automate ;*
- Σ *est un alphabet fini ;*
- τ *est la fonction de transition* $\tau : Q \times \Sigma \rightarrow Q$ *qui à un état et à un symbole associe un état ;*
- q_0 *est l'état initial ;*
- F *est un sous-ensemble d'états, appelés* états acceptables, *autrement dit réalisables par l'automate.*

C'est la forme la plus simple d'automate. Elle se représente par un graphe dirigé dont les nœuds décrivent les états et les arcs sont étiquetés par des symboles de Σ. Ces derniers décrivent la condition permettant de passer d'un état à un autre. Il existe un très grand nombre de type d'automates finis déterministes.

Exemple 6.4 *[68] Soit* $S = \{1, 2, 3, 4\}$, $\Sigma = \{a, b, c\}$, $q_0 = 1$ *et* $F = \{4\}$. *La fonction* τ *est définie par :*

$$\begin{aligned} \tau(1, c) &= \{2\} \\ \tau(2, a) &= \{2\} \\ \tau(2, b) &= \{3\} \\ \tau(3, a) &= \{3\} \\ \tau(3, c) &= \{4\} \end{aligned}$$

La figure suivante montre la représentation graphique de cet automate.

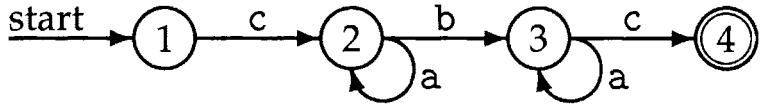


Figure 6.4 – Automate déterministe simple

Considérons à présent une généralisation des automates finis déterministes.

Définition 6.12 *(Automate fini non déterministe) Un* automate fini non déterministe *est défini par le 5-uplet* $(Q, \Sigma, \tau, q_0, F)$ *où :*

- *Q est l'ensemble des états possibles de l'automate ;*
- *Σ est un alphabet fini ;*
- *τ est la fonction de transition $\tau : Q \times (\Sigma \times \{\epsilon\}) \rightarrow \mathcal{P}(Q)$ qui à un état et à un symbole (éventuellement la chaîne vide) associe un sous-ensemble (non nécessairement un singleton) d'états ;*
- *q_0 est l'état initial ;*
- *F est un sous-ensemble d'états, appelés* états acceptables, *autrement dit réalisables par l'automate.*

Dans le cas d'un automate non déterministe, contrairement à leur équivalent déterministe, plusieurs arcs avec la même étiquette (un symbole de Σ) peuvent partir d'un nœud. Cela implique qu'à partir d'un état donné, une même condition peut produire des effets différents. Il en résulte une richesse plus grande mais également une complexité plus importante d'un point de vue calculatoire[17].

Exemple 6.5 *[68] Soit $S = \{1,2,3,4,5\}$, $\Sigma = \{x,y,z,\epsilon\}$, $q_0 = 1$ et $F = \{3,5\}$. La fonction de transition est alors définie par $\tau(1,x) = \{2,4\}, \tau(2,y) = \{3\}, \tau(4,z) = \{5\}$ et $\tau(1,\epsilon) = \{3\}$. La figure suivante montre la représentation graphique de cet automate.*

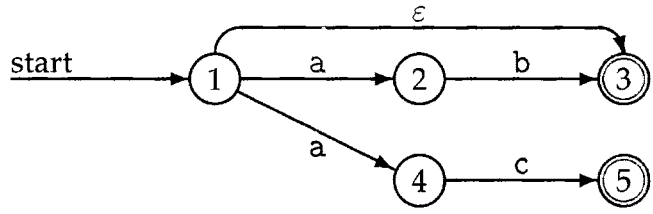


Figure 6.5 – Automate non déterministe

Ces outils vont nous permettre de définir l'action (potentielle) d'un antivirus. Nous nous placerons dans le cas général des automates finis non déterministes dans la mesure où il est facile de montrer qu'un tel automate peut se ramener à une version déterministe. Cependant, cette conversion peut produire un

[17] Le lecteur aura peut être noté le parallèle existant entre les automates et les machines de Turing, ces dernières existant également sous forme non déterministe. Ce parallèle est non seulement pertinent mais également essentiel pour distinguer certaines classes de complexité. Notons que la notion d'automate, au sens large, peut être vue comme un cas particulier de machines de Turing, qui elles constituent un modèle théorique global et universel de la notion de calcul. Les principales différences entre automates et machines de Turing résident essentiellement dans le fait que les premiers ne peuvent envisager que des ensembles finis de chaînes de Σ^*, tandis que les secondes considérent naturellement des chaînes infinies. Par conséquent, pour tout automate fini, il existe toujours une machine de Turing calculant la même fonction alors que l'inverse n'est bien sûr pas vrai.

automate déterministe ayant un nombre exponentiellement plus grand d'états que la version non déterministe [99].

Définition 6.13 *On dit qu'une chaîne* $x = x_1 x_2 \dots x_n$ *avec* $x_i \in \Sigma$ *est acceptée par un automate* $A = (Q, \Sigma, \tau, q_0, F)$ *s'il existe une séquence d'états* $q_1, q_2, \dots, q_{n+1}$ *de* Q *et une séquence de symboles* $x_1, x_2, \dots, x_n$ *de* $\Sigma \cup \{\epsilon\}$ *telles que* $q_{i+1} \in \tau(q_i, x_i)$ *pour tout* $i \in \{1, 2, \dots, n\}$ *avec* $q_0 = q_1$. *Alors, on note* $L(A)$ *l'ensemble de toutes les chaînes acceptées par* A. *C'est le* langage accepté *par l'automate* A. *On dit qu'un automate* A *décide qu'une chaîne* $x = x_1 x_2 \dots x_n$ *avec* $x_i \in \Sigma$ *est acceptable pour une grammaire* G *– ou le langage associé* $L(G)$ *– si* $L(A) = L(G)$. *L'automate* A *est alors une solution pour le problème d'appartenance relativement à la grammaire* G.

Cette définition permet de voir que, dès lors qu'un antivirus contient un automate A capable de résoudre le problème d'appartenance à une grammaire polymorphe, il est en mesure de détecter toutes les formes mutées produites par le moteur de polymorphisme réalisant ce langage. Le problème tient au fait que la complexité calculatoire de cet automate diffère selon le type de grammaire que ce moteur utilise. En outre, chaque fois que la grammaire polymorphe change, il faut changer l'automate correspondant. De ce point de vue, un virus métamorphe efficace arrive sans peine à mettre en échec les antivirus. En effet, chaque mutation métamorphe produit à la fois une nouvelle grammaire et un mot du langage généré par cette grammaire. Un virus métamorphe peut donc être décrit comme un langage de grammaires, même si ce concept n'est pas encore formellement défini.

Définition 6.14 *(Virus métamorphe) Soit une grammaire* $G_1 = (N, T, S, R)$ *et une grammaire* $G_2 = (N', T', S', R')$ *où l'ensemble* T' *est un ensemble de grammaires formelles,* S' *est la grammaire* G_1 *et* R' *un système de réécriture défini sur* $(N' \cup T')^*$. *Un virus métamorphe est représenté par* G_2 *et chaque forme mutée du virus est un mot de* $L(L(G_2))$.

Cette définition est intuitive et exprime le fait que, d'une manière similaire à la production des mots d'un langage, les grammaires sont générées à partir d'une grammaire initiale et de productions permettant de construire d'autres grammaires. Nous verrons une manière, parmi certainement de nombreuses autres, d'implémenter cela en pratique avec le moteur métamorphe PBMOT présenté dans la section 6.4.4.

Les gestion du métamorphisme impose donc de disposer d'automates capables de résoudre le problème d'appartenance pour des grammaires de type G_2. Déterminer le type de cette grammaire relativement à la classification de Chomsky est un problème ouvert. L'intuition semble militer en faveur du type 0.

Détecter un virus polymorphe ou un virus métamorphe consiste donc à disposer d'un automate capable de décider le langage généré par une grammaire

donnée. Donnons maintenant quelques résultats concernant la complexité du problème de l'appartenance selon la grammaire considérée.

Proposition 6.1 *Le problème d'appartenance :*
- *est indécidable pour une grammaire de type 0 ;*
- *appartient à la classe NP pour les grammaire de type 1 et 2 ;*
- *appartient à la classe P pour les grammaires de type 3.*

Preuve.
Nous ne donnerons pas la preuve de cette proposition. Le lecteur pourra consulter [99] ou [64]. ■

Concernant les grammaires de type 0, on démontre qu'elles génèrent les langages récursivement énumérables (les productions décrivent et simulent alors les actions d'une machine de Turing). Décider alors, pour G et $x \in \Sigma$ donnés, que $x \in L(G)$ se ramène au problème de l'arrêt d'une machine de Turing, lequel est généralement indécidable. Pour les grammaires de type 1 et 2, les automates permettant de résoudre le problème d'appartenance sont du type non déterministe alors que ceux traitant le cas des grammaires de type 3 sont déterministes.

Ces résultats montrent que le choix de la grammaire pour écrire un moteur polymorphe a une grande incidence sur sa détection (potentielle) future. La plupart des moteurs polymorphes connus relèvent des grammaires régulières (classe 3). Ils sont donc facilement détectables (classe polynomiale). En revanche, et cela relativise fortement les affirmations de [106] concernant la détection systématique des moteurs polymorphes, la détection des techniques polymorphes relevant des autres grammaires a une complexité au minimum exponentielle voire est indécidable. Cela signifie en pratique que cette détection est illusoire, les antivirus ne pouvant s'offrir le luxe de consacrer des ressources en temps trop importantes.

Ce qui nous sauve actuellement est que les programmeurs de codes malveillants semblent ignorer quelles sont les « bonnes » grammaires à utiliser. Mais cela durera-t-il ? Enfin, si les travaux menés actuellement au laboratoire de virologie et de cryptologie montrent que ces grammaires sont plus complexes à envisager et à manipuler, ils ont permis de confirmer, à ce jour, le potentiel formidable résidant dans les langages qui en relèvent... et toute l'inquiétude qu'il faut concevoir pour l'avenir.

Il existe un grand nombre de résultats théoriques [18] dans le domaine des langages formels sur lesquels s'appuyer si l'on veut réaliser un polymorphisme viral impossible à appréhender par les antivirus (de manière absolue ou calculatoirement). Les plus intéressants sont ceux concernant l'indécidabilité de certains problèmes. L'un des résultats généraux les plus intéressants de ce point de vue concerne le théorème de Rice. Soit une propriété P sur les langages. On dit que P est non triviale s'il existe au moins un langage récursivement énumérable (type 0) L vérifiant P et au moins un langage récursivement énumérable L' ne vérifiant pas P. Le théorème de Rice [18,68] est alors le suivant.

Théorème 6.4 *(Théorème de Rice) Pour toute propriété non triviale P sur les langages, le problème de savoir si le langage $L(M)$ d'une machine de Turing M vérifie P est indécidable.*

Ce théorème indique donc clairement dans quel contexte se placer pour contrer les antivirus. Nous verrons dans la section 6.4.4 un autre aspect lié aux grammaires formelles et aux systèmes de réécriture, permettant de concevoir des vers potentiellement indétectables.

Pour terminer avec les grammaires formelles, il est intéressant de donner le résultat suivant, dont la preuve sera trouvée dans [68, §10.4].

Théorème 6.5 *Soient deux grammaires non contextuelles $G_i = (N_i, T_i, S_i, R_i)$ avec $i = 1, 2$. Décider si $L(G_1) \cap L(G_2) = \emptyset$ ou non est un problème indécidable. Décider pour $G = (N, T, S, R)$ une grammaire non contextuelle si $L(G) = T^*$ est un problème indécidable.*

Ces deux résultats, dans le cadre des grammaires non contextuelles, illustrent d'une autre manière le modèle statistique de l'indécidabilité du problème de la détection virale. Le premier illustre la notion de fausse alarme (la grammaire G_1 est « non virale » alors que la grammaire G_2 l'est) alors que le second concerne le problème de la non détection (voir exercices).

6.4.4 Mutation absolue à détection indécidable

Pour clore ce chapitre consacré aux techniques de mutation de code, nous allons présenter un concept de mutation de code fondé sur un problème généralement indécidable. Ce concept repose sur le problème du mot que nous allons présenter. Cette approche a été validée avec le moteur métamorphe *preuve-de-concept* PBMOT, dont nous présenterons les principaux mécanismes. En particulier, le concept de grammaire, présenté dans la définition 6.14, a été mis en œuvre avec un certain succès, même si cette étude n'en est qu'à la phase initiale et que de nombreux problèmes ouverts demeurent.

6.4.5 Le problème du mot

Le problème du mot a été formalisé par Émile Post en 1950 [103]. Ce problème, avec le problème de l'arrêt étudié par A. Turing, est l'un des plus célèbres exemples de problème généralement indécidable. De manière résumée, ce problème consiste à décider si deux mot finis r et s définis sur un alphabet Σ sont équivalents relativement à un ensemble de règles de réécriture R. Définissons formellement ce problème (le lecteur se référera à la section 6.4.2 pour les notations communes aux grammaires formelles).

Définition 6.15 *(Système semi-Thue) Un* système de réécriture *(encore appelé* système semi-Thue*) sur un alphabet Σ est un sous-ensemble fini de $\Sigma^* \times \Sigma^*$,*

c'est-à-dire un ensemble $R = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}$ *de couples de chaînes de* Σ^* *appelées* règles de réécriture *ou* productions.

Une production se note généralement $u ::= v$. Un système de réécriture R permet alors de définir une relation de réécriture (encore appelée relation de réduction), notée $\Rightarrow_R$ et définie, par

$$rus \Rightarrow rvs \text{ si et seulement si } (u, v) \in R \text{ et } (r, s) \in \Sigma^* \times \Sigma^*.$$

Cette relation signifie que l'on peut obtenir la chaîne $rvs \in \Sigma^*$ directement (ou à l'aide d'une seule règle ou encore en une seule étape) à partir de la chaîne $rus \in \Sigma^*$.

Exemple 6.6 *[68] Soit l'alphabet* $\Sigma = \{A, a, b, c\}$ *et soit*

$$R = \{(A, aAa), (A, bAb), (A, c), (A, aca)\}.$$

Alors nous avons

$$\begin{aligned} A &\Rightarrow_R aAa \\ aAa &\Rightarrow_R aaAaa \\ aaAaa &\Rightarrow_R aacaa \end{aligned}$$

Cette relation permet de définir alors la clôture réflexive et transitive de la relation $\Rightarrow$. Nous la noterons $\Rightarrow_R^*$. Elle est définie, pour tout r, g, h de Σ^*, par :

1. si $g \Rightarrow_R h$ alors $g \Rightarrow_R^* h$
2. $g \Rightarrow_R^* g$
3. Si $g \Rightarrow_R^* r$ et $r \Rightarrow_R^* h$ alors $g \Rightarrow_R^* h$.

Autrement dit, deux mots sont liés par cette relation si, en un nombre fini de règles, l'un peut être produit à partir de l'autre.

Exemple 6.7 *Selon l'exemple précédent, nous avons*

$$\begin{aligned} A &\Rightarrow_R^* A \\ A &\Rightarrow_R^* aAa \\ A &\Rightarrow_R^* aacaa \end{aligned}$$

Avec ces notations et concepts, donnons à présent une définition rigoureuse du problème du mot.

Définition 6.16 *(Problème du mot) Soit un système de réécriture* R *sur un alphabet* Σ *et deux chaînes* r *et* s *de* Σ^*. *Le problème du mot consiste à décider si* $r \Rightarrow_R^* s$.

Dans tout ce qui suit, nous omettrons la référence au système utilisé quand il n'existe aucune ambiguïté et nous noterons la relation simplement $\Rightarrow^*$.

Nous avons alors le théorème fondamental suivant.

Théorème 6.6 *[103] Le problème du mot relativement à un système semi-Thue est indécidable.*

Preuve.
Nous ne donnerons pas la preuve ici. Le lecteur la trouvera soit dans la publication originale [103] soit dans [68, §10.2.2] soit dans [9, pp. 571-577]. Le ressort essentiel de la preuve consiste à se ramener au problème de l'arrêt étudié par A. Turing, lequel est indécidable. ■

Le lecteur aura remarqué que les productions d'un système de réécriture de type semi-Thue sont ordonnées. En d'autres termes, $u \Rightarrow v$ n'implique pas obligatoirement que $v \Rightarrow u$. Nous allons spécifier la notion de réécriture aux systèmes pour lesquels il y a équivalence.

Définition 6.17 *L'inverse d'une production (u, v) est la production (v, u). Un système de réécriture de type semi-Thue R est un* système de Thue *si pour toute production (u, v) de R alors la production (v, u) est aussi dans R.*

Dans un système de Thue, la relation $\Rightarrow_R^*$ est symétrique. On la note alors $\Leftrightarrow_R^*$. Post a montré que le problème du mot était également indécidable relativement à un système de Thue.

Exemple 6.8 *(Système Thue de Tzeitzin) Soit le système R suivant défini sur un alphabet $\Sigma = \{a, b, c, d, e\}$.*

$$
\begin{array}{rl}
(ac, ca), & \textit{commutation} \\
(ad, da), & \\
(bc, cb), & \\
(bd, db), & \\
(eca, ce), & \\
(edb, de), & \\
(cca, ccae) & \textit{effacement/insertion}
\end{array}
$$

Ce système semi-Thue est appelé système de Tzeitzin [133]. C'est le plus petit semi-Thue indécidable. Nous le noterons T_0. Tout système semi-Thue R contenant T_0 est donc lui-même indécidable. Il existe d'autres systèmes Thue indécidables. Citons par exemple un autre système de Tzeitzin [133] que nous

noterons T_1 :

$$(ac, ca),$$
$$(ad, da),$$
$$(bc, cb),$$
$$(bd, db),$$
$$(eca, ce),$$
$$(edb, de),$$
$$(cdca, cdcae),$$
$$(caaa, aaa),$$
$$(daaa, aaa)$$

Il existe dans la littérature d'autres systèmes de Thue, comparables aux systèmes de Tzeitzin, mais ces derniers seront utilisés dans ce qui suit, étant donné qu'ils sont les plus simples à appréhender.

6.4.6 Mutation de code et problème du mot : le moteur PBMOT

L'idée maîtresse est d'utiliser une grammaire dont le système de réécriture est un système de Thue contenant le système de Tzeitzin ou un autre système indécidable. Cela implique que, d'un point de vue général, la détection du moteur de mutation relèvera d'un problème indécidable. Pour compliquer encore plus les choses et ainsi réaliser du métamorphisme, nous allons implémenter le concept de la définition 6.14, dont nous reprendrons ici les notations. Autrement dit, les règles de réécriture seront elles-mêmes réécrites de mutation en mutation. Pour cela deux contraintes seront respectées :

- le sysème de réécriture de G_2 contient un système de Thue indécidable ;
- chaque grammaire (mot) générée par $L_i(G_2)$, lors de la mutation i, contient un système de Thue indécidable.

Du point de vue de l'implémentation, le principe central consiste à identifier le système de réécriture des grammaires $L_i(G_2)$ à un mot de l'alphabet $(N \cup T)^*$ où les ensembles T et N sont ceux des grammaires G_1^i. Autrement dit, l'ensemble R de règles suivant (de la grammaire G_1^i) :

$$R = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}$$

est transformé en le mot suivant :

$$u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n, \tag{6.1}$$

formé de symboles terminaux et non terminaux. Il est ensuite nécessaire de construire la grammaire G_2 capable de manipuler ces mots. L'ensemble T' est formé des mots construits sur $(N \cup T)^*$ (expression 6.1). L'ensemble N' est

formé de symboles spécifiques à la grammaire G_2 mais des symboles de N peuvent être également considérés. L'élément S' est un mot de $(N' \cup T)^*$. Il reste à définir le système de réécriture R' manipulant les mots de $(N \cup T)^*$ avec la contrainte $R' \supset T_0$ ou $R' \supset T_1$.

Si le principe général d'une grammaire de type G_2 est relativement simple à appréhender, en revanche sa construction effective est techniquement complexe. Nous ne l'aborderons pas ici par manque de place – plusieurs dizaines de pages seraient nécessaires – et pour ne pas donner une technique « clef en main ». Nous donnerons les deux principes essentiels :

- le code final doit être pensé en terme fonctionnel et non pas en terme de code (instructions machine) – ce qui est essentiel, ce n'est pas tant la forme des instructions mais leur articulation et leur interrelations. Si les règles de réécriture ne sont pas triviales, la mutation du code d'un point de vue formel (au niveau des opcodes) se fera toute seule. D'un point de vue technique, cela signifie que les règles doivent autant modifier l'ordre des $u_i v_i$, et ce profondément, que les couples (u_i, v_i) dans le mot $u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n$;
- le code doit être organisé et pensé en procédures, même si au final le codage n'est pas structuré ainsi.

Certains exercices en fin de chapitre suggéreront d'autres éléments concernant la construction du système R'. La nature des règles et leur sophistication auront un impact direct sur le caractère de détectabilité du moteur les mettant en œuvre.

Pour le moteur PBMOT, le code du moteur *MetaPHOR* a été pris comme point de départ. Les étapes sont décrites ci-après.

1. Les différents modules du moteur (voir section 6.3.2) ont été analysés. L'alphabet T a été construit dans un premier temps : il correspond à peu de choses près aux différentes instructions possibles.
2. Cette analyse s'est ensuite attachée à identifier les grandes fonctionnalités mises en œuvre au niveau des fonctions de transformation du code. Un proto-système de réécriture R'_0 est défini dans un premier temps. Il a été élaboré de sorte à inclure le système T_0. Le rôle de R'_0 est de réaliser la mutation proprement dite du code. En d'autres termes, la réécriture concerne les (u_i, v_i) des mots $u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n$. À ce stade, l'ensemble N n'est pas encore défini.
3. L'analyse a ensuite porté sur la modification des fonctions de transformation de code (le point central du métamorphisme). Cela a permis en premier lieu de construire l'ensemble N des symboles non terminaux, qui pratiquement permettent de modifier la structure globale des mots du type $u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n$. En second lieu, le proto-système R'_0 est modifié pour produire le système de réécriture final R', lequel inclut un système T_1.

Le point critique concerne la transformation d'un mot

$$u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n$$

en un système

$$R = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}.$$

En effet, les réécritures successives peuvent conduire à des variations de taille des sous-mots u_i et v_i. Il est alors nécessaire de notariser toutes ces variations. Enfin la gestion de la réécriture doit prendre en compte, de manière analogue au moteur *MetaPHOR*, la gestion des sauts conditionnels ou non.

Nous pouvons alors énoncer le résultat suivant.

Proposition 6.2 *La détection d'un virus utilisant le moteur métamorphe* PBMOT *est en général indécidable.*

Preuve.
Chaque version mutée ν_i consiste en un mot de type

$$L(L_i(G_2)) = L(u_1^i v_1^i u_2^i v_2^i \ldots u_{n-1}^i v_{n-1}^i u_n^i v_n^i).$$

Détecter ce virus consiste donc à décider si deux mots

$$\nu_i = u_1^i v_1^i u_2^i v_2^i \ldots u_{n-1}^i v_{n-1}^i u_n^i v_n^i$$

et

$$\nu_j = u_1^j v_1^j u_2^j v_2^j \ldots u_{n-1}^j v_{n-1}^j u_n^j v_n^j,$$

avec $j > i$, sont tels que $\nu_i \Leftrightarrow_{G_2}^* \nu_j$. Or la grammaire G_2 contient les systèmes T_0 et T_1, lesquels sont indécidables. D'où le résultat. ■

Remarque. La proposition précédente concerne la détection par analyse de forme. Ce résultat implique que tout espoir de détection doit considérer une autre approche, autrement dit la détection comportementale. Dans ce dernier cas, il n'est pas sûr actuellement que les antivirus soient capable de gérer efficacement un moteur comme PBMOT (voir chapitre 2).

6.5 Conclusion

La protection par mutation de code permet de fortement contrarier les antivirus voire d'interdire leur détection en pratique. Encore faut-il choisir les techniques efficaces. L'approche théorique par les grammaires formelles est une voie d'exploration très prometteuse pour les identifier. Les codes polymorphes/métamorphes actuels – en tout cas ceux qui sont connus – sont en fait décrits par des grammaires pour lesquelles la détection reste « facile ».

Les techniques de mutation de code, qu'elles soient polymorphes ou métamorphes, ne visent somme toute qu'à contrarier avec plus ou moins de succès la détection par analyse de forme. En revanche, les comportements d'un code peuvent consituer un invariant utilisable par un antivirus. La prochaine évolution logique concerne donc le polymorphisme/métamorphisme fonctionnel : les actions de base (comportements) voire l'action globale du code malveillant

changent d'une forme mutée à une autre. Si conceptuellement il n'y a pas une grande différence avec la mutation de forme – c'est principalement là l'intérêt de la notion de stratégie de détection (voir section 2.7.1) –, la mise en œuvre technique est un peu plus complexe. Les travaux en cours au laboratoire de virologie et de cryptologie de l'École Supérieure et d'Application des Transmissions et les premiers résultats démontrent non seulement la validité de cette approche mais également les inquiétudes à concevoir dans la maîtrise de ce genre de technologie – si ce n'est déjà le cas. Les antivirus actuels sont totalement inadaptés. En outre, les contraintes calculatoires qu'il faudra nécessairement accepter pour une prise en compte à peu près efficace du polymorphisme fonctionnel sont incompatibles avec la volonté de disposer d'un produit commercialement viable.

Exercices

1. Écrivez un virus par écrasement de code V_N, en Bash (voir [38, chapitre 7] pour plus de détails) dont les premières lignes sont

   ```
   #!/bin/bash
   declare -i sig=N
   ....
   ```

   La valeur entière N est incrémentée lors de la duplication virale (le code est alors « polymorphe », certes de manière triviale). En reprenant les notations de la démonstration présentée dans la section 6.2, nous noterons A la génération initiale V_0 et P_i la i-ième génération (mutée) de A. À l'aide des remarques faites en fin de section 6.2 :

   (a) Proposez un codage e_A (respectivement e_{P_i}) pour A (respectivement P_i).

   (b) Proposez une formule logique F telle que e_A ne satisfait pas F mais qui est satisfaite par e_{P_i} et pour un i unique fixé.

   (c) Programmez un détecteur $\mathcal{D}$ utilisant cette formule F pour déterminer que e_{P_i} est une forme mutée de A.

   (d) Écrivez un programme en Bash, non viral, provoquant une fausse alarme relativement à $\mathcal{D}$.

2. Démontrez, en vous aidant d'un exemple générique simple contenant n instructions de saut conditionnel et m instructions de saut inconditionnel, pourquoi le permutateur de la section 6.3.2 produira au bout d'un certain nombre N de générations un code ayant une structure identique à celui d'une génération antérieure. Exprimez N en fonction de n et m.

3. Démontrez que dans l'algorithme présenté dans le tableau 6.2, les nombres générés (Registre1 $\oplus$ Registre2) sont tous compris entre 0 et 2^m, qu'ils sont générés une et une seule fois avant de voir réapparaître la valeur initiale IV (utilisez la notion de groupe cyclique).

4. Soit le code du décrypteur suivant :

```
     LEA ESI,[VIRUSBODY + EBP]
     MOV ECX,ENDVIRUSBODY - VIRUSBODY  ; taille corps viral
     MOV AL,BYTE[KEY + EBP]            ; clef de chiffrement
DECRYPT:
     XOR BYTE[ESI],AL
     INC ESI
     LOOP DECRYPT
VIRUSBODY:
   ............
ENDVIRUSBODY:
```

   Codez en assembleur ce décrypteur en utilisant la technologie d'évasion de saut de code présentée dans la section 6.3.1, avec un niveau de récursivité de 2. Pour la fonction `XOR`, vous pourrez notamment utiliser l'équivalence entre la forme algébrique normale $x \oplus y$, la forme disjonctive normale $(x \wedge \overline{y}) \vee (\overline{x} \wedge y)$ et la forme conjonctive normale $(x \vee y) \wedge (\overline{x} \vee \overline{y})$.

5. Montrez que la grammaire formelle de l'exercice 6.3 génère bien toutes les chaînes contenant au moins un zéro.

6. Expliquez comment peuvent se définir les notions de non détection et de fausse alarme par rapport au problème de la complétude d'une grammaire formelle (voir en particulier le théorème 6.5).

7. Déterminez quels sont les langages acceptés par les automates présentés dans les exemples 6.4 et 6.5.

8. Soit la transformation de code consistant à remplacer l'instruction `OP REG, REG2` en la suite d'instructions suivante :

```
MOV MEM, REG
OP  MEM, REG2
MOV REG, MEM
```

   Déterminez une grammaire minimale décrivant cette réécriture. Généralisez ensuite à l'ensemble des règles de la section 6.3.2. Montrez que la grammaire obtenue possède un système de réécriture ne contenant pas les systèmes de Tzeitzin T_0 et T_1.

9. Montrez que les grammaires décrivant l'action du compresseur d'un virus métamorphe (voir section 6.3.2) sont du type 1 (grammaires contextuelles). Déduisez-en la complexité générale d'un virus métamorphe.

Chapitre 7

Résister à la détection : la furtivité

7.1 Introduction

La furtivité est probablement la fonctionnalité anti-antivirale la plus difficile à définir et à formaliser. À ce jour, la seule définition théorique et les premières tentatives de formalisation connues autour de ce concept sont celles de Z. Zuo et M. Zhou [149] datant de 2004.

Soit Ω un ensemble dénombrable dont tout élément décrit soit un programme soit une donnée. Soit alors la fonction totale $\sigma : \Omega \rightarrow \mathbb{N} \cup \perp$. Cette fonction représente une abstraction des ressources d'un système informatique (programmes exécutables, système d'exploitation, appels systèmes, horloge, mémoire, disques...). Si $\sigma(x) = \perp$ alors la fonction σ n'est pas définie pour cet élément. Ce prolongement permet de considérer une fonction récursive totale.

Définition 7.1 *(Virus furtif [149]) Soit v une fonction récursive totale et $\mathfrak{S}$ une fonction récursive. Alors la paire $(v, \mathfrak{S})$ est un* virus furtif *relativement à la fonction $\mathfrak{S}$ s'il existe une fonction récursive h telle que, pour tout i,*

$$\phi_{v(i)}(\sigma) = \begin{cases} D(\sigma), & si\ T(\sigma) \\ \phi_i(\sigma[v(S(\sigma)^\sigma), h(\mathfrak{S}^\sigma)]), & si\ I(\sigma) \\ \phi_i(\sigma), & sinon \end{cases}$$

et

$$\phi_{h(i)}(x) = \begin{cases} \phi_i(y), & si\ x = v(y) \\ \phi_i(x), & sinon \end{cases}$$

où les prédicats récursifs $T(\sigma)$, $I(\sigma)$ désignent respectivement la condition de déclenchement de charge finale et la condition d'infection. La fonction $S(\sigma)$ est une fonction récursive dite de sélection.

Rappelons (voir section 6.1) que lorsque le prédicat $T(\sigma)$ est vérifié, alors la charge finale est lancée (représentée par la fonction $D(\sigma)$) ; si le prédicat $I(\sigma)$ est vérifié, alors le virus choisit un programme à l'aide de la fonction de sélection $S(\sigma)$, l'infecte d'abord et ensuite exécute le programme original i (transfert de contrôle à la partie hôte). Mais dans le cas d'un virus furtif, non seulement il infecte d'autres programmes, mais il modifie ou utilise également certains appels système de telle sorte que lorsque une vérification est faite par le système ou l'utilisateur (via le système néamnoins) pour contrôler l'intégrité des programmes, ces derniers paraissent sains alors qu'en réalité ils ont été infectés.

Concernant la détection des virus furtifs, là encore un seul résultat est connu, établi par Z. Zuo et M. Zhou [149]. Notons D_f l'ensemble des virus furtifs et $D_f^{\text{fixé}}$, l'ensemble des virus furtifs ayant un même noyau donné (voir section 6.1 pour la définition du noyau d'un virus et les autres notations).

Théorème 7.1 *[149] L'ensemble $D_f^{fixé}$ est Π_2-complet et l'ensemble D_i est Σ_3-complet.*

Ce résultat montre que la détection des virus furtifs est un problème dont la complexité dépasse de loin celle de beaucoup d'autres virus classiques. L'utilisation de la furtivité à des fins de lutte anti-antivirale se révèle une stratégie payante pour le programmeur de virus, si cette fonctionnalité respecte un certain nombre de principes et est correctement implémentée.

Les techniques de furtivité ont connu une évolution récente assez médiatisée avec la notion de *Rootkits*. Cependant, conceptuellement, cette technologie ne constitue pas une nouveauté en soi, mais juste une généralisation à des systèmes plus récents de techniques anciennes connues sous le terme de techniques de furtivité. Toutefois, l'intérêt de la technologie *Rootkit* a été de frapper les esprits et de relancer la réflexion dans le domaine de la dissimulation et le camouflage au sens large du terme. L'affaire du *Rootkit* de la firme Sony [114], les déclarations d'autant plus alarmistes qu'elles sont inhabituelles de la société Microsoft[1], l'utilisation constatée de la technologie dans des vers comme *W32/Bagle.GE* ou des virus comme *Gurong.A*[2] ont largement contribué au regain d'intérêt pour les technologies de furtivité de code.

À ce jour, malheureusement, il reste encore beaucoup de problèmes ouverts. Le principal concerne l'étude théorique de la notion même de furtivité. Nous tenterons d'esquisser, dans ce chapitre, une telle étude en établissant un parallèle avec la stéganographie.

[1] Lors de sa conférence *(Windows et sa sécurité)* au salon InfoSec d'Orlando, en avril 2006, Mike Danseglio, responsable sécurité chez Microsoft, a reconnu que face à la menace des *rootkits*, la seule réponse aujourd'hui envisageable est d'adopter des procédures automatisées de désinstallation et de résinstallation des systèmes infectés : *« Quand vous êtes infectés par des* rootkits *ou des* spywares *très évolués, la seule solution est de repartir de zéro. Dans certains cas, il n'existe aucun autre moyen pour retrouver un système stable que de tout effacer et de tout résintaller ! »*

[2] Ce dernier se propage via les répertoires partagés dans KaZaa et via le courrier électronique.

7.2 La furtivité « classique »

La plupart des techniques de furtivité connues ont été imaginées pratiquement en même temps que les premiers virus eux-mêmes. Depuis, le passage des plates-formes 16 bits aux plates-formes 32 ou 64 bits (systèmes d'exploitation actuels) n'a pas vu l'émergence de techniques vraiment novatrices en matière de furtivité. Les programmeurs se sont contentés de les faire évoluer et de les recycler pour prendre en compte toutes les fonctionnalités offertes par les systèmes d'exploitation actuels. Même ce que certains considèrent comme la révolution « *rootkit* » n'est en fait qu'une généralisation de principes déjà anciens.

Les techniques utilisées à des fins de furtivité sont trop nombreuses pour être toutes décrites ici. En outre, le lecteur aura tout intérêt à consulter les ouvrages existants qui les présentent en détail, en particulier [88, chapitre 21–23]. Nous allons ici présenter en détail le virus *Stealth* qui résume un grand nombre d'entre elles, mais qui surtout constitue un excellent exemple de l'esprit même de la furtivité.

7.2.1 Le virus *Stealth*

Le virus *Stealth* (« furtivement » en anglais) a été créé par Mark Ludwig en 1991 [87]. Fonctionnant sous DOS et d'anciennes versions de Windows, il reprend certains mécanismes du célèbre virus pakistanais *Brain*. Virus de démarrage très évolué, (voir [38, chapitre 4]), il est un condensé de tout ce qu'un tel virus peut déployer de ruse pour leurrer les tentatives de détection. Si ce virus sous sa forme publiée est désormais détecté par les antivirus, il n'est pas absurde de penser que sous une forme modifiée, adaptée et améliorée, il parvienne à infecter de nombreux ordinateurs. Le virus *Stealth* reste en quelque sorte un virus actuel. Le projet récent *NTBoot* en reprend d'ailleurs bien des approches.

L'intérêt principal d'un virus de *boot* réside dans le fait qu'il intervient avant le lancement du système d'exploitation (et donc de tout logiciel, en premier lieu l'antivirus). Il est donc très difficile de stopper son lancement au niveau du système par le biais d'un quelconque antivirus. Ce dernier doit intervenir avant, c'est-à-dire directement au niveau du BIOS.

Agissant très tôt, un virus de *boot* peut éventuellement mettre en place un certain nombre de mécanismes lui permettant d'augmenter son efficacité et en particulier de limiter ou d'interdire sa détection. C'est la raison principale pour laquelle les virus de *boot*, dans leur forme moderne (par exemple *NTBoot*) représentent une menace toujours actuelle, en particulier avec les technologies de type *rootkit*. Il faut malheureusement compter sur l'ingéniosité des programmeurs pour développer une capacité de furtivité toujours plus grande. Il faut surtout insister sur le fait qu'un virus de *boot* peut concerner n'importe quel système d'exploitation et pas seulement les systèmes Windows. Cela offre un champ de possibilités intéressant, notamment sous Linux ou autres Unix libres, qui sont désormais très répandus.

Le processus de démarrage

Afin de bien comprendre comment agit un virus de démarrage comme *Stealth*, il faut d'abord savoir ce qui se passe lors de la séquence de démarrage.

La séquence de démarrage À la mise sous tension, le processeur entre en mode réinitialisation, remet à zéro tous les emplacements mémoire, effectue un contrôle de parité de cette mémoire et initialise le registre `CS` *(Code Segment)*[3] avec l'adresse `FFFFH` et le registre `IP` *(Instruction Pointer)*[4] à zéro. Le code du BIOS (Basic Input/Output System), localisé à l'adresse `FFFFH:0000H`, est alors lancé.

Le BIOS effectue alors un certain nombre d'opérations parmi lesquelles (voir [38, chapitre 11]) :

- parcours des principaux ports pour détecter et initialiser les périphériques présents (appel notamment aux interruptions `11H` (reconnaissance matérielle) et `12H` (reconnaissance de la mémoire)). Il vérifie également que les périphériques indispensables pour la suite fonctionnent correctement ;
- le chargement de la table des vecteurs d'interruptions en mémoire basse (256 adresses des routines d'interruptions). Cette table est utilisée par le BIOS et le système d'exploitation pour la gestion et l'appel des interruptions (quand c'est encore le cas). La *BIOS Data Area* est également chargée. Localisée à l'adresse `0040H:0000H`, elle permet la gestion des périphériques présents (par exemple à l'adresse `0040H:0013H` se trouve mémorisée la quantité de mémoire physique effectivement disponible).

Ensuite, le BIOS détermine s'il existe un périphérique amorçable (disquette, disque dur, CDROM, ZIP...), c'est-à-dire contenant un programme de démarrage de 512 octets en tête 0, piste 0 et secteur 1, contenant la signature d'un secteur de démarrage valide[5]. Dans le cas de machines *multiboot*, ce secteur est dit principal. Il est alors chargé en mémoire par le BIOS à l'adresse `0000H:7C00H` qui lui transfère finalement le contrôle.

Le secteur de démarrage principal lance ensuite les programmes spécifiques au système d'exploitation. Dans le cas du *multiboot*, il y a d'abord lecture de la table de partition présente dans le secteur maître puis lancement du programme de démarrage localisé dans un secteur de démarrage secondaire de la partition correspondant au système d'exploitation sélectionné.

Au final, il est essentiel de constater que, jusqu'à cette dernière étape, tout le processus est indépendant du système d'exploitation. C'est ce fait qui rend l'action des virus de démarrage universelle et potentiellement très dangereuse.

[3] Ce registre de 16 bits agit comme un sélecteur d'accès aux adresses 32 bits des segments mémoire. Il gère les segments de code d'un programme.

[4] Ce registre contient les 16 bits de poids forts du registre 32 bits `EIP` *(Extended Instruction Pointer)* et indique l'adresse relative au début du segment de code considéré *(offset)* de la prochaine instruction à exécuter. L'adresse complète d'une instruction est alors donnée par `CS :IP`.

[5] Cette signature est localisée dans les deux derniers octets (offset `1FEH`) et vaut `AA55H`.

Leur action consiste donc à infecter le programme de démarrage contenu dans le secteur principal. Le plus souvent, cela se fait (pour les systèmes vulnérables) par l'intermédiaire d'un périphérique *bootable* infecté, laissé connecté à l'ordinateur au moment du démarrage.

Notons que pour les systèmes d'exploitation modernes, qui n'utilisent plus l'interruption matérielle `13H` une fois le système d'exploitation lancé, les virus de démarrage classique ne fonctionnent plus. Si un système peut encore être infecté par de tels virus, il ne peut, en revanche, plus propager cette infection à un autre périphérique amorçable.

L'infection proprement dite est assez délicate car la contrainte expresse est que le programme de démarrage doit conserver après infection une taille maximale de 512 octets. Nous allons voir comment un virus de *boot* comme *Stealth* s'affranchit aisément de cela.

Le secteur de démarrage Afin de bien comprendre les mécanismes du virus, il convient de bien comprendre quelles données sont à sa disposition, en premier lieu celles contenues dans le secteur de démarrage. Elles sont les suivantes :

- des données concernant l'unité physique sur laquelle se trouve ce secteur (disquette, disque...). Elles concernent essentiellement l'organisation et la structure de l'unité (nombre de secteurs, de têtes de lecture, de FAT – *File Allocation Table*, structure de données permettant de déterminer si les secteurs sont affectés à un fichier ou non ou s'ils sont défectueux –, nature du support, son type, ...). Le BIOS et le DOS s'en servent pour piloter l'unité ;
- la table des paramètres du disque. Elle permet au BIOS de pouvoir physiquement piloter l'unité et en contient les caractéristiques techniques. Une table par défaut est d'abord chargée par le BIOS puis le secteur de démarrage la remplace par la table spécifique à l'unité.

Lors de son exécution, le secteur de démarrage recherche les fichiers systèmes spécifiques à lancer pour charger le système d'exploitation. Il lui faut pour cela déterminer où commence le répertoire racine de ce système. Les données dont il dispose en son sein lui permettent un tel calcul (valeur `FDRS` (*First Root Directory Sector*)) :

```
FDRS  = FAT_COUNT x SEC_PER_FAT + HIDDEN_SECS + FAT_START
```

Puis, à partir de ce répertoire, il doit charger le premier fichier système à lancer. Son adresse est indiquée par la valeur `FDS` *(First Data Sector)* donnée par :

```
FDS = FDRS + [(32 x ROOT_ENTRIES) + SEC_SIZE  - 1 ]/SEC_SIZE
```

Le nombre d'octets à charger est finalement trouvé dans la table de répertoire contenant toutes les données de fichiers présents.

L'infection par le virus *Stealth*

Le virus comprend trois parties :
- la première est un secteur de démarrage viral (SDV) remplaçant le secteur original lors de l'infection ;
- la deuxième est le corps principal du virus (CPV). Le virus présentant de nombreuses fonctionnalités, la limite des 512 octets est trop contraignante et le virus utilise pour cette partie six secteurs qu'il doit dissimuler. La fonction du SDV sera de charger ces six secteurs afin de restaurer le virus dans son intégralité ;
- la troisième consiste en une copie du secteur de démarrage avant infection (SDO).

L'action du virus se décrit alors simplement selon les trois phases suivantes :

1. le secteur viral SDV est chargé lors du démarrage de la machine ;
2. il charge en mémoire le corps principal du virus (CPV) de maniére résidente. Le virus est alors pleinement actif et est susceptible d'infecter d'autres secteurs de démarrage ;
3. le contrôle est alors redonné au secteur de démarrage originel (SDO) pour que la machine démarre normalement. Cela se fait par simple lancement du secteur SDO. Cette derniére étape est particulièrement astucieuse car elle permet au virus d'être totalement indépendant du système d'exploitation.

Nous ne présenterons pas la routine de recherche. Le lecteur pourra consulter [37]. Elle agit de manière classique en détournant l'interruption 13H. La routine de copie, en revanche, est moins classique car elle doit se faire de sorte à être compatible avec les fonctionnalités de furtivité souhaitées. Cette routine de copie doit donc faire en sorte :
- qu'il n'y ait aucune interférence avec le système susceptible de trahir la présence du virus ;
- que le virus soit le plus indétectable possible (furtivité) d'un point de vue statique ;
- que le virus soit le plus portable possible (l'infection doit réussir quel que soit le support cible, en particulier toute sorte de disquette et de périphérique). Notons que cette portabilité est un élément essentiel dans tout mécanisme de furtivité. La détection peut en effet survenir en cas d'échec de l'infection en raison de certains périphériques non envisagés et donc non pris en compte. Le disfonctionnement ou l'effet de bord qui résultera trahira la présence du virus.

En fait, tout est dicté par la contrainte de taille sur le secteur de démarrage : 512 octets. Or *Stealth* occupe sept secteurs dont six doivent être cachés quelle que soit la cible. La procédure de copie se décompose alors de la manière suivante :

1. le virus mémorise le secteur de démarrage original (SDO) ;
2. si l'unité à infecter est une disquette, le virus déclare certains *clusters* inoccupés et sains comme défectueux. Il y installe la plus grosse partie de

son code (CPV et SDO). Pour cela, il suffit d'écrire la valeur FF7H dans chaque entrée concernée par ces *clusters* dans la FAT[6]. Les *clusters* choisis dépendent du type d'unité à infecter. Par exemple, pour une disquette de 1,4 Mo, *Stealth* utilise les *clusters* 13 à 17 (pour le corps principal du virus) et 18 pour le SDO, de la piste 79, tête 0. Le choix des pistes est judicieux car il correspond à des *clusters* utilisés en dernier. Cette approche est subtile car les secteurs défectueux sont ignorés par le système d'exploitation et donc par les logiciels lancés par lui (en premier lieu les antivirus). Le virus modifie également la FAT2 (copie de la FAT pour restauration en cas de problème) ;

3. si l'unité est un disque dur, le virus installe CPV et SDO dans la piste 0, tête 0, à partir du secteur 2. Cette piste est indépendante du système d'exploitation et permet d'y cacher avantageusement le virus. Le plus petit des disques durs contient au minimum une bonne dizaine de secteurs ;
4. le virus installe le secteur de démarrage infecté (SDV) en tête 0, piste 0 et secteur 1 après l'avoir modifié pour tenir compte des paramètres de l'unité en cours d'infection (informations sur la table de partition, les paramètres techniques...).

Les mécanismes d'antidétection

Ce qui fait de *Stealth* un virus particulièrement élégant sont ses mécanismes de furtivité. Certes, il existe désormais des techniques beaucoup plus évoluées pour contrer les techniques de détection actuelles, mais elles reprennent en général, pour la plupart, la philosophie de celles mise en œuvre par *Stealth*. De ce point de vue, les auteurs de codes malveillants ne font que revisiter des concepts imaginés par Mark Ludwig, l'auteur de *Stealth*.

La première des astuces est la lutte contre un éventuel examen du secteur de démarrage par un antivirus. Ce secteur contient la seule partie du virus qui n'est pas cachée dans des *clusters* défectueux, ce afin de pouvoir accéder aux parties CPV et SDO et les charger.

- Dans le cas d'une disquette, tout ordre de lecture du secteur en question, via l'interruption 13H est intercepté par le virus (qui est alors résident ; voir plus loin). Si la demande de lecture concerne un secteur autre que le secteur de démarrage, le virus redonne le contrôle à l'interruption 13H originale. Dans le cas contraire, le virus redirige la lecture vers le secteur contenant la copie saine du secteur de démarrage (SDO). L'antivirus conclut à la non infection.
- Dans le cas d'un disque dur, le virus interdit toute lecture ou toute écriture dans les secteurs 2 à 7 de la piste 0, tout en faisant croire que l'ordre demandé s'est effectué sans problème.

[6] La FAT attribue une valeur à chaque *cluster* décrivant son état d'occupation : 0 si le *cluster* est libre, un pointeur sur le *cluster* suivant dans le cas d'un fichier (structure de liste chaînée), FF8H <-> FFFH si le *cluster* correspond à la fin du fichier et FF7H si le *cluster* est défectueux.

Dans ces conditions, à l'époque, seul un examen extérieur au système d'exploitation (démarrage sur une disquette saine ou détection par un antivirus de BIOS) permettait de détecter le virus (SDV).

Le virus, étant résident, et après avoir passé le contrôle au secteur de démarrage original, doit veiller à ne pas être détecté par un examen de la mémoire. Pour cela, le virus va soustraire de la mémoire au système d'exploitation. Lors du démarrage, le BIOS stocke la quantité de mémoire physique disponible à l'adresse `0040H:0013H`, en kilo-octets. Cela permet au système de savoir de quelle quantité de mémoire il peut disposer (dans la limite des 640 Ko). Le virus *Stealth* prenant le pas avant le système, il « dérobe » de la mémoire en soustrayant la quantité qui lui est nécessaire (soit quatre kilo-octets) et s'installe en partie haute de mémoire. Le système n'y accédera pas car, pour lui, elle n'existe pas. Le virus agira, lui, via le déroutement de l'interruption 13H. En résumé :

1. la partie SBV se charge en mémoire haute à l'adresse `9820H:7C00H` puis il lit les 6 autres secteurs (CPV et SDO) et les place en mémoire juste après lui (de `9820H:7000H` à `9820H:7BFFH`) ;
2. le SBV soustrait alors 4 kilo-octets à la valeur stockée en `0040H:0013H` ;
3. le SBV détourne l'interruption `13H` vers le virus ;
4. enfin il déplace SBO de `9820H:7A00H` vers `0000H:7C00H` et l'exécute. Une séquence de démarrage normale s'effectue ensuite.

Cette dernière fonctionnalité de furtivité est particulièrement astucieuse, la partie de la mémoire utilisée n'existant plus pour le système. La détection du virus résident ne peut plus se faire directement.

7.2.2 Les techniques de furtivité « modernes »

Les techniques de furtivité mises en œuvre dans le virus *Stealth* résument dans l'esprit les différentes techniques et approches que l'on peut considérer en pratique. Deux aspects majeurs sont à la base de toute technique de furtivité :

- agir le plus bas possible dans le système. Il est essentiel de pouvoir prendre le pas sur les couches basiques du système pour prendre le contrôle de toutes les ressources disponibles ;
- agir le plut tôt possible, afin de couper couper l'herbe sous le pied au système. De fait, la séquence de démarrage (et les organes qui y sont impliqués) est une structure critique qui intervient assez souvent. Mais ce n'est pas là la seule.

Depuis *Stealth* et le début des années 90, les auteurs de codes malveillants se sont contentés de reprendre les techniques existantes, de les transposer aux nouvelles plates-formes logicielles et matérielles (quelquefois non sans génie). Aucune évolution notable n'a été véritablement constatée dans le domaine de la furtivité, et ce jusqu'en 2006 (voir la section 7.3).

Pour bien comprendre quels sont les différents niveaux auxquels il est possible d'agir pour dissimuler des données ou des processus, il est nécessaire de

bien comprendre le mode de fonctionnement du système Windows. Chaque application, chaque processus passe par l'une des couches de ce système pour obtenir des services ou des informations du système. Chaque fois qu'un appel est fait à l'une de ces couches, le code malveillant peut intercepter cet appel et en manipuler soit la requête elle-même, soit les données qui lui sont retournées (liste des processus par exemple dans le cas du *Task Manager*). La structure du système Windows et la hiérarchie des couches le composant sont données en figure 7.1. Les différents niveaux d'action sont alors les suivants [115]. Chaque

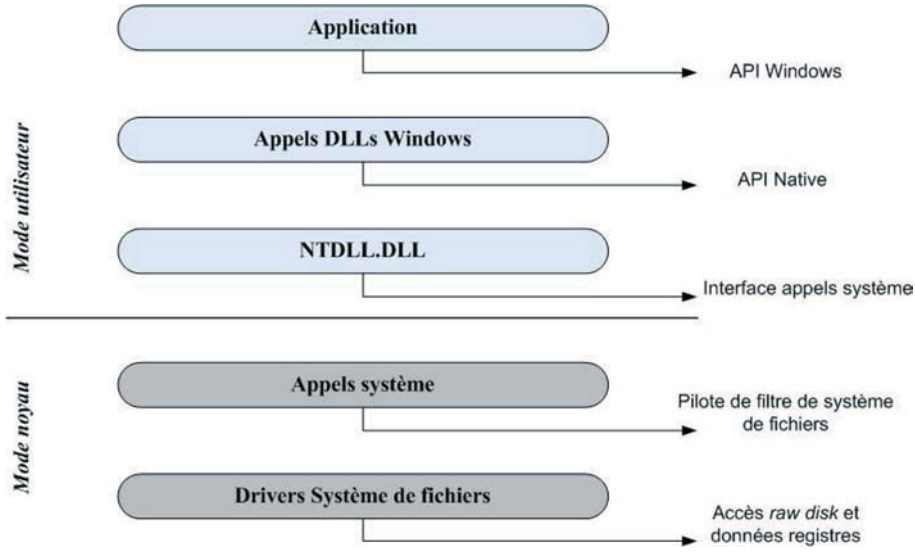


Figure 7.1 – Structure du système Windows [115]

API sollicitée est gérée au niveau d'une couche bien spécifique.

- La couche la plus haute est celle généralement désignée par le terme d'API Windows utilisateur. Elle permet de gérer les actions de base des applications en mode utilisateur. L'application importe alors statiquement ou dynamiquement l'API à partir du fichier `\windows\C\winnt\system32-\kernel32.dll`. À titre d'exemple, pour lister les fichiers présents dans un répertoire, les fonctions `FindFirstFile` et `Find NextFile` sont utilisées. Des techniques de furtivité peuvent intercepter un appel à ces fonctions et en manipuler les données retournées, comme par exemple faire disparaître le nom des fichiers de nature virale. De fait, on peut considérer que les appels à l'API Windows correspondent aux anciennes interruptions DOS (`INT 21H`).
- La seconde couche (Dll Windows) est sollicitée, en mode dynamique, pour des services et fonctionnalités plus spécifiques. Dans ce cas, il n'y a chargement dans un processus que lorsqu'un exécutable du processus requiert

les API exportées par le ou les fichiers Dll concernés. En mode statique, il y a passage par la table d'importation (section dans l'image de l'exécutable). La dissimulation peut alors agir à ce niveau en substituant, dans la table d'importation, des références aux fonctions de furtivité aux références aux fonctions Dll système légitimes (voir [63, pp. 73-76 et 105-106]).

- Le fichier `\windows\C\winnt\system32\ntdll.dll` est lui sollicité essentiellement pour la gestion des processus, des mémoires et des fichiers en mode utilisateur, lorsque cela implique le noyau. C'est à ce niveau que des techniques furtives agiront pour dissimuler un ou plusieurs processus (manipulation de l'API `NtQuerySystemInformation`; voir [63, pp. 87-91]).
- La couche des *appels système* (mode noyau). La furtivité à ce niveau, utilise le principe des *system-call hooking* (interception des appels système). La technique est similaire à celles consistant à manipuler la table d'importation pour les Dlls. Dans le cas présent, c'est la table des fonctions contenue dans chaque pilote de périphérique. Lorsqu'un pilote est installé, il initialise une table de pointeurs de fonctions gérant les différent types d'IRPs (*I/O Request Packets*) [63, pp. 96-106 et chapitre 6].
- La couche des objets du noyau, gérant les structures de *drivers* de périphériques, les clefs de la base de registre, les fichiers, répertoires ...*(raw access mode)*. Les techniques de furtivité attaquent, à ce niveau, tant directement les objet eux-mêmes que les appels à ces objets (couche immédiatement supérieure) [63, chapitre 6 et 7].

Cette hiérarchie est en tout point comparable à celle qui existait entre les interruptions liées au système d'exploitation (par exemple avec la célèbre interruption `21H`) et celle liées au matériel (interruptions BIOS). Les systèmes actuels n'ont fait que diminuer la granularité de ces couches en augmentant leur nombre. Mais globalement la philosophie reste identique. Il en est de même pour les codes malveillants.

Les techniques de furtivité les plus sophistiquées sont donc celles qui agissent directement en mode noyau : manipulation directe des objets et structure du noyau, interception des API natives en mode noyau au moment de leur passage vers la couche utilisateur (technique de *system call hooking*)... Ainsi, l'action permet d'accéder à des ressources et services de bas niveau autorisant ainsi une manipulation en profondeur des données, ressources et actions du système. En outre, cela diminue les possibilités d'analyse et d'action en vue de la détection de ces techniques.

Que ce soit sous Windows ou sous Linux, l'approche des techniques de furtivité est toujours la même : agir dans les couches les plus inférieures et donc le plus tôt possible dans la chaîne d'événements impliqués dans un processus. Mais fondamentalement, les techniques d'API *hooking*, d'infection de Dlls, d'infection de *Thread*, de détournement de services du noyau, de manipulation de données du noyau ne sont fondamentalement pas différentes des détournements ou déroutements d'interruptions et autres actions mises en œuvre par

les codes malveillants comme *Stealth*. Les seules différences sont essentiellement formelles et relèvent d'une richesse structurelle et fonctionnelle plus importante des plates-formes qui ont suivi.

Nous ne présenterons pas les techniques de furtivité dites « modernes ». Cela nécessiterait trop de place et il existe déjà d'excellents ouvrages très complets traitant du sujet. Nous recommandons au lecteur intéressé par les aspects les plus techniques de la furtivité de se référer à l'ouvrage de référence dans le domaine [63] et au site web qui y est lié[7].

7.3 La technologie des *rootkits*

Le terme de *rootkit*, apparu il y a environ quatre ans, ne fait que désigner sous un terme plus à la mode un ensemble de techniques de furtivité, avec la caractéristique notable que l'action se situe, de manière privilégiée, au niveau du noyau du système. Cette appellation est abusive et n'est pas vraiment justifiable par les évolutions de la furtivité. Ce n'est qu'en 2006, que le terme de *rootkit* a pu prendre tout son sens, dans la mesure où des techniques véritablement nouvelles ont récemment fait leur apparition et par conséquent méritent une appellation spécifique.

7.3.1 Principes généraux

Les techniques de furtivité présentées dans la section précédente présentent toutes le défaut de modifier l'intégrité de données systèmes (en particulier certaines structure de données du noyau), de manière plus ou moins importante [116]. L'intégrité de ces données ne peut être modifiée ou manipulée qu'en mémoire [126].

Un autre aspect important concerne le niveau d'action de la furtivité. Plus cette dernière agira à un niveau bas, et ce le plus tôt possible, moins il existera de possibilités de se situer, pour un détecteur, à un niveau encore plus bas et/ou encore plus tôt dans la chaîne d'exécution. Les premières techniques de furtivité agissaient au niveau simplement utilisateur (par exemple manipuler l'affichage des données d'une commande comme `ps` sous Unix et supprimer l'existence des processus viraux [38, chapitre 8]) ce qui les rendaient aisément détectables par des outils situés au niveau de la couche noyau. En réaction, sont apparus les *rootkits* en mode noyau (comme, par exemple, le *rootkit* FU). Mais là encore, la technologie classique des *rootkits* (qui en fait ne font que généraliser les techniques de furtivité) laisse des traces et des opportunités de détection permettant de les détecter.

La réponse est venue en 2006 avec deux *rootkits*, *Subvirt* puis *BluePill*, qui ont illustré comment faire pour prendre définitivement le contrôle d'un système, sans laisser de traces pour le second, et sans possibilité de lutte autre que...

[7] Les codes sources de nombreux *rootkits* sont disponibles sur ce site : `http://www.rootkit.com`.

d'agir en amont pour éviter que des codes mettant en œuvre ce type de *rootkit* pour se cacher ne s'introduisent sur une machine. Les techniques utilisées par ces deux *rootkits*, lorsque bien implémentées et maîtrisées[8], rendent inopérantes toutes les techniques de détection utilisées par le système d'exploitation, une fois ce dernier démarré.

Le principe général de fonctionnement et d'action de ces nouveaux *rootkits* est assez simple. Une machine virtuelle de contrôle (ou moniteur virtuel[9]) est installée préalablement au système d'exploitation cible. Ce moniteur virtuel va héberger les fonctionnalités malveillantes et totalement contrôler et/ou manipuler les actions, requêtes et interactions du système, notamment en direction du code malveillant. De fait, il est alors possible pour ce dernier de se dissimuler vis-à-vis des applications de sécurité (antivirus, IDS...) et du système d'exploitation lui-même (contrôle d'intégrité, gestion des processus...).

Le lecteur pourra objecter que l'installation de tels *rootkits* n'est pas chose aisée et représente un risque d'autant plus marginal qu'il est hypothétique. Ce serait une grave erreur de le penser. Cette erreur, pourtant répandue, tient au fait que beaucoup de professionnels de la sécurité informatique ont une perception fausse de ce que sont les *rootkits*. Ces derniers ne sont pas des codes malveillants mais des boîtes à outils, des « librairies » de fonctions de furtivité utilisées, à des degrés divers, essentiellement par des codes malveillants, mais aussi, dans quelques cas, encore marginaux, par des applications « légitimes » (*rootkit* de la firme Sony, par exemple [114]). Si un code parvient à s'introduire dans une machine cible (utilisation de vulnérabilités par exemple), il lui sera alors possible d'installer ce type de « *package* de furtivité ». Rappelons que lors de la phase initiale d'infection, ces codes ne seront pas détectés par les antivirus !

Une autre manière d'agir et de régler le problème de contrôle en faveur de l'attaquant est d'utiliser des codes malveillants situés directement au niveau du BIOS, codes dont la faisabilité a été démontrée dans [38, chapitre 11]. Là, il est alors possible d'installer ce type de technologie *rootkit* – ou du moins quelques prémices suffisantes – dont nous allons présenter les deux seuls représentants connus à ce jour.

7.3.2 Le *rootkit Subvirt*

Ce *rootkit* a été conçu en mars 2006 par des chercheurs de l'université du Michigan et du département recherche de la société Microsoft [75] et présenté en mai 2006 à Oakland, États-Unis. Leur objectif a été de prouver qu'il est possible de pallier les principaux inconvénients et limitations des technologies de

[8] En veillant notamment à ce que les ressources en CPU, en mémoire et en espace disque, ainsi qu'en bande passante réseau, ne soient pas grevées de manière perceptibles. Mais là encore, cela peut être obtenu par la réalisation d'un compromis judicieux entre l'utilisation des ressources de la machine d'une part, le temps de mise en œuvre et le nombre de fonctionnalités du code malveillant, d'autre part.

[9] Le terme de *superviseur* ou d'*hyperviseur* est également utilisé. Nous utiliserons indifféremment les trois.

furtivité ou des *rootkits* existants et ainsi de rendre quasi-inopérantes toutes les techniques de détection. Le résultat est dramatique. Et même les quelques pistes données par les auteurs de cette étude, dans le but de tenter de contrôler et de détecter ce nouveau type de *rootkit*, semblent illusoires si certaines précautions sont prises dans le développement de ce dernier.

Le *rootkit SubVirt* a été implémenté et testé sur des plates-formes *Linux/VMware* et *Windows/VirtualPC* prouvant ainsi l'universalité de ce type de rootkits. Différents « services » offensifs ont été implémentés : espion de clavier, serveur Web de *phishing*, recherche de données sensibles dans le système de fichiers. En outre, *SubVirt* déploie un système de contre-mesures destinées à défaire les techniques connues de détection de moniteur virtuel.

Principe général des machines virtuelles

Une machine virtuelle est en fait un environnement d'exécution complet et simulé sur un ordinateur. Plusieurs de ces environnements peuvent coexister sur une même machine, chacun réalisant une émulation de l'ordinateur hôte. En fait chacun de ces environnements simule de manière totale un véritable ordinateur, de manière entièrement indépendante des autres environnements. La partie logicielle du système d'exploitation fournissant et contrôlant ces machines virtuelles est dénommée *superviseur* ou *hyperviseur* (ou encore *moniteur virtuel*).

Ce moniteur virtuel gère les ressources matérielles fournit une ou plusieurs de ces abstractions de systèmes d'exploitation (les machines virtuelles) [59]. Le principe en est illustré en figure 7.2. Les principes de fonctionnement sont les suivants :

- le système d'exploitation virtuel et les applications (hôtes et virtuelles) relèvent du mode utilisateur *(user mode)*. Seul le moniteur viruel est en mode noyau *(kernel mode)*. Une machine virtuelle est composée du système d'exploitation virtuel et des applications que ce dernier exécute (applications virtuelles) ;
- le moniteur virtuel émule le matériel à destination des applications hôtes. Il n'y a, de fait, aucune différence entre une gestion émulée de ce matériel et sa gestion réelle (directe). La seule différente tient au fait que les interactions matériel/applications sont totalement et systématiquement supervisées par le moniteur virtuel ;
- plusieurs systèmes d'exploitation différents peuvent être ainsi exécutés, chacun avec leurs applications respectives, et ce de manière tout à fait transparente l'un vis-à-vis des autres. Le moniteur virtuel, par exemple, répartit et alloue l'espace mémoire physique à chaque machine virtuelle en pages mémoire distinctes. De manière similaire, l'espace disque physique partagé est divisé en deux ou plusieurs disques virtuels, lesquels n'ont aucun secteur physique en commun ;
- le moniteur virtuel fournit également plusieurs services au système d'exploitation en place : services de débogage et d'analyse, configuration

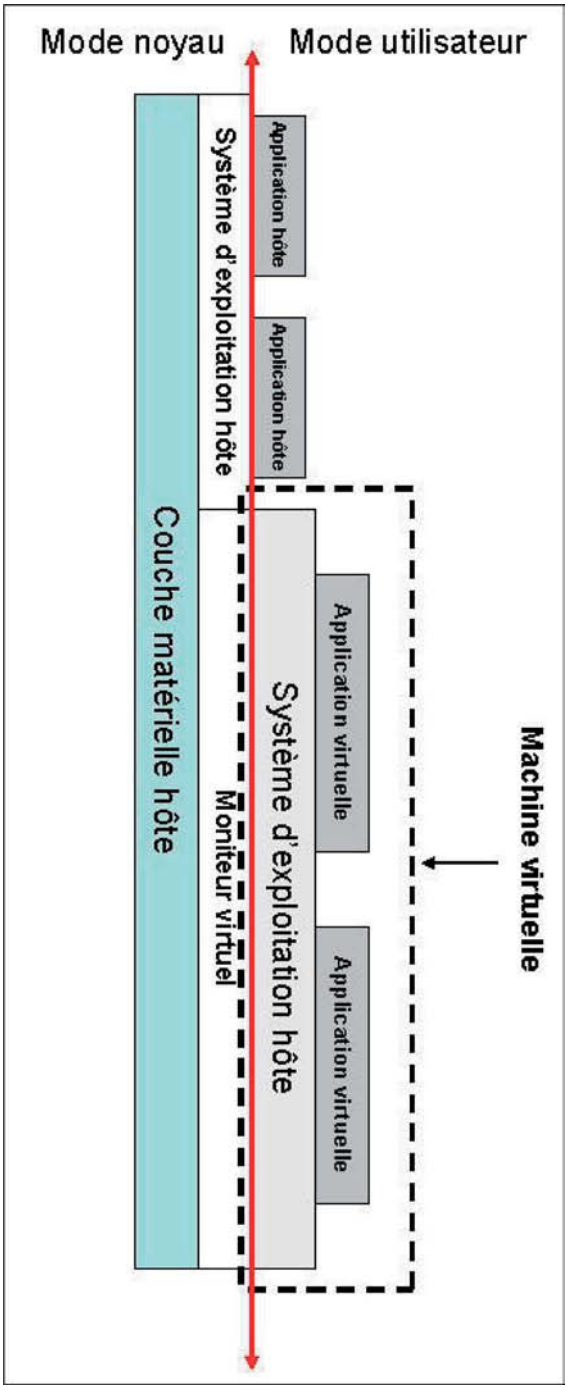


Figure 7.2 – Structure d'un moniteur virtuel (type VMware ou VirtualPC) [75].

du système, détection et prévention d'intrusions, fonctions d'intégrité de code...

Machine virtuelle et *rootkit*

La philosophie des *rootkits* à base de machines virtuelles est excessivement simple. Elle n'est pas non plus techniquement très complexe. À l'installation, ce type de *rootkit* fait migrer le système d'exploitation cible dans une machine virtuelle et ensuite exécute le code malveillant ou les fonctions offensives qu'il soutient au sein du moniteur virtuel ou dans une autre machine virtuelle.

Ainsi, pour le système cible, tout se passe comme s'il était seul. Que ce soit au niveau de son espace mémoire, de son espace disque ou en termes de processus, aucune différence pour le système d'exploitation visé n'est perceptible si le degré de virtualisation est élevé. Les actions et états propres au code malveillant sont complètement isolés de la machine virtuelle hébergeant le système cible, ce qui implique que ce dernier ou ses applications ne peuvent détecter le code malveillant et donc agir contre lui.

En outre, le moniteur virtuel supervise toutes les actions du système d'exploitation cible, en contrôle tous les états (espace mémoire, espace disque, activités des périphériques, activité réseau...). Il est par conséquent en mesure de les analyser et de les modifier de manière totalement transparente pour le système cible, lequel demeure dans l'incapacité la plus totale de détecter ce contrôle et ces manipulations[10].

Le principe général de *SubVirt* est schématisé en figure 7.3. Dans *SubVirt*, le moniteur virtuel doit s'installer sous le système d'exploitation cible et exécuter ce dernier dans une machine virtuelle. Pour réaliser cela, il est donc nécessaire pour le *rootkit* de prendre la main avant le système d'exploitation cible. La seule possibilité est de le faire pendant la séquence de démarrage, qui doit par conséquent lancer prioritairement le moniteur virtuel du *rootkit*. Cette contrainte impose d'accéder au système avec des privilèges suffisants. Plusieurs cas sont possibles :

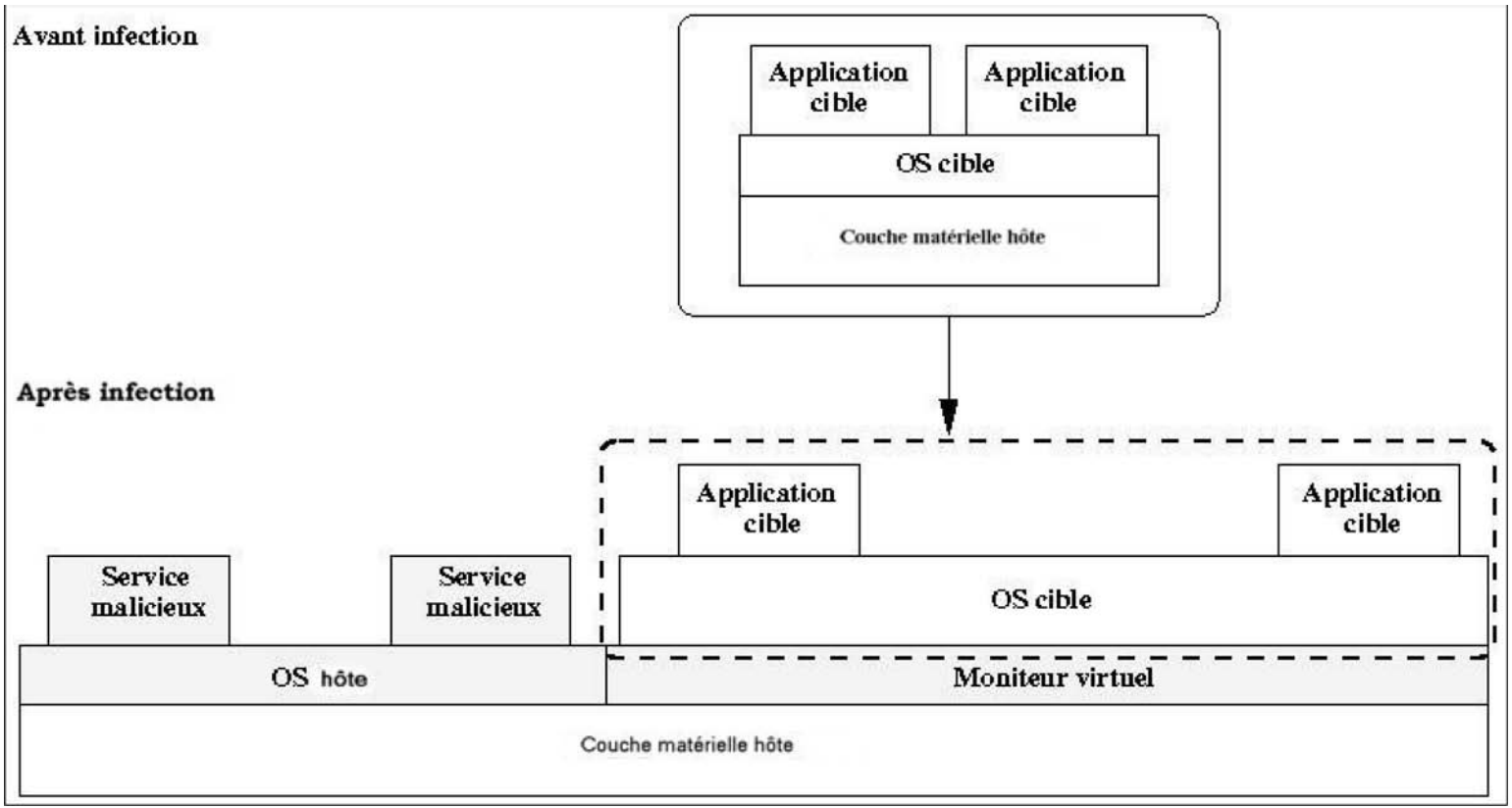
- utilisation d'un BIOS malicieux (voir [38, chapitre 11]) flashé à la place d'un BIOS légitime ;
- utilisation de vulnérabilités logicielles (type 0-Day par exemple) ;
- utilisation de périphériques bootables (CDROM, clef USBA) ;
- attaque par codes malveillants en mode noyau (même les administrateurs ne respectent pas les règles de sécurité informatique les plus élémentaires) ;
- le *rootkit* peut avoir été installé volontairement par le constructeur...

Les possibilités sont bien plus nombreuses qu'on ne peut l'imaginer.

La seconde étape consiste à installer le *rootkit* lui-même. Afin qu'il puisse être lancé à chaque démarrage (mode persistant), le code doit être stocké et

[10] Le lecteur pourra juger du caractère novateur pour ne pas dire visionnaire du virus *Stealth* qui agissait de la sorte, notamment en ce qui concerne les mécanisme d'anti-détection (voir section 7.2.1).

Figure 7.3 – Principe général du *rootkit SubVirt* [75].



dissimulé quelque part sur une unité physique[11]. Reprenant les principes fondamentaux imaginés par Mark Ludwig, auteur du virus *Stealth*, des emplacements OS-indépendants seront privilégiés. *SubVirt* se place au début de la permière partition active. Les données qui s'y trouvaient sont déplacées vers d'autres secteurs inutilisés. Dans le cas d'un système Linux, le mécanisme de *swap* est désactivé et la partition de *swap* est utilisée pour stocker le code du *rootkit*.

La dernière étape consiste à modifier la séquence de démarrage pour que le moniteur virtuel du *rootkit* soit lancé prioritairement. Selon la même philosophie adoptée par le virus *Stealth*, le moniteur virtuel contrôle et supervise les phases critiques de démarrage, de redémarrage à chaud et d'arrêt (`shutdown`) pendant lesquelles le système cible pourrait parvenir à analyser les structures impliquées dans le démarrage du système [75] (voir plus loin).

Ainsi, par exemple dans le cas d'un système Linux, la séquence de démarrage est modifiée en mode utilisateur. Les scripts de (`shutdown`) sont modifiés de sorte que le *rootkit* reste encore actif après que tous les processus aient été tués et avant que le système ne s'arrête totalement.

Sécurité de *SubVirt*

Le *rootkit SubVirt* assure sa propre sécurité vis-à-vis du système cible et de ses applications de sécurité essentiellement du fait que ce dernier n'accède qu'à un disque virtuel et non pas au disque physique.

Différents mécanismes additionnels sont également mis en œuvre par *SubVirt* pour assurer sa protection durant certaines phases critiques de la vie du système :

- lors du redémarrage à chaud, plutôt que de réinitialiser la couche matérielle, seule la couche matérielle virtuelle l'est. Le principe est de créer l'illusion d'un *reset*. Tout moyen externe et alternatif de démarrage restera sous le contrôle du moniteur virtuel. Cette approche était déjà mise en œuvre par des virus comme *Joshi* ou *March 6*;
- *SubVirt* est également capable d'émuler des arrêts de la machine – (`shutdown`) via l'ACPI [2] *(Advanced Configuration and Power Interface)* – qui sont capables de faire basculer le système en mode veille, sans altérer les états mémoire, mais avec un mimétisme quasi-parfait avec un arrêt véritable.

Les auteurs de *SubVirt* indiquent certaines solutions contre des *rootkits* de ce type mais leur efficacité et leur mise en œuvre restent illusoires dans la plupart des cas, en particulier dans celui de *rootkits* qui seraient implantés directement au niveau du BIOS, ou pire par des constructeurs eux-mêmes [75, §5]. Deux cas sont possibles :

- la détection intervient avant le *rootkit*. Il faut obligatoirement démarrer de manière externe à partir de supports de confiance. La solution éventuelle

[11] Il est évident que cela doit être fait de sorte à ce que l'analyse ne puisse trahir sa présence. Il sera donc nécessaire de combiner cela avec des techniques de blindage, de polymorphisme/métamorphisme...

est essentiellement organisationnelle ;
- la détection intervient après le *rootkit*, au sein du système cible. Les auteurs mentionnent la mesure de perturbations du temps CPU, de la mémoire, de la bande passante, de l'espace disque. Mais ces techniques sont illusoires. Un *rootkit* conçu et implémenté de manière optimale soit ralentira son activité de sorte à limiter ces perturbations en deçà de ce qui est mesurable, soit manipulera les mesures retournées au système cible.

À la suite de la publication de la technologie *SubVirt*, une méthode de détection a été proposée sous le nom de technique *RedPill* [117]. Dans certains cas, la détection des *rootkits* à base de machine virtuelle (type *SubVirt*) est possible. Cette technique permet de déterminer si un processus est exécuté au sein d'un environnement virtuel ou non, grâce à l'instruction SIDT, gérant le contenu du registre IDTR *(Interrupt descriptor Table Register)*. Comme ce registre est unique pour une machine donnée, lorsque deux (ou plus) systèmes d'exploitation sont en concurrence, le moniteur virtuel doit reloger (transférer) la valeur du registre IDTR pour le système cible – lequel a été basculé dans une machine virtuelle – afin qu'il ne rentre pas en conflit avec celui du système lié au *rootkit*. Notons que ces techniques de transfert ne sont pas nouvelles et que les plus anciens virus, sous DOS, utilisaient ce principe.

Le problème est que le moniteur virtuel n'est pas capable de savoir si un programme tournant dans une machine virtuelle (au niveau du système cible) sollicite l'instruction SIDT. Cette incapacité lui interdit donc de manipuler une éventuelle requête de ce type. De fait, tout processus tournant au sein d'une machine virtuelle sera en mesure de récupérer la valeur **relogée** de l'adresse de l'IDT. Pour *WMWare*, cette adresse est du type `0xFFXXXXXX` tandis que pour *VirtualPC*, elle est du type `0xe8XXXXXX`. Un processus peut alors facilement déterminer s'il tourne au sein d'une machine virtuelle (présence d'un moniteur virtuel actif) ou non. Le simple programme suivant [117] le permet :

```
/* VMM detector, based on SIDT trick
 * written by joanna at insiblethings.org
 *
 * should compile and run on any Intel based OS
 *
 * http://invisblethings.org
 */
#include <stdio.h>
int main () {
 unsigned char m[2 + 4],
     rpill [ ] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
 *((unsigned *)&rpill[3]) = (unsigned)m;

 ((void(*) ())&rpill) ( );
 printf(("idt base: %#x\n", *((unsigned *)&m[2]));
 if(m[5] > 0xd0) printf("Inside Matrix!\n", m[5]);
```

```
 else printf("Not in Matrix.\n");
 return 0;
}
```

Toutefois, cette technique aussi élégante soit elle, souffre d'un défaut. En effet, elle n'est capable que d'identifier la présence d'un moniteur virtuel. Mais cela ne signifie pas nécessairement que ce moniteur virtuel soit lié à un *rootkit* de type *SubVirt*. Le risque de fausses alarmes est donc présent. Une stratégie intéressante, pour un attaquant serait, parmi un grand nombre de machines, de choisir aléatoirement de n'infecter qu'une partie (éventuellement faible) d'entre elles avec un code malveillant à base de *rootkit* de type *SubVirt*. Pour les autres, le système cible est simplement basculé dans une machine virtuelle, sans infecter la machine. Enfin, cette détection ne résout pas le problème de l'analyse et de la restauration du système. Sur un parc important de machines, une attaque de ce type peut devenir un véritable cauchemar à gérer.

7.3.3 Le *rootkit BluePill*

Ce *rootkit*, dénommé *BluePill*,[12] a été présenté en juillet 2006 lors de la conférence *SysCan'06* à Singapour [118]. Son auteur, Joanna Rutkowska a fait à cette occasion une démontration d'installation en direct de son *rootkit* sur une plate-forme Windows Vista x64, prouvant par la même occasion que ce nouveau système d'exploitation n'est pas aussi sécurisé que le prétend l'éditeur. Toutefois, cette démonstration a été à l'origine d'une certaine confusion des esprits. Deux aspects différents ont été mélangés, lesquels sont pourtant indépendants :

- le contournement des protections d'un système d'exploitation donné (en l'occurence Windows Vista x64). Bien que tout code chargé en mode noyau doive obligatoirement être cryptographiquement signé, la démonstration a montré qu'il était possible de charger en mémoire dans le noyau du code non signé, et ce à la volée, sans redémarrage. La technique, utilisant des données documentées dans le SDK *(Software Development Kit)*, est la suivante :
  1. on alloue de la mémoire en quantité excessive à un processus – via la primitive *VirtualAlloc()* ;
  2. le système actualise la pagination mémoire en liaison avec la mémoire physique ;
  3. du fait de l'excès de mémoire allouée au processus, comme il n'y a plus de pages mémoires physiques, il y aura déchargement de certains pilotes ou de portions de code inutilisées sur le disque en *raw-access mode* – utilisation des fonctions `CreateFile(\\.\C:)` et `CreateFile(\\.\PHYSICALDRIVE0)` –, il faut cependant disposer de privilèges administrateur ;

[12] L'auteur a emprunté aux films de la trilogie *Matrix* les appellations à la fois pour son *rootkit* et son outil de détection de code exécuté sous machine virtuelle, *RedPill*.

4. les secteurs du disque où se trouve le fichier contenant ce code (fichier de type *PageFile*) sont alors accessibles en lecture et en écriture, autorisant ainsi l'insertion de code malveillants ;
5. il suffit ensuite de s'assurer que le code sera non seulement chargé à nouveau dans le noyau mais aussi qu'il y sera ensuite exécuté, activant en dépit des protections le code malveillant.

Le code malveillant peut être de n'importe quel type, pas nécessairement un code viral à base de *rootkit*.

- la technologie *BluePill* elle-même qui repose, pour la version présentée, sur une particularité propre, à ce jour, aux processeurs AMD.

Cependant, si cette technologie est efficace – et a priori, elle semble l'être –, la démonstration réussie d'une infection d'une plate-forme annoncée comme sécurisée n'est pas en soi une preuve de son efficacité. Une démonstration n'a jamais été une preuve. En particulier, l'affirmation selon laquelle *BluePill* serait « 100 % » indétectable doit être prise avec énormément de précaution.

Principes de *BluePill*

D'un point de vue conceptuel, le *rootkit BluePill*, pour le peu qui en a été dit par son auteur, ne semble pas très différent du *rootkit SubVirt* présenté quatre mois plus tôt, et ce malgré un certain battage médiatique. Il en reprend les idées maîtresses et semble en améliorer certains aspects et limitations éventuelles.

L'idée est d'utiliser, au lieu d'un moniteur virtuel, un hyperviseur minimaliste qui viendrait contôler le système d'exploitation cible. Le moniteur virtuel réduit, dans le cas de *BluePill*, utilise le mécanisme de virtualisation sécurisée (SVM ou *Secure Virtual Machine*) [4] proposé par les processeurs AMD64 depuis mai 2006. Le système d'exploitation cible est chargé dans une machine virtuelle tandis que l'hyperviseur, lui, s'exécute au sein d'une machine virtuelle sécurisée.

L'intérêt principal de *BluePill* est que sa mise en place se fait à la volée (passage du système d'exploitation cible en machine virtuelle et prise de contrôle de l'hyperviseur), sans modification de structures de données ni redémarrage. Il n'est ainsi pas nécessaire de modifier, comme pour *SubVirt*, la séquence de démarrage, le BIOS ou autre structure de même type impliqué dans un mécanisme de persistance. Contrairement à *SubVirt*, la couche matérielle n'est pas émulée mais est directement accessible à partir de l'hyperviseur.

Limitation et sécurité de *BluePill*

Aussi séduisant soit le *rootkit BluePill*, toutefois, les améliorations ont un coût qui limite sa portée d'utilisation. D'autre part il repose sur des caractéristiques techniques propres à certaines plates-formes récentes (technologie *Pacifica* de virtualisation sécurisée d'AMD). Ainsi si *BluePill* semble plus effi-

cace[13], il est en revanche d'une portée plus limitée et la prévention contre des *rootkits* de ce type semble moins difficile.

Les principales limitations de *BluePill* sont les suivantes :

- le *rootkit* ne survit pas au redémarrage. L'installation se faisant à la volée, aucune structure de donnée n'est modifiée pour assurer un quelconque mécanisme de persistance. Toutefois, comme cela est le cas pour le *rootkit SubVirt*, l'auteur de *BluePill* annonce la gestion des redémarrages à chaud et de l'arrêt de la machine, actions qui peuvent être interceptées par l'hyperviseur ;
- la technique *BluePill* dépend d'une technologie spécifique (technologie SVM). Elle ne fonctionne pas sur les ordinateurs qui ne l'utilisent pas. En outre, cette technologie peut à l'avenir être interdite extérieurement par l'ajout de fonctionnalités essentiellement matérielles, ce qui n'est pas le cas pour des *rootkits* de type *SubVirt* [118, Vues 47 et 48].

Concernant la sécurité de *BluePill* et les capacités de détection, il est pour le moment impossible d'appuyer aveuglément les affirmations de l'auteur, selon lesquelles cette technologie est indétectable à 100 %. Il est clair que sa détection, comme pour *SubVirt*, est particulièrement difficile, voire impossible en pratique dans beaucoup de cas.

Toutefois, il semble que si la détection des *rootkits* à base de machines virtuelles (type *SubVirt*) soit possible dans certains cas par l'utilisation de techniques *RedPill* (voir section 7.3.2), dans le cas de *BluePill*, l'utilisation de machines virtuelles sécurisées et un accès direct et non plus émulé au matériel semblent interdire toute détection, du moins en l'état actuel des connaissances. En effet, l'approche adoptée par *BluePill* permet un cloisonnement total des deux systèmes concurrents – le système d'exploitation cible et l'hyperviseur du *rootkit*. En outre, l'hyperviseur peut totalement manipuler toute requête aux fonctions et ressources basses du système, en provenance du système cible.

La solution n'est donc plus interne à la machine potentiellement corrompue. Des techniques externes, notamment de mesures de temps, finement calibrées et complétées par des techniques statistiques pourraient permettre d'agir contre *BluePill* [116, Vues 35-39]. D'autre part, selon la manière dont l'hyperviseur de *BluePill* s'installe, il reste à prouver que cela ne laisse effectivement aucune trace sur le disque. Dans le cas du contournement de Windows Vista, forcer l'écriture de fichiers de type *Pagefile* peut être analysée statistiquement et montrer certains biais, de nature à trahir une tentative d'installation. Mais n'oublions pas qu'en retour cette analyse statistique peut être également simulée (voir section 3.6).

[13] Notons que pour le moment très peu de détails techniques ont été publiés, qui permettraient d'étudier ces résultats et surtout d'en vérifier la validité, la portée et les limitations. Actuellement, il est impossible de reproduire l'expérience sinon en la reprogrammant en totalité. C'est là une « dérive » préoccupante et regrettable où quelques transparents et une démonstration technique en public tiennent lieu de « preuve scientifique ». En particulier, contrairement aux affirmations pour le moment invérifiables de l'auteur, rien ne prouve que la technologie *BluePill* ne soit transposable aux processeurs *Intel* (technologie *Vanderpool* VT-x).

Même si le recul et plus d'informations sont nécessaires, les technologies *SubVirt* et *BluePill* posent et poseront des problèmes de sécurité probablement impossibles à appréhender en pratique, sinon avec un coût technique, organisationnel et humain très élevé.

Il est certain que le paysage de la sécurité traditionnelle – laquelle consiste à multiplier les barrières de protection comme autant de lignes Maginot (antivirus, pare-feu, logiciels anti-espions...) – est susceptible de subir une véritable révolution et de nous obliger, enfin, à avoir une vision de la sécurité, beaucoup plus globale et proactive qu'elle ne l'est actuellement. Enfin, il n'est absolument pas sûr que ce coût soit compensé par les avantages procurés par les technologies de virtualisation, du moins pour les machines qui doivent impérativement être sécurisées. Ces *rootkits* d'un nouveau type posent avec force le bien-fondé d'une évolution forcenée de la technologie, évolution qui fait de moins en moins bon ménage avec la sécurité.

7.4 Modéliser la furtivité

Le concept de furtivité, à ce jour, n'a pratiquement pas fait l'objet d'études théoriques. La seule tentative se limite à la définition formelle de Zuo et Zhou et d'un unique, mais général, résultat de complexité (voir section 7.1).

L'essentiel du concept de furtivité se résume à un ensemble de techniques, plus ou moins élaborées, qui ont été présentées dans les sections précédentes. Cette absence de formalisation ne permet pas de définir d'une manière générale ce concept, de le distinguer d'autres techniques comme le camouflage et surtout d'en évaluer rigoureusement les qualités. Pour ce dernier aspect en particulier, rien de permet de véritablement juger des forces et faiblesses respectives de chaque méthode, si tant est que cela ait un sens.

Pourtant la notion même de dissimulation suggère assez naturellement un lien avec la stéganographie dont nous proposons la définition suivante.

Définition 7.2 *(Stéganographie et stéganalyse) La stéganographie regroupe l'ensemble des techniques assurant non seulement la protection de l'information (aspect* COMSEC*) mais surtout la protection du canal de transmission de l'information (aspect* TRANSEC*). La stéganalyse regroupe l'ensemble des techniques permettant de détecter l'usage de la stéganographie et d'accéder à l'information stéganographiée.*

En d'autres termes, la stéganographie a pour objectif de cacher l'existence même d'un canal de transmission. Or, le concept de furtivité, tel qu'il est entendu généralement en virologie informatique, consiste principalement à cacher dans un système, que nous comparerons au canal de transmission, des données ou des actions liées à un code malveillant (le message à dissimuler). Cette première comparaison permet alors immédiatement de définir le camouflage comme la stéganographie appliquée à des données « non actives » – le code hors contexte d'exécution –, tandis que la furtivité elle-même sera vue comme de la

stéganographie appliqué à une ou plusieurs actions liées à un code malveillant : le code et/ou les données malveillantes doivent être cachées dans le contexte d'une exécution, que cette dernière soit le fait de ces données elles-mêmes (dissimuler la présence d'un processus) ou qu'elles soient le fait du système lorsqu'il agit ou tente d'agir sur ces données (dissimulation de fichiers).

Nous allons donc tenter d'explorer cette voie et d'esquisser ce qui pourrait être une formalisation de la furtivité.

7.4.1 Stéganographie et théorie de l'information

Il existe plusieurs tentatives de formalisation de la stéganographie. Hopper, Langford et von Ahn [65] ont fondé leur approche sur la théorie de la complexité tandis que Cachin [16] a lui choisi la théorie de l'information et les tests statistiques d'hypothèses. Nous considérerons la seconde dans la mesure où elle est plus adaptée au parallèle que nous souhaitons établir entre la furtivité virale et la stéganographie.

Définissons tout d'abord les principaux termes de la stéganographie, ainsi que quelques notations, et établissons les parallèles avec la furtivité virale :

- un *stégano-medium* C désigne une donnée anodine dans laquelle un *message secret* M peut être dissimulé (mais il ne l'est pas obligatoirement). Nous parlerons de *medium stéganographié*, noté S, pour désigner une donnée anodine dans laquelle un message M est effectivement dissimulé ;
- le stégano-medium correspond aux fichiers, structures et processus d'un système pouvant être utilisés par un code malveillant pour y dissimuler son code, ses données et ses actions ;
- le procédé de dissimulation est réalisé au moyen d'un algorithme de dissimulation lequel est dépendant ou non d'une clef secrète K. Dans le cas de la furtivité classique, la clef est en général absente ;
- l'adversaire de la communication cherche, d'une part, à déterminer s'il existe des messages stéganographiés S au sein d'une population de stégano-medium donnée et, d'autre part, à accéder au message M à partir d'un modèle statistique décrivant la population des stégano-medium C ;
- la population des *stégano-media* C, notée $\mathcal{C}$ est modélisée par la distribution $\mathcal{P}_{\mathcal{C}}$.

Le dernier point montre que l'on peut définir le problème de la stéganographie comme un problème de test d'hypothèses statistiques (voir section 3.2).

D'un point de vue général, un système stéganographique est alors défini de la manière suivante [16].

Définition 7.3 *(Système stéganographique) Soit $\mathcal{P}_{\mathcal{C}}$ une distribution décrivant une population $\mathcal{C}$ de* stégano-media *C. Un* système stéganographique *est un triplet d'algorithmes probabilistes polynomiaux (SK, SD, SE) ayant les propriétés suivantes.*

- *L'*algorithme de génération de clef SK *utilise un paramètre de sécurité n et produit une clef K (la clef stéganographique).*

- *L'*algorithme de dissimulation stéganographique SD *utilise le paramètre* n*, un stégano-medium* C*, la clef* K *et un message binaire* $M \in \{0,1\}^l$ *à dissimuler. Il produit un message stéganographié* S*. L'algorithme* SD *peut utiliser ou non la distribution* $\mathcal{P}_\mathcal{C}$*.*
- *L'*algorithme d'extraction stéganographique SE *utilise le paramètre* n*, la clef* K *et un élément* $C \in \mathcal{C}$*. Il produit soit un message* $M \in \{0,1\}^l$ *soit le symbole spécial*? *indiquant une erreur d'extraction ou l'absence de message* M *(*C *était alors un stégano-medium et non pas un medium stéganographié).*

Pour toute clef K *produite par* SK *et pour tout message secret* $M \in \{0,1\}^l$*, la probabilité*

$$P[SE(n, K, SD(n, K, M) \neq M] \text{ (Fiabilité du système stéganographique)}$$

doit être une valeur négligeable en n*.*

Il est alors possible d'établir un parallèle entre une technique de furtivité et un système stéganographique. Ce parallèle est très général : la plupart des techniques de furtivité n'utilisent pour le moment pas de clef K (et donc pas d'algorithme SK). Les algorithmes ne sont pas non plus probabilistes. Toutefois, cette définition concerne bien, d'une part, les techniques de furtivité classiques en tant que sous-ensemble trivial, et surtout, d'autre part, permet d'envisager et d'identifier des systèmes de furtivité beaucoup plus élaborés que ceux actuellement connus. En particulier, la combinaison de techniques purement stéganographiques avec des techniques de furtivité classiques devrait permettre de considérer de nouvelles classes pour ces dernières.

L'intérêt du parallèle établi entre la furtivité et la stéganographie réside dans le fait que, dans les deux cas, la simple détection du canal (compromission de l'aspect TRANSEC) permet au défenseur d'agir, même si l'aspect COMSEC est préservé. En effet, dans le cas de la stéganographie, identifier l'existence d'une communication secrète, même si elle est chiffrée, permet, au minimum d'agir contre le canal (brouillage ou coupure). Dans le cadre de la furtivité, même si le code malveillant n'est pas identifié, la détection de sa présence suffit à mettre le système en quarantaine.

7.4.2 Sécurité de la furtivité

La vision en tant que système stéganographique rend possible une tentative de modélisation de la sécurité procurée par la furtivité (pour un code qui la met en œuvre) face à une analyse du système. Pour ce faire, nous allons utiliser les concepts développés dans [16] et les appliquer aux techniques de furtivité, dans leur acception la plus large, ce qui inclut les *rootkits*. Pour cela nous noterons, en nous fondant sur les parallèles établis précédemment, *DSys* la distribution décrivant les fichiers, structures et processus « stégano-medium » possibles et *DFurt* la distribution des fichiers, structures et processus « stégano-medium »

effectivement utilisés par des techniques de furtivité. Nous pouvons alors poser la définition qui suit[14].

Définition 7.4 *Un système de furtivité est dit ϵ-sécurisé contre une attaque passive si*

$$D(P_{\mathrm{DSys}}||P_{\mathrm{DFurt}}) = \sum_{x \in Q} P_{\mathrm{DSys}}(x) \log \left(\frac{P_{\mathrm{DSys}}(x)}{P_{\mathrm{DFurt}}(x)} \right) \leq \epsilon.$$

Si $\epsilon = 0$ alors le système est dit parfaitement sécurisé.

En d'autres termes, la sécurité d'un système de furtivité se définit comme l'entropie relative entre les deux distributions *DSys* et *DFurt*. Nous avons $\epsilon = 0$ lorsque les deux distributions *DSys* et *DFurt* sont identiques.

Le cadre fixé ici est celui d'analyses passives, c'est-à-dire d'analyses qui ne modifient pas le système. Cela correspond au plus grand nombre de techniques de détection. Si l'on admet des analyses « actives » – l'analyste modifie la distribution *DSys* et très probablement la distribution *DFurt* également –, il est alors nécessaire de considérer une modification Y apportée par cette analyse. La définition précédente impose alors de considérer non plus les probabilités P_Q mais les probabilités conditionnelles $P_{Q|Y}$. Toutefois, conceptuellement, le modèle reste identique.

La définition 7.4 propose seulement un modèle théorique de la furtivité, dans son acception la plus large. Elle ne précise pas comment mesurer ou caractériser en pratique une différence entre deux distributions, sachant qu'en règle générale la distribution *DFurt* sera a priori inconnue et que définir la distribution *DSys* est extrêmement compliqué[15]. Nous sommes à nouveau dans le cadre présenté dans le chapitre 3, de la modélisation statistique et de son corollaire, celui de sa simulabilité. C'est là l'un des principaux intérêts du modèle proposé dans la définition 7.4.

En reprenant la classification proposée par C. Cachin [17], et en la transposant aux techniques de furtivité, ces dernières peuvent alors être classées en trois catégories selon la sécurité qu'elles procurent. La notion de sécurité concerne ici la capacité à rester indétecté, face à un adversaire dont l'objectif est prioritairement de déterminer si dans un système donné des codes furtifs sont actifs on non.

- Les *techniques furtives inconditionnellement sûres*[16]. L'adversaire dispose de capacités illimitées en temps et en mémoire. Cependant, pour ces systèmes, nous avons $\epsilon = 0$. L'auteur du *rootkit BluePill* affirme que son système appartient à cette catégorie. Il n'est pas possible, à l'heure actuelle, d'infirmer ou de confirmer cela. Il se peut, par exemple, que la prise de

[14] La notation $\mathcal{P}_Q(x)$ désigne la probabilité de x relativement à la distribution Q.

[15] En outre, le choix des estimateurs décrivant *DSys* est dicté par la nature des techniques de furtivité utilisées.

[16] Ces techniques sont conceptuellement comparables aux techniques de chiffrement relevant du secret parfait, tel que défini par C. E. Shannon.

contrôle par le *rootkit*, laisse des traces (par exemple, lors de l'écriture sur le disque de fichiers de type *PageFile*), lesquelles modifient la distribution *DSys* en distribution *DFurt*, relativement à certains estimateurs.

- Les *techniques furtives statistiquement sûres*. L'adversaire dispose d'un algorithme non borné en temps et en mémoire. En outre, $\epsilon = \mathcal{O}(\frac{1}{n})$ est une fonction négligeable[17] en n. Des techniques furtives de type *SubVirt*, en fonction de la technique de prise de contrôle, semblent pouvoir être classées dans cette catégorie. En effet, les modifications du système (au niveau de la séquence de démarrage par exemple) peuvent être dissimulées avec un paramètre de sécurité arbitrairement grand.
- Les *techniques furtives calculatoirement sûres*. L'adversaire dispose seulement d'un algorithme probabiliste polynomial et $\epsilon = \mathcal{O}(\frac{1}{n})$ est une fonction négligeable en n.
- Les *techniques furtives non sûres*. L'attaquant dispose d'un algorithme polynomial déterministe. En outre, la valeur ϵ ne dépend pas de n. Cette catégorie regroupe la plupart des techniques de furtivité classiques et modernes. L'absence de sécurité de ces techniques provient du fait que les modifications apportées au système, en vue de la dissimulation, sont telles que seule la connaissance de la distribution *DSys* est nécessaire pour la détection. En d'autres termes, les modifications sont très facilement identifiables. En outre, l'absence de clef et d'un système cryptographique associé rend la détection du système de furtivité quasi-inévitable. De ce fait, les techniques de cette classe sont en tous points comparables aux techniques de stéganographie simples (sans clef), dont la sécurité repose uniquement sur le secret du procédé de dissimulation.

La formalisation de la furtivité, calquée sur celle utilisée pour la stéganographie et la classification qui en découle, permet de voir que la furtivité n'en est qu'à ses balbutiements. Les techniques connues – celles qui ont été finalement détectées – relèvent de la classe la plus faible. Le statut de codes de type *SubVirt* ou *BluePill* est encore incertain mais il est probable qu'ils appartiennent à des classes supérieures. L'étude de ces systèmes permettra de le confirmer ou de l'infirmer.

7.5 Conclusion

La conception de systèmes de furtivité inconditionnellement sûrs ou au minimum statistiquement sûrs représente un challenge et un nombre conséquent de problèmes techniques ouverts. Dans tous les cas, la classification précédente montre clairement la voie pour l'attaquant. Combinées avec des techniques de simulabilité de tests, les techniques de furtivité, au sens général du terme, vont – si ce n'est déjà le cas – représenter une menace très difficile voire impossible à gérer, à l'avenir. Car finalement comment lutter contre ce qui est invisible ?

[17] Le paramètre de sécurité n a précisément pour rôle de faire en sorte que $\lim_{n\to\infty} \epsilon = 0$.

Mais, le problème peut être vu, de manière très intéressante, dans l'autre sens. Après tout, les techniques de furtivité, en elles-mêmes, ne sont ni dangereuses ni criticables. La société Sony l'a bien compris [114]. Et d'autres sociétés semblent s'intéresser très fortement à ces techniques. Elles peuvent même représenter une réponse pro-active et préventive contre les attaques malveillantes elles-mêmes. En effet, nous pouvons imaginer protéger nos systèmes en intégrant systématiquement des fonctions et logiciels de sécurité équipés de technologies de type *SubVirt* ou *BluePill*. L'attaquant sera alors pris à son propre piège. Toute attaque sera inévitablement confinée.

Mais cette solution réclame une révolution des esprits car qui est prêt à l'accepter, même pour davantage de sécurité, face aux risques sociétaux qu'elle représente ?

Chapitre 8

Résister à l'analyse : le blindage viral

8.1 Introduction

La lutte antivirale est uniquement et nécessairement déterminée par la capacité à, d'une part disposer d'un échantillon de code malveillant et d'autre part, à procéder à son analyse. Cette dernière se fait par désassemblage/décompilation (produire un code source en assembleur/langage de haut niveau à partir d'un code binaire) et analyse du code par débogage (mode pas à pas). La connaissance acquise par ce travail permet alors de mettre à jour les bases de signatures des antivirus, voire des moteurs eux-mêmes.

Cette phase d'analyse est particulièrement critique dans un contexte de prolifération des souches virales. Les éditeurs annoncent une moyenne de 400 nouveaux codes malveillants par mois en « saison creuse » et jusqu'à 1 200 en « haute saison » (le premier semestre 2004 par exemple). L'analyse d'un code « trivial » (une simple variante d'une souche connue) peut nécessiter uniquement une heure, alors qu'un code plus élaboré (une nouvelle souche) requérera plus de temps. Il est alors aisé de comprendre que la mise à jour effective d'un produit antivirus est beaucoup plus lente que les chiffres, quelquefois absurdes, annoncés par les éditeurs.

L'expérience a montré que le délai existant entre l'apparition « dans la nature » d'un code malveillant donné et sa prise en compte par la mise à jour était en moyenne de 24 à 48 heures. Pour le mesurer, les dates de réception ou d'interception du code et la date de prise en compte par l'antivirus ont été considérées. Pour certains codes, semble-t-il, diffusés en un faible nombre de copies, ce délai peut être plus important. Pour certains codes utilisés dans des attaques ciblées et dont l'analyse a été menée en laboratoire, aucune mise à jour n'est encore disponible, et ce, plusieurs mois après l'identification manuelle du code.

Certains programmeurs de virus ont alors cherché à rendre cette phase d'analyse encore plus difficile en implémentant diverses techniques dont l'objectif est de retarder l'analyse d'un code et la compréhension de ses mécanismes : techniques de chiffrement, d'obfuscation, de réécriture... Il s'agit de techniques regroupées sous le terme générique de *blindage de code*. Précisons les choses avec la définition suivante.

Définition 8.1 *(Codes blindés) Un* code blindé *est un programme contenant des instructions ou des mécanismes algorithmiques dont le but est de retarder, contrarier ou interdire son analyse soit durant son exécution en mémoire soit après rétro-ingénérie logicielle (désassemblage/décompilation).*

L'exemple le plus célèbre de virus blindé est probablement le virus *Whale*, qui a fait son apparition aux début des années 90. Depuis, des codes malveillants sont régulièrement analysés, qui tentent, avec plus ou moins d'efficacité, de réaliser du blindage viral : du ver *W32/Mydoom* qui chiffre le nom des variables avec un simple système de César (décalage de 13) au code protégé par un logiciel comme *Armadillo*, la gamme des possibilités est large. Si la plupart du temps ces techniques, pour leur grande majorité, ne contrarient pas une analyse manuelle, et par conséquent la mise à jour des produits, dans certains cas, la gestion automatisée de leur détection peut être efficacement et sérieusement mise en défaut. Il ne faut pas oublier que le facteur temps joue toujours contre la défense : un code malveillant peut finaliser son action au bout de plusieurs minutes voire plus. Cela est inconcevable pour un antivirus dont le temps d'analyse doit être réduit au minimum sous peine de lui voir préférer par l'utilisateur un produit concurrent. Dans ce qui suit, nous considérerons uniquement le cas de l'analyse manuelle selon le principe que si l'on parvient à contrarier voire à empêcher cette phase initiale, la phase automatique au niveau de l'antivirus sera également concernée. Pour le reste – la phase manuelle a été couronnée de succès –, l'analyse des performances et des méthodes algorithmiques (voir les chapitres de la première partie de cet ouvrage) des produits antiviraux permettra au pirate de déterminer comment compliquer au mieux la détection.

Mais jusqu'à présent, l'analyse au minimum manuelle a toujours été possible et couronnée de succès[1]. La raison de cet échec apparent des tentatives de blindage tient essentiellement à deux aspects :

- les analystes de codes malveillants finissent toujours par obtenir une copie des codes, du moins dans le cas des attaques « grand public ». Cela tient au fait que les attaquants ne limitent en général pas la virulence[2] de leurs codes. Obtenir la copie d'un ver qui se répand sur un réseau planétaire comme Internet n'est pas difficile. C'est tout à fait différent pour les

[1] Il faut cependant garder à l'esprit que, dans ce domaine, il n'est possible d'évoquer que ce que l'on connaît et que ce que les éditeurs ont bien voulu rendre public.

[2] La *virulence* est un index mesurant le niveau de risque pour un code autoreproducteur. Cet index, qui a été défini dans [38, chapitre 4, pp. 89 et suivantes] est lié au nombre de copies du code qui ont été produites en fin de vie du code.

attaques ciblées, pour lesquelles quelques dizaines ou quelques centaines de copies tout au plus circulent ;
- les techniques de blindage utilisées en général sont tellement faibles qu'elles sont vouées à l'échec. Cela tient au fait que les problèmes sous-jacents (les problèmes que l'analyste doit résoudre pour contourner le blindage) ont une complexité polynomiale. À titre d'exemple, les techniques de chiffrement utilisées sont trop frustres pour résister à une (crypt)analyse basique. L'espace clef est en général tellement faible qu'une technique de recherche exhaustive – par exemple la technique générique de *X-Raying* [130, chapitre 11]) en vient facilement à bout. Les procédés de chiffrement eux-mêmes sont généralement très faibles.

Les techniques de blindage de code peuvent être divisées en trois classes :
- l'*obfuscation* de code. L'objectif est de transformer un programme en un autre programme fonctionnellement identique au précédent mais qu'il est plus difficile d'analyser (par désassemblage/décompilation et débogage). En d'autres termes, le programme est réécrit afin de réduire plus ou moins fortement sa lisibilité et sa compréhension par l'analyste. Trois grands types de techniques d'obfuscation sont généralement utilisés[3] :
  - les transformations lexicales (par exemple, la modification des noms de variables) ;
  - les transformations du flux d'exécution (l'objectif est de rendre le suivi des étapes d'exécution le plus complexe possible ; l'enchevêtrement de code ou l'insertion de code « placebo » sont deux méthodes fréquemment utilisées) ;
  - les transformations des flux de données (actions sur les structures de données en modifiant le stockage, le codage, l'agrégation ou l'ordre des données durant l'exécution).

  Le lecteur pourra consulter [23, 28, 105, 122] pour une présentation détaillée de ces techniques. Dans un contexte viral, l'obfuscation de code est d'un intérêt et d'une efficacité limités, du moins pour les techniques généralement considérées. Nous présenterons plus en détail des résultats théoriques généraux concernant l'obfuscation dans la section 8.2. Ces résultats permettent de mieux appréhender la portée du concept d'obfuscation. En particulier, l'échec actuel de ces techniques tient à une mauvaise perception du potentiel pourtant important de l'obfuscation mise en œuvre dans un contexte viral ;
- le *polymorphisme/métamorphisme*[4]. Là, l'objectif est de faire varier le code le plus souvent possible afin de contrarier fortement l'analyse. L'analyste doit étudier plusieurs codes différents – certes le plus souvent fonctionnellement identiques. Nous ne considérerons que le polymorphisme par réécriture de code, qui est la seule technique à mériter l'appellation de polymorphisme. Nous sommes ici dans le cadre théorique évoqué

[3] Le lecteur pourra consulter [69] pour étudier quelques exemples appartenant à chacun de ces types de transformations.
[4] Le lecteur trouvera la définition de ces termes dans la section 6 ou dans [38].

dans [38, chapitre 2] avec le théorème de récursion de Kleene. Ce polymorhisme génère plusieurs codes source différents, auxquels correspondent des codes exécutables également différents[5], mais fonctionnellement identiques. En règle générale, les techniques de polymorhisme par réécriture de code finissent toujours par être contournées par les analystes, au bout d'un temps quelquefois plus long que pour une analyse technique. La gestion en automatique par les logiciels antivirus est une autre affaire. Ce type de polymorphisme a été étudié dans le chapitre 6. Nous verrons que les techniques généralement rencontrées correspondent, pour l'analyste, à des problèmes de complexité polynomiale. Nous envisagerons alors d'autres techniques pour lesquelles la complexité est telle qu'en pratique elle rend l'analyse impossible. Le lecteur pourra consulter [100] pour une présentation intéressante du polymorphisme ;

- le *chiffrement.* Il s'agit en fait d'une forme faible de polymorphisme : à un code source unique et au code exécutable qui en dérive correspondent plusieurs exécutables chiffrés, fonctionnellement identiques. Le changement de clef de chiffrement modifie à chaque fois la forme du code. En outre, l'usage partiel ou total du chiffrement vise à perturber l'analyse. Les techniques mises en œuvre jusqu'à présent, du moins pour les codes officiellement connus, n'offrent aucune résistance vraiment sérieuse et ce, pour deux raisons principales :
  - les procédés de chiffrement sont faibles voire très faibles [25] : procédé de masquage constant, procédé de substitution type ROT13 – un simple procédé de César de décalage 13 –, fonctions arithmétiques triviales (ADD, SUB, XOR, NEG, NOT, ROL, ROR comme dans le virus *DarkParanoid* ou *Whale*)... ;
  - la gestion de la clef de chiffrement est généralement calamiteuse. L'analyse de la partie initiale du code permet facilement, le plus souvent, de trouver la clef. Un simple « décodage » suffit ensuite. Par nature mobiles, les codes malveillants contiennent la clef de déchiffrement dans leur propre code.

Il est nécessaire de comprendre que la nature essentiellement **déterministe** – même si dans certains cas très rarement rencontrés une approche probabiliste partielle soit tentée – de l'algorithmique fait qu'au final, l'analyste, au bout d'un temps plus ou moins long, viendra à bout de toutes les techniques mises en œuvre par le pirate pour blinder le code. Toute la stratégie de l'attaquant consiste à ce que le temps d'analyse manuelle, et la complexité sous-jacente, jouent en sa faveur. Même l'utilisation conjointe des techniques précédemment décrites n'est jusqu'à présent pas parvenue à faire échouer une analyse – le meilleur exemple, à ce jour, est probabblement celui du virus *Zmist* [34]. Nous allons, dans ce chapitre, montrer, avec la technologie des codes dénommés BRADLEY, comment, malgré le déterminisme algorithmique, il est possible de réaliser un blindage total qui rend, dans des conditions opérationnelles ac-

[5] Cette condition est indispensable mais elle n'est pas systématiquement réalisée, du fait de l'optimisation réalisée par certains compilateurs, comme `gcc` par exemple.

ceptables, l'analyse d'un code impossible. Nous rappellerons quelques résultats théoriques concernant l'obfuscation de code puis nous présenterons un exemple historique de virus blindé : le virus *Whale*.

8.2 Le problème de l'obfuscation de code

L'obfuscation peut être définie comme le processus consistant à rendre un programme ou des circuits logiques (dans le cas de logique câblée) inintelligible ou du moins le plus difficilement compréhensible possible en cas d'analyse. Lors de la compilation, un code source compréhensible – lorsqu'il est bien écrit – est traduit en code binaire. La structure de ce dernier suit à peu près celle du code source dont il est issu. Cette propriété permet de mettre en œuvre des techniques de rétro-ingénierie (désassemblage ou décompilation) et ainsi de retrouver à partir du binaire, sinon le code source original, du moins un code fonctionnellement équivalent. Ce procédé, déjà long et difficile en l'absence de toute technique de protection, peut devenir extrêmement complexe lorsque le code a été rendu volontairement inextricable.

Les programmes commerciaux sont particulièrement concernés par le problème de l'analyse de binaire. Cette dernière est, en effet, susceptible de révéler le savoir-faire du programmeur et de permettre l'utilisation du logiciel de manière frauduleuse. Cette préoccupation est également celle des programmeurs de codes malveillants, mais dans ce contexte, l'objectif est de perturber l'action antivirale le plus longtemps possible. Le code doit être le moins détectable possible. Les techniques de polymorphisme et de métamorphisme (voir chapitre 6) ont pour but de réaliser une telle lutte anti-antivirale. Les techniques d'obfuscation peuvent de ce fait être vues comme relevant des techniques de polymorphisme/métamorphisme. Il n'est d'ailleurs pas évident qu'une distinction soit vraiment pertinente et, comme nous le verrons dans la section 8.6, il est plus intéressant d'utiliser la notion unifiée de protection de code.

Dans ce contexte général, l'étude de l'obfuscation est intéressante et plusieurs questions se posent.

- Comment définir l'obfuscation ?
- Peut-on réellement obfusquer un programme ?
- Dans la négative, comment malgré tout protéger un exécutable ?

Les travaux les plus aboutis à ce jour sont ceux de Barak *et al.* [8]. Ces auteurs ont proposé une formalisation de la notion d'obfuscation. Leur résultat majeur a été de prouver ensuite qu'il n'existe pas réellement d'obfuscateurs efficaces. Ils ont étendu ce résultat d'impossibilité à des versions plus souples de la notion d'obfuscation : les obfuscateurs approchés – c'est-à-dire des obfuscateurs produisant des programmes approximativement équivalents.

Des travaux plus récents [10] ont repris la formalisation de Barak *et al.* et proposé une nouvelle définition de la notion d'obfuscateur. Mais il a été montré que, même pour cette nouvelle formalisation, l'obfuscation débouche sur une impossibilité. Ces travaux ont alors proposé la notion de τ-obfuscation, qui

consiste à imposer que la protection du code reste effective pendant un temps au moins égal à τ. Pour cette dernière, différents scénarii seront proposés dans la section 8.6.

Nous reprenons ici les résultats de Barak *et al.* [8] et ceux présentés dans [10]. Les démonstrations seront omises : elles ne sont pas essentielles à la compréhension. Le lecteur pourra cependant les consulter dans les articles originaux.

8.2.1 Notations et définitions

Notations

Considérons deux algorithmes A et B. Nous noterons $|A|$ la taille de A alors que $t(A(x))$ désigne le temps d'exécution de A sur une entrée x – avec la convention que $t(A(x)) = \infty$ si A ne s'arrête jamais. On dira que A appartient à la classe $\mathcal{PPT}$ ou que $A \in \mathcal{PPT}$ si A est un algorithme probabiliste en temps polynomial. Notons $A(x) \nearrow$ le fait que l'algorithme A ne s'arrête jamais sur l'entrée x (pas de résultat en sortie).

Les programmes A et B seront dits fonctionnellement équivalents, et nous noterons $A \equiv B$, s'ils produisent le même résultat sur leurs entrées. Si A et B sont définis sur les domaines respectifs $\mathcal{D}_A$ et $\mathcal{D}_B$, alors nous pouvons écrire :

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

Considérons A et B comme des algorithmes probabilistes. Soient deux entrées x et y. Nous noterons $\Pr[A(x)] \approx \Pr[B(y)]$ le fait que les sorties de A et B sur leur entrée x et y ont la même distribution, à des différences négligeables près. La notation $A(1^t)$ désigne le fait que le temps d'exécution de l'algorithme A est inférieure ou égale à t :

$$\forall x \in \{0,1\}^\star, A(x, 1^t) = \begin{cases} A(x) & \text{si } A(x) \text{ peut être évaluée en un temps au plus } t \\ \perp & \text{sinon (la fonction associée à } A \text{ n'est pas définie en } x) \end{cases}$$

La classe des programmes TC_0 désigne la classe des programmes polynomiaux. Autrement dit, leur temps d'exécution s'exprime comme une fonction polynomiale en fonction de la taille de l'entrée x. Si $P \in TC_0$, alors il existe un polynôme p tel que :

$$\forall x, t(P(x)) = p(|x|)$$

Obfuscation

On dira que l'on a un accès de type oracle à un programme P quand nous n'avons pas accès à la description interne du programme P, mais pour chaque entrée x que l'on soumet à P, on a accès au résultat $P(x)$ en temps polynomial.

Un accès de type oracle est donc équivalent à une analyse en boîte noire. Un algorithme ayant un accès de type oracle à un programme P est noté A^P.

Soit Ω l'ensemble des programmes obfuscateurs (tels qu'ils vont être définis dans ce qui suit) et soit Π l'ensemble des programmes que l'on souhaite protéger par des obfuscateurs de Ω. L'ensemble $\overline{\Omega}$ désigne l'ensemble des désobfuscateurs[6] tandis que l'ensemble $\overline{\Omega}_{\mathcal{O}}$ est l'ensemble des désobfuscateurs relativement à un obfuscateur donné $\mathcal{O} \in \Omega$.

Soit R le résultat produit sur des programmes de Π. Nous dirons que R est *trivial* s'il peut être calculé pour un programme P simplement en faisant des requêtes de type oracle sur P et pour certaines entrées spécifiques. Un obfuscateur peut donc être vu comme un programme capable de dissimuler tout résultat non trivial. En d'autres termes, tout résultat qu'un adversaire peut calculer à partir d'un programme obfusqué est en réalité trivial.

8.2.2 Formalisation de Barak *et al.* et variations

Avec les notations précédentes, nous pouvons maintenant préciser ce qu'est l'obfuscation en donnant la définition de Barak *al.* [8].

Définition 8.2 *Un algorithme probabiliste $\mathcal{O}$ est un obfuscateur s'il satisfait les propriétés suivantes.*

- propriété de fonctionalité *: pour tout $P \in \Pi$, P et $\mathcal{O}(P)$ calculent la même fonction ;*
- propriété de ralentissement polynomial *: pour tout $P \in \Pi$, le temps de calcul et la taille de $\mathcal{O}(P)$ sont au plus polynomialement supérieurs respectivement au temps de calcul et à la taille de P ;*
- propriété de « boîte noire virtuelle » *: pour tout adversaire $A \in \mathcal{PPT}$, il existe un simulateur $S \in \mathcal{PPT}$ tel que :*

$$\forall P \in \Pi,\ \Pr\left[A\left(\mathcal{O}\left(P\right)\right)\right] \approx \Pr\left[S^P\left(1^{|P|}\right)\right]$$

Un obfuscateur sera dit efficace si son temps d'exécution est polynomial en la taille de ses entrées. La propriété de *« boîte noire virtuelle »* signifie que tout ce qui peut être calculé efficacement à partir de $A\left(\mathcal{O}\left(P\right)\right)$ peut être efficacement calculé par un accès de type oracle à P.

Le résultat essentiel établi par Barak *et al.* est le suivant :

Théorème 8.1 *Sous réserve qu'il existe des fonctions à sens-unique*[7] *alors, il existe un ensemble de fonctions inobfuscables.*

[6] La notion de désobfuscateur sera définie dans la section 8.2.2.

[7] Une fonction à sens unique est une fonction f telle qu'il est calculatoirement « facile » (autrement dit en temps polynomial) de calculer $f(x)$ pour tout x (en utilisant éventuellement un ordinateur), mais étant donné y, il est calculatoirement impossible de calculer un x tel $y = f(x)$.

Étant donné sa complexité et sa longueur, nous ne donnerons pas la preuve ici. Le lecteur consultera [8]. L'approche générale est de montrer, en considérant différentes constructions et définitions, qu'à chaque fois il est possible d'exhiber des programmes qui ne peuvent être obfusqués. Pour démontrer ce résultat, les auteurs utilisent des fonctions à sens unique, dont l'existence fait partie des grands problèmes ouverts des mathématiques.

Le théorème d'impossibilité de Barak *et al.* a été généralisé de la manière suivante [10].

Proposition 8.1 *Pour tout résultat non trivial R concernant un programme de Π, il existe un programme $P \in \Pi$ tel que, quel que soit l'obfuscateur considéré, $R(P)$ ne peut être dissimulé.*

Ce résultat d'impossibilité a conduit à imaginer d'autres définitions formelles de la notion d'obfuscation. Mais le même résultat d'impossibilité s'applique pour toutes ces définitions [10, §3.2 et 3.3].

La notion de méthode d'obfuscation conduit de manière logique à réfléchir aux méthodes inverses, dites de désobfuscation. Comment formaliser cette dernière notion et quels résultats peut-on en espérer ?

La désobfuscation peut être envisagée comme la transformation d'un programme incompréhensible en un programme intelligible et fonctionnellement équivalent. Plus formellement, cela consiste à produire un nouveau programme avec lequel les résultats qu'il était calculatoirement difficile de calculer sur le programme obscur deviennent faciles (au sens de la complexité). Autrement dit, des résultats non triviaux qui étaient dissimulés par l'obfuscation doivent pouvoir être calculables sur sa version désobfusquée. La taille et le temps d'exécution du programme désobfusqué sont nécessairement une fonction polynomiale (en la taille et temps d'exécution du programme obfusqué) sinon il ne serait pas intelligible. Un désobfuscateur sera dit efficace s'il s'exécute en temps polynomial en fonction de la taille du programme d'entrée. Cela implique que si des programmes obfuscateurs existaient, nous pourrions prouver que les désobfuscateurs ne peuvent être efficaces. Sinon, en considérant un résultat révélé par désobfuscation, il aurait pu être calculé en un temps polynomial à partir d'un programme obfusqué que l'on aurait préalablement désobfusqué. La formalisation de la désobfuscation ne contredit pas, de façon logique, celle de l'obfuscation.

Pour définir formellement la notion de désobfuscation, nous partons du principe qu'il doit satisfaire la propriété d'*intelligibilité*. Alors, en reprenant la définition 8.2 de Barak *et al.*, nous obtenons la définition suivante.

Définition 8.3 *[10](Désobfuscation) Un algorithme probabiliste D est un désobfuscateur s'il satisfait les deux propriétés suivantes.*

- propriété de fonctionalité *:*

$$\forall \mathcal{O} \in \Omega, \forall P \in \Pi,\ D(\mathcal{O}(P)) \equiv P$$

- propriété d'intelligibilité *:*

$$\begin{aligned}&\forall \mathcal{O} \in \Omega, \forall R \in \mathcal{PPT},\\ &\quad \exists T \in \mathcal{PPT} \text{ } s.t. \text{ } t(T) = \mathrm{O}(t(R)) \text{ } et\\ &(\forall P \in \Pi,\ \Pr[T(D(\mathcal{O}(P)))] \approx \Pr[R(P)])\end{aligned}$$

Il est possible de restreindre la définition d'un obfuscateur relativement à un obfuscateur donné $\mathcal{O}$. Le désobfuscateur ainsi défini sera noté $\overline{\mathcal{O}}$.

D'une manière moins formelle, la propriété d'intelligibilité implique que tout résultat non trivial pouvant être calculé à partir d'un programme non obfusqué P peut également être calculé à partir de la désobfuscation de toute obfuscation de P, en un temps approximativement équivalent.

Comme attendu, le résultat suivant a été établi.

Proposition 8.2 *[10] Il n'existe pas de désobfuscateur efficace relativement aux définitions de la notion d'obfuscation proposées dans [8] et [10].*

Il faut cependant relativiser ce résultat dans la mesure où la définition de la désobfuscation n'est pas encore suffisamment aboutie. Celle qui a été donnée dans [10] ne prend pas en compte l'existence de méthodes d'obfuscation irréversibles. Cette question figure actuellement parmi les problèmes ouverts.

8.2.3 La τ-obfuscation

Une seconde approche, face aux résultats d'impossibilité présentés précédemment, consiste à assouplir le concept d'obfuscation. Cette approche consiste à demander à ce que l'obfuscation demeure effective au moins pendant un temps donné. En d'autres termes, nous autorisons que des résultats non triviaux puissent être calculés mais en un temps de calcul au moins égal à un certain temps τ. Nous appellerons cela la τ-obfuscation et les programmes qui la mettent en œuvre des τ-obfuscateurs. Ces derniers serons définis de la même manière que les obfuscateurs précédents, excepté en ce qui concerne la propriété de boîte noire virtuelle.

Définition 8.4 *Un algorithme probabiliste $\mathcal{O}$ est un τ-obfuscateur s'il satisfait les propriétés suivantes :*

- propriété de fonctionalité *: pour tout $P \in \Pi$, P et $\mathcal{O}(P)$ calculent la même fonction ;*
- propriété de ralentissement polynomial *: pour tout $P \in \Pi$, le temps de calcul et la taille de $\mathcal{O}(P)$ sont au plus polynomialement supérieurs respectivement au temps de calcul et à la taille de P ;*
- propriété de « boîte noire virtuelle » *: pour tout adversaire $A \in \mathcal{PPT}$, il existe un simulateur $S \in \mathcal{PPT}$ tel que :*

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

Cette propriété décrit le fait que tout résultat pouvant être calculé en un temps inférieur à $\tau \times t(\mathcal{O}(P))$ – où $t(\mathcal{O}(P))$ désigne le temps nécessaire pour obfusquer le programme P – est en réalité calculable par un accès de type oracle sur P.

Cette définition s'applique également à la désobfuscation. En d'autres termes, un τ-obfuscateur est nécessaire pour prévenir toute désobfuscation en un temps inférieur à $\tau \times t(\mathcal{O}(P))$. Contrairement à la notion générale d'obfuscation, nous montrerons qu'il existe des τ-obfuscateurs. Dans la section 8.6, nous décrirons des techniques qui la mettent en œuvre efficacement.

8.3 Le virus *Whale*

Le virus *Whale*[8] a fait son apparition en septembre 1990. D'une taille d'environ 9 Ko, ce virus a tout d'abord surpris par sa taille : c'était le plus gros virus jamais publié à cette époque. Mais davantage que sa taille, c'est sa capacité à résister à l'analyse qui a le plus surpris les spécialistes de l'époque. Le niveau de complexité du code, jamais observé pour un virus, s'est révélé tel que les policiers de Scotland Yard, chargés de l'enquête concernant ce code, ont créé le terme, adopté depuis par l'ensemble de la communauté antivirale, de blindage viral. Pour se faire une idée de la capacité de blindage du virus *Whale*, il suffit de savoir que près de 70 % du code de ce virus sont dédiés aux techniques de blindage.

Whale s'est très vite révélé inefficace, du fait de très nombreux bugs, de sa lourdeur et des fortes perturbations occasionnées dans les ordinateurs infectés, lesquels ne disposaient pas de puissance nécessaire pour exécuter un tel code sans erreur. Apparu plus tard, ce virus aurait pu causer de réels dégâts. Son rôle aura été d'initier la réflexion sur le blindage viral et, d'autre part, de gaspiller l'activité et l'énergie des experts de tous les éditeurs d'antivirus, effet « collatéral » qui n'a certainement pas manqué de réjouir l'auteur du code. L'analyse du virus *Whale* a nécessité près de deux semaines. De ce point de vue, ce virus a atteint son but. De nos jours, la propagation d'un ver à l'échelle planétaire se fait en quelques dizaines de minutes (le meilleur exemple est celui du ver *W32/Blaster*, en janvier 2003). Tout délai dans l'analyse de code peut avoir des conséquences désastreuses.

Nous allons étudier ce virus en prenant de la hauteur et en considérant plus l'aspect algorithmique que le code lui-même (ce qui aurait été impossible sans détailler, jusqu'à écœurement, le code assembleur extrêmement complexe qui utilise intensivement des spécifités du DOS et du BIOS, aujourd'hui sans grand intérêt). L'étude présentée ici – tirée de [43]) se fonde sur l'analyse de code assembleur (produit en 1991 par Ralf Hörner et disponible sur la page web de l'auteur ; pour aider le lecteur et par souci de place, les adresses ou les

[8] Le nom du virus vient de la chaîne de caractères contenue dans le code `Vir_NAME : DB "THE WHALE"`.

noms de procédures seront quelquefois indiqués pour repérer les parties de code évoquées dans cet article).

Le virus *Whale* est non seulement blindé mais également furtif. En revanche, il n'est pas à proprement parler polymorphe. Les techniques polymorphes à base de mécanismes cryptologiques utilisées par *Whale* ont pour principal but le blindage du code. Ce virus infecte essentiellement les fichiers de type `COM` et `EXE` selon un mode tout à fait classique. Il est capable de mettre en œuvre cette infection avec une virulence maîtrisée (une sorte de contrôle des naissances).

La description du virus *Whale* ne portera que sur les fonctions liées aux fonctionnalités recherchées de blindage ou qui y sont liées. Pour une description complète du virus, le lecteur consultera [43].

8.3.1 Les mécanismes d'infection

Les mécanismes d'infection, bien que traditionnels, sont toutefois mis en œuvre de sorte à participer au blindage viral lui-même. Ils sont présentés séparément ici afin de faciliter la vision générale du virus mais le lecteur doit garder présent à l'esprit que tout concourt au blindage du code.

Le virus *Whale*, une fois résident en mémoire (voir plus loin), intercepte les appels à la fonction (service) `4BH` – *Load or Execute Program* de l'interruption `21H` (DOS)). À chaque appel d'un programme (chargement et/ou exécution), *Whale* en profite pour l'infecter. Les fichiers de type `COM` et `EXE` sont donc infectés mais du fait de certaines insuffisances dans le code (absence de contrôles adéquats par exemple), d'autres fichiers (exécutables ou non) peuvent être infectés par erreur et/ou certains de leur attributs (taille et date/heure) être modifiés indûment.

Le mécanisme d'infection est assez basique : le code viral est de type *appender*. Il est ajouté en fin d'exécutable et une fonction de saut permet d'accéder au code viral. Le code viral qui est ensuite copié est en fait une forme mutée. La séquence d'infection est la suivante.

1. Préparation de l'infection : récupération de la date et des caractéristiques des fichiers cibles, recherche d'une infection préalable (lutte contre la surinfection), contrôle divers (nature des exécutables, vérification des tailles et des dates...).

2. Le virus n'infecte que les fichiers dont la date possède une heure supérieure ou égale à 16 (procédure *CheckFileTime* et code situé à l'adresse `J0451F`). Le but est ici de limiter la virulence du virus.

3. Appel de la procédure *Mutate_ Whale* qui génère essentiellement un code muté avec une probabilité de $\frac{1}{2}$ (30 variantes au total sont possibles, contenues dans le code et choisies aléatoirement). Sinon, le code est dupliqué sans mutation (mais le code réellement copié est cependant différent du fait d'autres mécanismes de blindage ; voir plus loin). Le code est ensuite copié (procédure *Code_ Whale*).

4. Si la cible est un fichier de type `COM`, déjà infecté[9] par lui, dans 10 % des cas, *Whale* le désinfecte dans un but de limitation de la virulence (contrôle des naissances ; procédure `@10_Prozent`). Le but ici est de compliquer la détection en maintenant l'activité infectieuse du virus en dessous d'un certain niveau[10].
5. Le virus ajoute ensuite du code mort *(garbage code)*, avec une probabilité de $\frac{1}{10}$ (le champ `TrashFlag` est mis à 1 lors de l'appel à la procédure `@10_Prozent`), par appel à la procédure *Write_ Trash_ To_ File*. La quantité de code mort ajoutée varie d'une infection à l'autre mais reste inférieure à 4 Ko. Le but de ce code mort est d'augmenter la difficulté d'analyse du code. Notons que ce code mort, de taille variable, contribue également à compliquer la détection, car il est accompagné de mécanismes de réalignement des paragraphes du code.
6. Enfin, le virus falsifie dans le système la taille et la date/heure des fichiers infectés à des fins de furtivité, à savoir :
   - l'heure du fichier est diminuée de 16 si elle supérieure à cette dernière valeur ;
   - on retranche `23FCH` octets à la taille du fichier (taille du virus).

   Ces falsifications contiennent de nombreux bugs. En effet, certaines vérifications basiques ne sont pas réalisées (la cible a-t-elle été bien infectée ?, cohérence des valeurs forgées...).

Quand un programme infecté par *Whale* est exécuté (retour de contrôle au programme hôte), le virus se désinfecte de ce programme, en mémoire (adresse `J03428`).

8.3.2 La lutte anti-antivirale

Elle est réalisée par les trois techniques suivantes : furtivité, polymorphisme et blindage spécifique. Mais les deux premières techniques, vu leur niveau jamais rencontré auparavant, ont pour but principal de contribuer activement au blindage final du code viral.

Techniques de polymorphisme

Le polymorphisme de *Whale* est multiple et il poursuit deux objectifs :

- compliquer l'analyse de forme, par un antivirus. C'est essentiellement le rôle des 30 formes mutées (statiques) du code qui est copié dans les fichiers, lors du processus d'infection. Ce polymorphisme est de nature cryptologique. Il est de très faible niveau. Les primitives cryptologiques de chiffrement sont frustes : XOR constant de tous les octets, d'un octet sur deux ou sur trois, addition (fonction ADD) d'une valeur constante, complémentation des octets... À titre d'exemple, nous avons pour la mutation 18, le code suivant :

[9] Un fichier infecté est marqué à l'aide de la valeur `Whale_ID = 020CCH`.

[10] Le virus n'infecte plus après la date du 1er avril 1991 (procédure *Check_ Verfallsdatum*).

```
mut_18    EQU     $
J03E60:   NOT     BYTE Ptr DS:[BX]   ; octet = octet xor FFH
          NEG     BYTE Ptr DS:[BX]   ; octet = -octet
          ADD     BX,+01h            ; octet = octet + 1
          LOOP    J03E60
```

alors que pour la mutation 6, autre exemple, nous avons :

```
J04178:   MOV     AX,0002h         ; AX vaut 2
          .......

J04188:   XOR     BYTE Ptr DS:[BX],67h ; masque constant
          DEC     CX
          ADD     BX,AX            ; on saute un octet
          DEC     CX
          JNZ     J04188           ; seuls 4547 octets sont
                                   ; xorés (un sur deux)
```

Le code de chaque procédure de mutation contient en fait la « clef » utilisée, ce qui réduit l'analyse, sur ce point, à un simple décodage. En outre, le code copié ne mute qu'avec une probabilité $\frac{1}{2}$ et un second procédé polymorphe est appliqué, lequel consiste en une simple permutation des différentes sous-routines ;

- compliquer l'analyse et la détection du code en mémoire dans le cadre du blindage. La technique est assez complexe et s'étale sur la quasi-totalité du code source de *Whale*. Pour simplifier et résumer :
  - le code utilise très fréquemment (exactement 94 fois) un mécanisme de déchiffrement/rechiffrement de son propre code en mémoire (le virus est résident en mémoire mais sous forme presque totalement chiffrée). Le code est déchiffré juste au moment où il doit être utilisé. Il est ensuite immédiatement rechiffré. Pour cela, une clef (en fait des octets générés aléatoirement au moyen de l'horloge interne) est utilisée et stockée (soit directement dans une instruction XOR soit juste après la routine de déchiffrement). Pour cela, il est fait appel à la macro *MDECODE* :

```
   MDECODE     MACRO   Adr
               CALL    DECODE ; la routine est localisée
                              ; à l'adresse J0491B
               DW      @L&adr-L&Adr
L&Adr:
               ENDM
```

et la macro *MCODE* :

```
   MCODE       MACRO   Adr
               CALL    CODEIT  ; la routine est localisée
                               ; à l'adresse J04990
               DB      @L&Adr-L&Adr+1
@L&Adr:
               ENDM
```

Finalement le code ainsi déchiffré puis rechiffré par partie présente la structure suivante :

```
....
MDECODE <numéro de partie de code>
....
partie du code
....
MCODE <numéro de partie de code>
MDECODE <numéro de partie de code + 1>
....
partie du code
....
MCODE <numéro de partie de code + 1>
....
```

- Le code s'autopermute également très fréquemment. Techniquement, cela est réalisé par des permutations fréquentes d'octets dans certaines parties du code à l'aide de valeurs précalculées.

Techniques de furtivité

Là encore, la furtivité a été étudiée pour contribuer largement au blindage final du code, même si certaines techniques avaient déjà été rencontrées dans des virus antérieurs, pour les seuls besoins de camouflage. Pour simplifier, la furtivité est réalisée par :

- un accès intense à la table des vecteurs d'interruption, lesquels sont modifiés par le virus selon ses besoins. Les interruptions utilisées par le virus sont les interruptions 1H (mode pas à pas), 2H (interruption non masquable), 3H (point d'arrêt), 9H (clavier), 13H (accès disques), 24H (erreur critique DOS , notamment pour contrôler les erreurs au niveau du DOS et les masquer), 2FH (gestion de la mémoire) et surtout 21H. Cette dernière étant intensivement sollicitée, le virus utilise la routine suivante de répartition des principaux services utilisés :

```
J02B45:
        if_then  <00fh,L0699>  ; 2EA9  ; open FCB
        if_then  <011h,L04F4>  ; 2D04  ; Findfirst FCB
        if_then  <012h,L04F4>  ;       ; Findnext  FCB
        if_then  <014h,L06E0>  ; 2EF0  ; Read Seq. FCB
        if_then  <021h,L06CA>  ; 2EDA  ; Read Random FCB
        if_then  <023h,L08CF>  ; 30DF  ; Get Filesize FCB
        if_then  <027h,L06C8>  ; 2ED8  ; Read Rndm Block FCB
        if_then  <03dh,L0996>  ; 31A6  ; OPEN FILE / HANDLE
        if_then  <03eh,L09E4>  ; 31F4  ; CLOSE File / Handle
        if_then  <03fh,L1E5E>  ; 466E  ; READ File / Handle
```

```
if_then  <042h,L1DA2>  ; 45B2  ; SEEK / Handle
if_then  <04Bh,L0AD4>  ; 325D  ; EXEC
if_then  <04Eh,L1F70>  ; 4780  ; FindFirst ASCIIZ
if_then  <04Fh,L1F70>  ; 4780  ; FindNext  ASCIIZ
if_then  <057h,L1D0F>  ; 451F  ; Set/Get Filedate
```

- pour échapper à la surveillance, *Whale* se cache en mémoire haute, sous la limite des 640 Ko (en général à l'adresse `9D900H`). Il diminue la quantité de mémoire disponible pour le système de la taille du virus (`2700H` octets). Pour cela, le virus passe par l'interruption `3H` (procédure *Wal_Ins_MEMTOP_Kopieren*). Le DOS est donc leurré car la quantité de mémoire dont il croit disposer est plus faible que celle réellement disponible.

Techniques de blindage

En plus des techniques qui viennent d'être présentées et qui contribuent fortement au blindage du code, le virus *Whale* met en œuvre des techniques spécifiques qui ne relèvent ni de la catégorie des techniques de polymorphisme ni de celles de la furtivité :

- l'utilisation de code sans utilité (par exemple à l'adresse `J02A7D`), de code redondant (code pouvant être simplifié ou factorisé), d'obfuscation qui « balade » l'analyste afin de lui faire perdre le fil de l'analyse... ;
- l'exécution du code est différente selon qu'il s'agit d'un processeur 8088 ou 8086 (les files d'attente pour les instructions diffèrent en taille, respectivement 4 et 6 octets). Ainsi, dans le code situé à l'adresse `J02CDA`, nous avons :

```
                ADD     WORD Ptr [BP+08h],+07h
                XCHG    BX,[BP+08h]
                MOV     DX,BX
                XCHG    BX,[BP+02h]

                SUB     SI,@0478
                MOV     BX,SI
                ADD     BX,SwapCode_6

                POP     BP

                PUSH    CS:[SI+SwapCode_8]
                POP     AX
                XOR     AX,020Ch
                MOV     CS:[BX],AL
                ADD     AX,020Ch
                CALL    EIN_RETURN
J02CFF:         INT     3
```

Sur un processeur 8086, l'intruction `INT 3` sera déjà dans la file et sera donc exécutée comme telle (point d'arrêt). En revanche, sur un processeur 8088, l'instruction sera modifiée avant d'entrer dans la file et sera exécutée comme une fonction `RET` (retour) ;
- mais la fonction la plus intéressante de *Whale* est celle qui consiste à surveiller la présence d'un débogueur au travail (indication que le code est en cours d'analyse, avec utilisation du mode pas à pas et de points d'arrêts). Le virus effectue ce contrôle en surveillant les interruptions `9H` (clavier), `1H` (mode pas à pas) et `3H` (point d'arrêt). Le code effectue ce contrôle en trois endroits (adresses `J03410`, `J04661` et `J04A85`). Il appelle alors la procédure *Debugger_ Check*. Si la présence d'un débogueur actif est détectée, alors *Whale* bloque le clavier et se désinfecte de la mémoire.

8.3.3 Conclusion

Le virus *Whale*, bien qu'ancien, résume à lui seul toutes les techniques anti-antivirales que l'on rencontre de nos jours. Si les techniques de polymorphisme et de furtivité étaient déjà connues et utilisées par des virus plus anciens, en revanche le blindage viral est né avec *Whale*. Ce virus n'a pas eu le succès escompté par son ou ses auteurs, et ce pour les trois raisons suivantes :

- les techniques de blindage (en incluant ici l'usage intensif du polymorphisme par cryptologie) mises en œuvre par ce virus ont finalement produit un code lourd, complexe et donc fortement consommateur de ressources, totalement inadapté aux machines de l'époque. Le ralentissement de la machine infectée, qui pouvait aller jusqu'à un gel de cette dernière n'a pas manqué de provoquer rapidement son identification. De fait, *Whale* était (trop) en avance sur son temps ;
- le code contient beaucoup de bugs voire d'erreurs de conception qui ont très fortement limité son efficacité et aidé à sa première identification. Si, pour l'époque, ce virus représentait une nouveauté, avec le recul, l'analyse du code montre des faiblesses algorithmiques certaines et l'absence d'une réflexion aboutie et pertinente. Il est clair que son ou ses auteurs se sont laissés aveugler par le codage en lui-même – et les spécificités techniques du système d'exploitation considéré – au détriment des aspects algorithmiques fondamentaux ;
- les techniques cryptographiques utilisées, même combinées en grand nombre, n'ont pas offert une très grande protection du code. Trop triviales, les « clefs » étant contenues dans les procédures de chiffrement ; un simple décodage et non pas une cryptanalyse, a très facilement permis d'analyser le virus. En cela également, *Whale* résume ce que l'on peut rencontrer dans les virus les plus récents.

8.4 Le blindage total : les codes Bradley

Avant de présenter la technologie BRADLEY, dans laquelle diverses techniques cryptologiques sont utilisées, il est nécessaire de faire un rapide aperçu de l'usage de la cryptologie dans les codes malveillants – le lecteur pourra également consulter [44, 45] –, non lié au problème du blindage de code.

Les techniques cryptographiques liées à la génération d'aléa ont été récemment considérées pour la propagation optimale des vers informatiques [7]. Par exemple, l'usage de fonctions cryptographiques de hachage est particulièrement adapté pour accélérer la propagation d'un ver comme le ver *Curious Yellow* [144].

L'usage de la cryptologie dans un cadre viral a conduit en 1996 au concept de *Cryptovirologie*, développé dans [29, 147, 148]. La cryptovirologie consiste à utiliser des outils de chiffrement dans des codes malveillants dans la protection de la charge finale (offensive) de ces codes. Plusieurs exemples pertinents ont été présentés dans [147] et des attaques réelles ont depuis été observées (par exemple avec le code *TrojanPGPCoder*). L'approche consiste à rendre la victime d'un virus, dépendante de ce virus – en d'autres termes, la survie du virus est assurée par la dépendance critique de la victime envers lui. Ces résultats sont essentiellement obtenus grâce à des systèmes de chiffrement à clef publique[11]. Le code malveillant peut chiffrer le disque dur de la victime avec cette clef publique et extorquer de l'argent contre la clef privée qui restaure les données en clair. Notons que si la victime éradique le virus avant la restauration des données, ces dernières sont irrémédiablement perdues, à moins de réinfecter la machine. Cependant, un code efficace sera métamorphe et une nouvelle infection concernera une clef ou un algorithme différent du précédent.

Bien que terriblement efficace, cette approche vise uniquement à protéger l'action du virus (sa charge finale) et non le virus lui-même. Cela implique que si une copie du code malveillant est analysée, aucun des outils de cryptographie utilisés n'empêchera l'analyse du code. Cette limitation tient au fait que les cryptovirus tels que définis par Young et Yung ne sont pas capables de gérer une clef secrète d'une manière efficace pour offrir un blindage efficace de code.

Nous allons maintenant considérer une approche différente qui permet d'interdire une telle analyse : il s'agit du *blindage total* de code. Cette approche combinée à celle développée dans [29, 147, 148] peut se révèler extrêmement dangereuse et totalement ingérable du point de vue technique, notamment dans le cas d'attaques ciblées. Seule une politique de sécurité draconnienne a des chances de prévenir un tel risque. Nous présenterons ensuite dans la section 8.4.4 quelques scénarii opérationnels de mise en œuvre des techniques BRADLEY. Le lecteur devra conserver à l'esprit qu'un tel code sera mis en œuvre dans le cadre d'un scénario plus large où le chiffrement aura été conçu de sorte à leurrer la détection de contenu chiffré et l'analyse de trafic (voir section 3.6).

[11] Un *cryptovirus* est défini comme un virus contenant et utilisant une clef publique.

8.4.1 Génération environnementale de clefs

Les codes malveillant sont mobiles par essence. La plupart du temps, ils évoluent dans des environnements qui leur sont naturellement hostiles. Ils peuvent être analysés de sorte que leur code soit totalement accessible à l'analyste[12]. Comme nous l'avons déjà précisé, le chiffrement utilisé dans les codes malveillants utilise des clefs de chiffrement qui sont statiques. Autrement dit, la clef est directement ou indirectement contenue dans le code malveillant lui-même.

En 1998, B. Schneier et J. Riordan [109] ont introduit la notion de *génération environnementale de clef* pour résoudre le problème lié à la faiblesse des clefs statiques. Dans leur modèle, les clefs de chiffrement sont construites à partir de certaines données prises dans l'environnement de l'agent mobile. La génération environnementale de clefs peut donc se révéler extrêmement utile lorsque l'émetteur souhaite communiquer avec un destinataire de telle sorte que ce dernier ne puisse accéder au(x) message(s) émis que lorsque certaines conditions environnementales sont réalisées. En outre, la génération environnementale de clefs peut devenir très utile lorsque les circonstances imposent que le destinataire ne doive pas connaître les conditions environnementales d'activation (l'agent est alors dit *aveugle*). C'est en fait ce dernier cas qui nous concerne dans le contexte viral d'analyse de code. Le « destinataire » est ici représenté par le code malveillant lui-même, présent dans un système donné (ordinateur, réseau), et l'émetteur est l'auteur de ce code ou éventuellement le système cible.

La difficulté de conception d'un protocole de génération environnementale de clefs tient au fait que le modèle de sécurité suppose que tout attaquant (en l'occurence celui qui analyse l'agent mobile) puisse totalement contrôler l'environnement. Toutes les informations disponibles et accessibles à l'agent mobile le sont aussi pour l'attaquant. Toutes les entrées du programme peuvent être fournies au programme par l'attaquant (simulation de l'environnement). Cela implique que tout protocole de génération environnementale de clefs doive résister à une analyse directe (essais exhaustifs par exemple) et à des attaques par dictionnaires ou assimilées.

Les auteurs [109] ont proposé et étudié divers protocoles de génération environnementale de clefs. Pour illustrer leur approche, considérons les constructions basiques suivantes. Nous noterons N un entier correspondant à une observation de l'environnement[13], $\mathcal{H}$ une fonction à sens unique – typiquement une fonction de hachage [92]– , M la valeur de hachage de l'entier N, $\oplus$ l'opération XOR bit à bit, $||$ l'opérateur de concaténation, R_i un entier d'initialisation et K une clef. La valeur M est contenue dans l'agent mobile (un code malveillant

[12] Il est intéressant de noter que les rôles sont renversés. Dans le cas du blindage de code, l'analyste antiviral joue le rôle d'attaquant alors que l'auteur du code et son code jouent celui de la défense. Notons que cette problématique peut concerner également la protection logicielle contre le piratage.

[13] Un tel codage est toujours possible. C'est un cas particulier de codage de Gödel (voir [38, chapitre 2]).

dans notre contexte). Le rôle de la fonction de hachage est de dissimuler les données environnementales d'activation (l'attaquant ne peut retrouver N à partir de M et de H, du moins calculatoirement). Citons quelques constructions possibles, parmi de nombreuses autres :

- si $\mathcal{H}(N) = M$ alors $K = N$;
- si $\mathcal{H}(\mathcal{H}(N)) = M$ alors $K = \mathcal{H}(N)$;
- si $\mathcal{H}(N_i) = M_i$ alors $K = \mathcal{H}(N_1||N_2||\ldots||N_i)$;
- si $\mathcal{H}(N) = M$ alors $K = \mathcal{H}(R_1, N) \oplus R_2$.

Le lecteur remarquera que la première construction est utilisée dans les schémas d'authentification par mots de passe statiques (type Unix). La propriété la plus importante pour toutes ces constructions est que la connaissance de M ne révèle aucune information sur K.

Riordan et Schneier ont proposé plusieurs constructions opérationnelles de protocoles de génération environnementale de clefs utilisant diverses techniques ou schémas : schémas à seuil, utilisation en série de plusieurs clefs, indexation temporelle...

La génération environnementale de clefs a été étudiée par Riordan et Schneier d'un point de vue théorique uniquement. Certains aspects pratiques ont cependant nécessité d'être testés grandeur nature. En particulier, la plupart des constructions qu'ils ont proposées autorisent la récupération de la clef en observant conjointement l'agent mobile et l'environnement. C'est systématiquement le cas dans le contexte de l'analyse de codes malveillants. L'espace clef peut être beaucoup plus réduit en pratique que ne le laisse espérer la théorie. Une attaque par essais exhaustifs est alors facilement réalisable. De plus, en observant les actions de l'agent mobile, l'analyste peut toujours facilement déterminer quelles données recherche l'agent et les endroits où elles se trouvent. Cela implique qu'un analyste patient accédera à ces informations précisément au moment où l'agent est activé par les données environnementales d'activation.

Nous allons à présent considérer comment réaliser un blindage total de code, de manière opérationnelle, à l'aide d'un protocole de génération environnementale de clefs dans lequel les limitations fortes, qui viennent d'être évoquées, ont été supprimées.

8.4.2 Technique générique de blindage total : les codes bradley

Nous allons décrire une technique générique de blindage total qui a été testée et validée à l'aide d'une famille de codes viraux : la famille BRADLEY[14]. Sans perte de généralités, nous ne décrirons dans cette section qu'une technique basique mais néanmoins extrêmement puissante. Des protocoles plus élaborés seront présentés dans la section 8.4.4. Cette technique s'applique bien évidemment aux autres types d'infections informatiques. Deux types de codes ont été conçus et testés :

[14] Cette appellation fait référence aux fameux chars de combat américains...

- un virus générique, visant, dans le cadre d'une attaque ciblée, à frapper un groupe donné de machines ou d'utilisateurs. La taille de ce groupe est paramétrable. Nous nommerons variante A ce type de virus ;
- un virus plus élaboré, destiné à frapper spécifiquement un seul utilisateur. Nous le désignerons comme la variante B.

D'autres variantes ont été testées (voir section 8.4.4) avec succès, mais toutes se rattachent d'une manière ou d'une autre à l'une des deux variantes principales A ou B. Ces codes ont été testés aussi bien pour les systèmes Unix que pour ceux sous Windows. L'approche étant essentiellement algorithmique, la nature du système d'exploitation n'est pas essentielle. Dans tous les cas, ces codes ont pu opérer sans être détectés par aucun antivirus (quinze au total ont été utilisés). Cette dernière constatation n'est pas réellement surprenante même si elle est particulièrement préoccupante. Elle montre – et c'est précisément le but de cette étude – que la solution contre des codes de type BRADLEY n'est plus technique mais réside essentiellement dans une politique de sécurité draconienne visant à la prévention. Une fois qu'un tel code parvient dans un système, il est trop tard. Outre l'absence de détection, il n'y a aucun espoir de mettre jamais à jour les antivirus.

Dans l'étude qui suit, les codes BRADLEY étant des preuves de concept, nous ne nous polariserons que sur l'algorithmique spécifique au blindage de code. Pour les autres techniques, le lecteur pourra lire le chapitre 6 (polymorphisme/métamorphisme) et le chapitre 7 (furtivité). Pour des raisons légales[15], nous ne donnerons pas le code source et nous nous limiterons aux seuls aspects algorithmiques. La structure générale des codes BRADLEY est décrite en

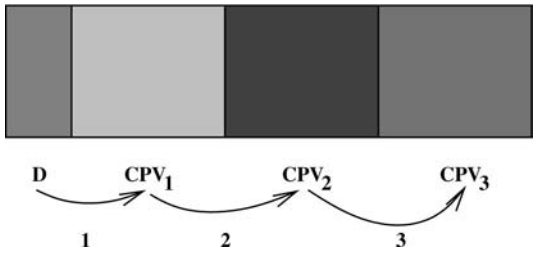


Figure 8.1 – Structure générale des codes BRADLEY

figure 8.1 et peut être résumée ainsi :

- une procédure de déchiffrement D (le *décrypteur*), dont le rôle est de collecter les données environnementales d'activation, de les tester et finalement de déchiffrer les différentes parties chiffrées du code ;
- une première partie de code chiffrée EVP_1 dépendant d'une clef de chiffrement K_1. Une fois déchiffrée, cette partie du code installe les fonctionnalités anti-antivirales à la fois passives et actives ;

[15] Article 323 du code pénal.

- une seconde partie chiffrée EVP_2 dépendant d'une clef de chiffrement K_2. Cette partie contient les fonctions d'infection et de propagation, ainsi que celles de polymorphisme/métamorphisme. Les codes BRADLEY sont des codes métamorphes, ce qui assure que la procédure D change également avec chaque génération de code ;
- une troisième partie chiffrée EVP_3 (optionnelle) dépendant d'une clef de chiffrement K_3. Elle contient la charge finale (dans nos codes-tests, il s'agit de l'affichage d'une simple fenêtre signalant la présence du virus et renseignant sur son activité dans le système, et ce à des fins de contrôle).

Décrivons à présent le protocole de génération/gestion environnementale de clefs. Les données d'activation, c'est-à-dire les données requises pour construire les différentes clefs, sont, pour la variante A :

- l'adresse DNS locale (par exemple, `@macompagnie.com`), notée α ;
- l'heure système courante (limitée aux heures `HH`) et la date système courante (`MMDD`), l'ensemble étant noté δ ;
- une donnée particulière présente dans le ou les systèmes cibles, que nous noterons ι. Dans notre cas, il s'agit d'un fichier particulier donné, que l'on sait avec certitude être présent dans les machines visées (parce qu'il aura été envoyé préalablement à l'attaque proprement dite, par exemple) ;
- une information particulière sous le contrôle exclusif de l'auteur de l'attaque, située hors du système attaqué mais accessible au code malveillant via un canal public. Dans notre cas, il s'agit d'une simple page web contenant une donnée particulière (information distante) dont la présence est limitée dans le temps et liée à la valeur δ, précédemment définie). Nous nommerons cette valeur particulière π, obtenue à partir du hachage de l'information distante[16].

Pour la variante B, la donnée ι est représentée par une clef publique présente dans un fichier *pubring.gpg* (cas de notre version). Ainsi, le virus va cibler un utilisateur $\mathcal{U}$ donné ou un groupe limité d'utilisateur communiquant par le biais de PGP avec $\mathcal{U}$. De nombreuses autres possibilités existent.

Les codes que nous avons conçus utilisent la fonction de hachage SHA-1 [98] – notée H – comme fonction à sens unique. Le protocole de génération et de gestion environnementales de clef proprement dit est alors le suivant :

1. la procédure de déchiffrement D collecte les données d'activation soit

[16] Le lecteur pourra objecter que la présence dans la procédure D d'une URL peut fournir des informations précieuses à l'analyste. Cela ne pose en réalité aucun problème. La page web utilisée est sous le contrôle de l'attaquant et la présence de la donnée d'activation est limitée dans le temps, ce qui rend l'utilité de cette information assez douteuse. Néanmoins, d'autres approches ont été testées. Plutôt que de considérer une unique valeur π, nous avons considéré deux données externes d'activation π et π'. Chacune d'entre elles provient d'une page web différente. L'URL de la seconde est chiffrée à l'aide d'une clef dont la construction environnementale dépend des valeurs α, δ, ι et π. Une fois déchiffrée, le virus récupère la seconde valeur d'activation π' ainsi qu'une permutation secrète P (localisée au début de EVP_1). Finalement, la clef K_1' est construite à partir de $P(\alpha), P(\delta), P(\iota)$ et de π' selon le même mécanisme que celui utilisé pour la clef K_1. Dans cette approche, la clef K_1 est remplacée, pour la suite, par la clef K_1'. Il est bien évident que cette approche peut être généralisée à davantage de pages web.

directement (α, δ et π) soit de manière répétitive[17] pour la valeur ι. Il calcule ensuite une valeur V de 160 bits donnée par $\mathcal{H}(\mathcal{H}(\alpha \oplus \delta \oplus \iota \oplus \pi) \oplus \nu)$ où ν représente les 512 premiers bits de EVP_1 (forme chiffrée) ;

2. si $V = M$ où M est la valeur d'activation contenue dans le code viral, alors $K_1 = \mathcal{H}(\alpha \oplus \delta \oplus \iota \oplus \pi)$, autrement la procédure de déchiffrement s'arrête et désinfecte le virus du système ;
3. D déchiffre EVP_1 en $\mathrm{VP}_1 = D_{K_1}(\mathrm{EVP}_1)$ et l'exécute. Ensuite, D calcule $K_2 = \mathcal{H}(K_1 \oplus \nu_2)$, où ν_2 représente les 512 derniers bits de VP_1 ;
4. D déchiffre EVP_2 en $\mathrm{VP}_2 = D_{K_2}(\mathrm{EVP}_2)$ et l'exécute. Ensuite, D calcule $K_3 = \mathcal{H}(K_1 \oplus K_2 \oplus \nu_3)$ avec ν_3 représentant les 512 derniers bits de VP_2 ;
5. D déchiffre EVP_3 en $\mathrm{VP}_3 = D_{K_3}(\mathrm{EVP}_3)$ et l'exécute ;
6. une fois l'action du virus terminée, ce dernier se désinfecte totalement du système et de manière sécurisée afin d'interdire une récupération des données.

Faisons quelques remarques concernant ce protocole :

- plusieurs procédures de chiffrement D_i, une partie chiffrée, peuvent être considérées. La procédure D_0 est en clair et gère le déchiffrement de la partie EVP_1, laquelle contient la procédure de déchiffrement de EVP_2. Plus généralement, la partie EVP_i contient la procédure D_i assurant le déchiffrement de la partie EVP_{i+1}. Chaque clef fait alors l'objet d'une gestion séparée de celle des autres clefs, conditionnée par une synchronisation temporelle. L'intérêt de cette méthode est de limiter l'impact d'une compromission éventuelle d'une clef tout en renforçant la sécurité globale du code ;
- de réplication en réplication, la totalité du code (procédure D et valeur M incluses) a été modifiée (processus de mutation métamorphe). Cela implique de la part de l'auteur du code une maîtrise et un contrôle total des fonctionnalités de polymorphisme/métamorphisme en liaison avec le protocole de génération et de gestion environnementales de clef (évolution des données d'activation δ et/ou π) ;
- les valeurs ν_i ont pour rôle de faire en sorte que les données en entrée de la fonction de hachage ne soient pas restreintes à un sous-ensemble réduit de valeurs, ce qui autoriserait, potentiellement, une recherche exhaustive sur ce sous-ensemble ;
- les différentes parties VP_i sont compressées avant chiffrement ;
- les clefs K_1, K_2 et K_3 peuvent être rendues indépendantes en utilisant un nombre plus important de données environnementales ;
- l'autodésinfection du code peut être temporisée pour une gestion plus souple des valeurs de temps et de date. La procédure D reste alors résidente. Cette variation est paramétrable de sorte à minimiser les impacts négatifs sur la sécurité générale du code ;

[17] Le virus scanne tout le système, selon une dynamique complètement paramétrable, à la recherche de la donnée. Dans notre cas – un fichier donné –, le virus recherche récursivement cette donnée dans l'arborescence.

- les ressources de chiffrement, hachage, compression peuvent être celles du système cible et non être incluses dans le code. Il est alors nécessaire de gérer au niveau de la valeur ι.

Pour toutes les variantes testées, les systèmes de chiffrement utilisés étaient RC4 [110] et RC6 [111].

8.4.3 Analyse du code viral et cryptanalyse

Pour évaluer la complexité de l'analyse des codes BRADLEY, deux situations doivent être envisagées :

- l'analyste dispose du code binaire du virus ;
- l'analyste n'en dispose pas.

La seconde situation est la plus probable dans le cas des codes BRADLEY, si l'on considère que le virus s'autodésinfecte afin de limiter sa présence dans le système attaqué et de ne pas laisser de trace analysable (code inclus).

Supposons toutefois que l'analyste soit dans la première situation et qu'il dispose du code binaire de l'une des versions du code. Le lecteur pourra en effet objecter que des techniques de *rootkit* (voir section 7) ou de *pot de miel* peuvent être utilisées pour capturer une telle copie et ensuite l'analyser pour étudier le protocole de génération et de gestion environnementales de clefs. Dans notre modèle, toutefois, cette solution reste impossible à mettre en œuvre d'un point de vue pratique et opérationnel : les codes mis au point interviennent dans des attaques ciblées (virus espions, virus à « frappe chirurgicale »...). Pour capturer au moins une copie d'un tel code avec une probabilité significative, il serait nécessaire de déployer sur toute la planète des millions de machines équipées de *rootkit* ou de pots de miel. Cette approche, si elle est efficace contre de grandes épidémies, se révèle, dans le cadre d'attaques de plus faible envergure, voire dans des attaques ciblées et localisées, totalement inefficace.

Montrons maintenant que le protocole de génération et de gestion environnementales de clefs, présenté dans la section 8.4.2 interdit opérationnellement l'analyse de code à moins de pouvoir résoudre un problème de cryptanalyse de complexité exponentielle.

Proposition 8.3 *L'analyse d'un code protégé par le protocole de génération environnementale de clefs présenté dans la section 8.4.2 est un problème qui a une complexité exponentielle.*

Démontrons cette proposition.

Preuve.
Tout d'abord, remarquons que la procédure D révèle naturellement (puisqu'elle est en clair) les informations suivantes :

- la valeur d'activation V ;
- le fait que le virus récupère l'heure et la date système ;
- le fait que le virus recherche les données α, ι et π.

En outre, l'analyste peut analyser le code viral si et seulement s'il connaît la clef secrète K_1. Ce paramètre peut être obtenu soit au moyen d'une cryptanalyse directe soit en essayant de « deviner » les différentes données d'activation requises pour générer la bonne clef. Ce dernier cas correspond à une attaque par dictionnaire.

L'approche cryptanalytique vise à retrouver la clef K_1 telle que $\mathcal{H}(K_1 \oplus C_1) = M$ où M et C_1 sont des valeurs facilement identifiables dans la procédure de déchiffrement D. Une fonction de hachage est fortement non injective par essence. Il est donc impossible – et mathématiquement cela n'a aucun sens – d'inverser une telle fonction (résistance au calcul d'une pré-image).

En conséquence, le problème doit être reformulé en un problème de recherche de collisions – pour plus de détails sur ce type d'attaque, consulter [92, chapitre 9]). En d'autres termes, il s'agit de trouver toutes les paires x et x', en entrée de H telles que $\mathcal{H}(x) = \mathcal{H}(x')$. Ce problème est calculatoirement impossible à résoudre pour une fonction de hachage solide comme SHA-1[18]. Plus précisément, trouver une telle paire requiert $2^{\frac{n}{2}}$ opérations pour une fonction de hachage opérant sur des blocs de n bits ($n = 512$ pour SHA-1). Comme l'analyste doit absolument déterminer la valeur exacte de la clef K_1 (celle qui a été utilisée pour chiffrer le code), il doit calculer toutes les valeurs x telles que $\mathcal{H}(x) = M$.

Pour une fonction de hachage travaillant sur des entrées de n bits et produisant des hachés de m bits, il existe en moyenne 2^{n-m} telles valeurs x (2^{352} pour SHA-1). Cela signifie que pour retrouver K_1, il faut $2^{\frac{n}{2}} \times 2^{n-m}$ opérations, soit $2^{\frac{3n-2m}{2}}$ opérations ($\approx 2^{608}$ pour SHA-1).

Considérons à présent l'attaque par dictionnaire. Elle consiste à énumérer toutes les valeurs possibles qui peuvent avoir été utilisées comme données d'activation. Notons que, dans ce cas particulier, l'analyste doit simultanément considérer à la fois le code viral chiffré et le système dans lequel ce dernier a été trouvé. L'analyste doit essayer toutes les valeurs possibles liées au système (c'est-à-dire α, ι et δ) sur lequel il a le contrôle durant l'analyse. Malheureusement la valeur π demeure hors de ce contrôle, il ne pourra donc déterminer sa valeur exacte. Il ne lui reste alors que l'approche exhaustive consistant à essayer toutes les valeurs $\alpha \oplus \delta \oplus \iota \oplus \pi$ possibles. Comme la valeur π sera choisie aléatoirement par l'attaquant, cette recherche exhaustive a une complexité en 2^n si la fonction de hachage travaille sur des blocs d'entrée de n bits (2^{512} pour SHA-1).

En conclusion, toutes approches confondues, la complexité résultante pour l'analyse de code est en $\min(2^n, 2^{\frac{3n-2m}{2}}) = 2^n$. ■

8.4.4 Scénarii divers à bases de codes Bradley

La technique de blindage de type BRADLEY a permis de valider le concept en lui-même. Nous allons montrer, à travers quelques scénarii testés et éprouvés

[18] Les attaques récentes contre certaines fonctions de hachage [140] ne remettent pas en cause la sécurité de notre protocole, comme le prouve la seconde partie de la démonstration.

en laboratoire – parmi de nombreux autres possibles –, comment un attaquant pourrait mettre en œuvre des codes totalement blindés de manière opérationnelle. La philosophie générale consiste à rendre la clef de chiffrement inaccessible à l'analyste tout en permettant un déchiffrement du code dans des conditions fixées par l'attaquant. Le lecteur doit conserver à l'esprit que toute analyse se fait *a posteriori*, c'est-à-dire qu'il faut récupérer un code et le transférer sur une machine dédiée à l'analyse, ce qui introduit un retard. Une analyse *in vivo* – directement sur la machine infectée – suppose l'existence d'un soupçon préalable concernant une éventuelle infection. En outre, cette solution est dangereuse car aucune information ne permet de garantir que l'action du code ne portera pas atteinte à la machine infectée.

Il est bien sûr possible de combiner ces différents schémas.

Blindage à double clef

Ce scénario a pour objectif de manipuler l'analyste en lui cachant la présence d'un blindage total et en lui faisant croire qu'il s'agit d'un blindage de faible niveau de sécurité. Cette manipulation, en créant un sentiment de victoire sur le code, permet de mieux assurer la protection du code réellement mis en œuvre lors de l'attaque.

Le chiffrement utilisé est un système de chiffrement par flot [41] dit *à clef une fois*. L'origine de ce chiffrement provient du système de Vernam [138].

Définition 8.5 *Le* chiffrement de Vernam *est un système de chiffrement par flot sur l'alphabet* $\mathcal{A} = \{0,1\}$. *Un message binaire* $m = m_1 m_2 m_3 \dots m_t$ *est combiné avec une suite-clef binaire* $k = k_1 k_2 k_3 \dots k_t$ *de même longueur pour produire une suite chiffrée* $c = c_1 c_2 c_3 \dots c_t$ *où*

$$c_i = m_i \oplus k_i, \quad 1 \leq i \leq t$$

Quand la suite-clef est générée indépendamment et aléatoirement, on parle alors de chiffrement par *bande aléatoire une fois (One time pad)*. Ce type de chiffrement a été prouvé inconditionnellement sûr[19] contre toute attaque, même à chiffré seul [123]. Plus précisément, si M, C et K sont des variables aléatoires désignant respectivement le texte clair, le texte chiffré et la suite clef (secrète), alors

$$H(M) = H(M|C)$$

où $H(.)$ désigne la fonction entropie[20]. De façon équivalente, cela revient à dire que la transinformation (ou information mutuelle) est nulle. Autrement dit, la connaissance du chiffré ne fournit aucune information sur le clair.

[19] Une façon d'expliquer la portée de ce résultat consiste à dire qu'il est impossible pour un attaquant de déterminer la clef utilisée et le clair envoyé même s'il disposait d'une puissance de calcul infinie et de l'éternité pour la mettre en œuvre. D'où le terme d'« inconditionnellement sûr ».

[20] Rappelons que pour une variable aléatoire discrète X prenant les valeurs $X_1, X_2, \dots, X_n$ avec les probabilités non nulles $p_1, p_2, \dots, p_n$, $H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$.

Shannon a prouvé [123] qu'une condition de sécurité parfaite était que

$$H(K) \geq H(M).$$

Cela signifie que l'incertitude sur la clef secrète doit être au moins aussi grande que celle sur le clair. Si la clef est de longueur k et si les bits sont choisis aléatoirement et indépendamment les uns des autres, alors $H(K) = k$ et la condition de Shannon devient $k \geq H(M)$. La bande aléatoire une fois est inconditionnellement sûre, quelle que soit la distribution statistique des bits de clair.

Remarquons enfin que si l'on ajoute une suite aléatoire différente k' au message clair m, nous obtenons un chiffré différent. Afin de mieux mesurer la portée de cette propriété dans le contexte du blindage, considérons l'exemple suivant.

Exemple 8.1 *Soit le texte chiffré suivant (octets exprimés en hexadécimal) :*

$$M = 2D\ 6B\ 1B\ 32\ 8E\ A6\ 25\ 73\ 69\ 61\ FF\ 2E\ 51\ AE\ 93\ B4$$

Considérons la première suite chiffrante suivante :

$$K_1 = 7D\ 39\ 54\ 75\ DC\ E7\ 68\ 3E\ 2C\ 4C\ BE\ 60\ 1E\ EA\ DA\ FA$$

qui, une fois additionnée modulo 2 bit à bit, donne le clair M_1 suivant :

$$M_1 = P\ R\ O\ G\ R\ A\ M\ M\ E\ -\ A\ N\ O\ D\ I\ N.$$

Si l'on considère alors la suite chiffrante K_2 suivante :

$$K_2 = 6E\ 24\ 5F\ 77\ D1\ EB\ 64\ 3F\ 3F\ 24\ B6\ 62\ 1D\ EF\ DD\ E0,$$

nous obtenons le clair M_2 donné par :

$$M_2 = CODE\_MALVEILLANT.$$

Le scénario proposé est alors le suivant :

- le code blindé collecte des données d'activation ;
- si ces données correspondent aux conditions réelles, désirées par l'attaquant, lesquelles sont impossibles en pratique à réaliser par l'analyste, alors une clef K_1 lui est fournie qui, additionnée au code chiffré, produit le code viral réel ;
- au contraire, si les conditions ne sont pas réunies, une clef K_2 est délivrée, qui permet de produire seulement un code viral en apparence de faible niveau de conception.

Dans ce scénario, l'objectif est de manipuler l'analyste et de lui faire croire que le code produit est le bon. L'antivirus sera mis à jour sur une base fausse. Le code réel restera protégé.

D'un point de vue pratique, la clef peut être générée de plusieurs manières :

- la suite chiffrante est la clef, les données environnementales sont des conditions de temps (voir section 8.4.4) données ;
- la suite chiffrante est produite à partir de valeurs de hachage concaténées et obtenues à partir de données environnementales dont l'une est de type π. Les valeurs différentes de cette dernière détermineront la valeur K_1 ou K_2. Cela implique un précalcul pour déterminer les valeurs respectives que doit prendre π. Un exemple de tel précalcul est donné dans [6].

Utilisation de clef usb

Le premier scénario consiste à activer le code au moment de la connexion d'une clef USB. Le scénario s'articule alors de la manière suivante :

- le code est déployé préalablement à l'attaque. Ce dernier est résident en mémoire, dans le système cible ;
- le code surveille la présence d'une clef USB active qu'il doit identifier dans un premier temps (présence d'un fichier donné, valeur particulière présente dans les secteurs initiaux, numéro de série de la clef[21]...) ;
- en cas de clef valide, le code récupère les données d'activation situées à un emplacement précis. Par exemple, ces données peuvent être constituées du nom du i-ième fichier présent à la racine de la clef, de n octets situés dans ce fichier entre les positions j_1 et j_2 ;
- le code efface sur la clef, de manière sécurisée, tout ou partie des données d'activation.

L'attaquant activera le code par une simple connexion d'une clef USB. Cette clef peut lui appartenir (cas d'un personnel extérieur en visite) ou appartenir à la victime elle-même, qui n'a pas conscience du rôle que va jouer sa clef. Dans ce dernier cas, l'attaquant aura fait en sorte que les données d'activation soient présentes sur la clef (un dossier client, un tarif...).

Du point de vue de l'analyste, les données d'activation ne sont disponibles que sur la clef, dans des conditions non reproductibles lors de l'analyse. En termes de politique de sécurité, cela implique une gestion draconienne des périphériques amovibles et une isolation des réseaux sensibles. Cette attaque peut être réalisée avec une clef USB ou tout autre périphérique équivalent : appareil photo numérique, PDA, téléphone portable, *pocketPC*, imprimante...

Synchronisation de temps

Ce scénario utilise une synchronisation fine entre le code viral blindé et le « serveur » de données d'activation, autrement dit le site sur lequel figure la valeur π. La philosophie de cette approche est la suivante : comme la capture éventuelle d'un code et son étude par l'analyste ont toujours un temps de retard, par définition, toute action de sa part introduira un décalage dans le

[21] Il est nécessaire de vérifier que les données choisies, notamment celles liées à la clef elle-même [91], ne se retrouvent pas sur le disque dur.

synchronisme mis en place entre le code et son « serveur ». Le scénario s'articule ainsi :

1. à l'installation dans la machine, le code se connecte une première fois avec le serveur pour initialiser une base temps de référence commune. Notons t_0 cette base temps. Ce temps de référence se généralise en considérant une fonction strictement croissante f_t qui accepte en argument la date D_v et le temps T_v de la machine cible, ainsi que les mêmes valeurs pour le serveur, soient D_s et T_s. En outre, le code blindé utilise également une valeur d'identification V_i spécifique à la version du virus[22] interdisant le rejeu (voir point suivant). Ainsi, nous avons $t_0 = f_t(D_v, T_v, D_s, T_s, V_i)$. La fonction f_t est localisée seulement sur le serveur et de fait est inaccessible à l'analyste ;
2. à la connexion initiale, le serveur vérifie si la valeur V_i est rejouée ou non. Si c'est le cas, le serveur décide que le code est une version en cours d'analyse. Soit il ne répond plus et cette souche (forme originale et mutations) est définitivement révoquée, soit il leurre l'analyste en lui renvoyant une clef correspondant à une version « anodine » du code (voir section 8.4.4) ;
3. le code se reconnecte exactement à $t_0 + t_\delta$ pour avoir la clef. Le serveur vérifie si $t_0 + t_\delta > \epsilon$ pour un ϵ donné connu du seul serveur. Si le code est en cours d'analyse, avec une probabilité proche de 1, on aura effectivement $t_0 + t_\delta > \epsilon$, l'analyste ayant forcément introduit un retard par le seul fait du traitement préparatoire (transfert sur une machine d'étude par exemple) ;
4. le serveur à l'émission du virus vers la machine (phase initiale d'infection) calcule sa propre base temps $t_{-\delta}$. Lors du calcul de la valeur t_0, à l'aide des éléments fournis par le code dès son arrivée dans la machine, le serveur vérifie si $t_0 + t_{-\delta} > \epsilon'$, pour une valeur ϵ' donnée. Si c'est le cas, le serveur suppose que le code a été intercepté lors de la transmission du code.

Le lecteur remarquera que tout retard, provoqué par l'analyste ou par une perturbation du trafic, fera échouer l'attaque. En revanche, le code restera protégé contre l'analyste. Rappelons là un principe fondamental en virologie : un nécessaire compromis existe toujours entre la virulence et la furtivité. Un code plus efficace sera moins discret et inversement. C'est le principe des attaques ciblées. Mais dans ce contexte précis, le caractère dangereux des attaques compense très largement leur (relative) portée limitée.

Chaîne d'infection

Ce scénario permet d'utiliser le blindage total dans le cadre d'une attaque classique de type ver informatique mais à la virulence limitée. Bien que les

[22] Cette valeur est gérée par le moteur de polymorphisme inclus dans le virus. Le serveur dispose de la fonction de mutation de V_i. Cette dernière est capable de gérer simultanément et de manière discriminante un grand nombre de codes blindés.

techniques précédemment décrites permettent de déployer un ver classique, le cas d'une attaque ciblée a été privilégié. Cela passe par une organisation de cette attaque « en étoile » : un serveur pilotant plusieurs copies. Le principal souci avec cette approche réside dans le fait que si le serveur est compromis ou non fonctionnel, l'attaque est elle-même vouée à l'échec.

Dans ce nouveau scénario, nous considérerons une approche décentralisée. L'objectif est de faire en sorte que le « serveur » délivrant les données d'activation change en permanence, compliquant encore plus la tâche de l'analyste. Cependant, des précautions particulières doivent être prises par l'attaquant pour gérer le cas d'un serveur compromis ou accessible à l'analyste. Les grandes étapes sont les suivantes, en décrivant le mécanisme pour une machine :

1. le ver blindé infecte une machine cible C_i. Son code contient la valeur d'activation M_i et une adresse IP_{i-1} où se connecter, par exemple pour jouer le scénario décrit dans la section 8.4.4. L'adresse IP_{i-1} en question est celle d'une machine C_{i-1} déjà infectée d'où provient la copie virale (mutée) en cours d'action. La connexion de C_{i-1} vers C_i utilise une adresse IP spoofée afin d'interdire une remontée d'infection – cas du ver *W32/Blaster* [40]). Le ver récupère sur C_{i-1} la donnée d'activation I_i et l'efface de manière sécurisée. L'analyse de la machine C_{i-1} ne permettra pas de la récupérer ;
2. le ver agit sur C_i et produit ensuite une version mutée de lui-même destinée à infecter une machine cible C_{i+1} ;
3. l'infection de la machine C_{i+1} est réalisée ;
4. le ver génère la valeur d'activation I_{i+1} et la stocke dans la machine C_i. Il est essentiel que la copie mutée destinée à la machine C_{i+1} et sa valeur d'activation correspondante I_{i+1} ne soient pas simultanément présentes dans la machine C_i ;
5. le ver se désinfecte de la machine C_i, de manière sécurisée, de sorte à ce qu'il ne subsiste plus aucune trace de l'infection (hors effets de la charge finale).

La propagation d'un tel code sera bien évidemment plus lente. Cependant, ce scénario de blindage permet de compenser une cinétique de propagation plus faible par une plus grande (voire totale) sécurité du code.

8.5 La technique Aycock *et al.*

John Aycock, Rennie de Graaf et Michael Jacobson Jr., de l'université de Calgary au Canada, ont proposé une méthode de blindage [6] quelque peu différente mais inspirée par la techniques des codes BRADLEY. Leur méthode réalise en fait de la génération dynamique de code. Leur objectif est de ne pas stocker le code directement sous forme chiffrée mais d'en stocker une représentation particulière. En fait, comme nous allons le montrer, la représentation utilisée dans leur méthode est quasi-identique à un chiffrement et donc susceptible

d'être détectée. Alors que la technique BRADLEY peut être mise en œuvre en assurant un aspect TRANSEC (voir section 7.4) par l'utilisation de système de chiffrement simulant les tests statistiques (voir chapitres 3 et 6), et ainsi rester indétectable, la technique Aycock n'offre quant à elle aucun aspect TRANSEC.

8.5.1 Le principe

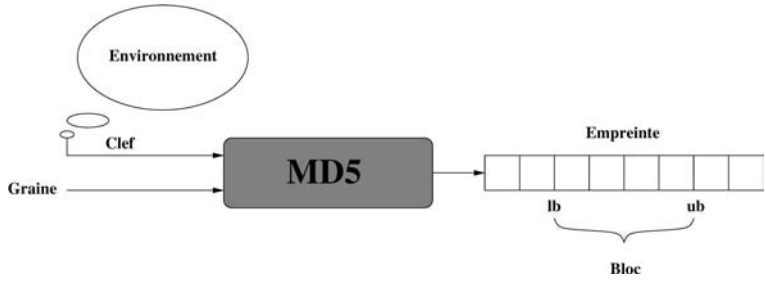


Figure 8.2 – Principe de la technique Aycock *et al.* (fonction MD5)

Ce principe consiste à combiner une clef avec une graine aléatoire et ensuite à en calculer l'empreinte numérique par une fonction de hachage. Cette empreinte est interprétée comme une suite d'octets, de laquelle est extraite une sous-séquence, située dans l'empreinte entre deux indices lb et ub, elle-même interprétée comme une suite d'instructions machine (opcodes). Cette sous-séquence est appelée *bloc d'exécution*. La graine aléatoire est choisie par le programmeur de sorte à ce qu'apparaisse, dans l'empreinte numérique produite, un bloc d'exécution donné lorsque la bonne clef est considérée. Ce processus est itéré pour tous les blocs d'exécution qui doivent être « blindés ».

La technique de blindage ainsi définie est illustrée par la figure 8.2 et son utilisation est schématisée par la figure 8.3. Chaque bloc d'exécution b_i est donc remplacé par les données suivantes, obtenues à l'aide d'une clef secrète K et d'une graine s_i (voir plus loin) :

- la graine aléatoire s_i ;
- deux valeurs d'indices lb_i et ub_i.

Chaque bloc d'exécution b_i est alors produit seulement au moment où le programme blindé en a besoin, de la manière suivante :

- calcul de $M = H(K||s_i)$ ($||$ désigne l'opérateur de concaténation) ;
- le bloc d'exécution b_i est la sous-séquence de M située entre les octets lb_i et ub_i.

La fonction de hachage est considérée comme publique. Tout le travail de blindage consiste à trouver, pour chaque bloc d'exécution b_i à protéger contre le désassemblage, la graine s_i et les valeurs lb_i et ub_i. L'analyste du code blindé, quant à lui, doit retrouver les blocs b_i réellement exécutés. Comme dans la technique BRADLEY, il doit retrouver la clef secrète K exacte utilisée, pour

tous les blocs b_i, et non une valeur provoquant une collision. Si une mauvaise clef est choisie par l'analyste, les blocs d'exécution produits seront alors faux et aucune analyse ne sera possible.

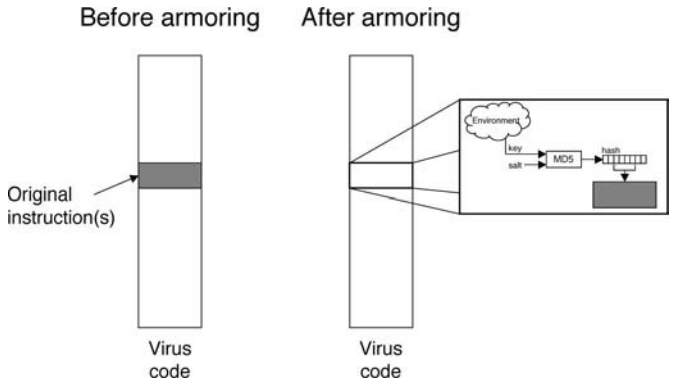


Figure 8.3 – Principe d'utilisation de la technique Aycock *et al.*

L'étape de recherche des données de blindage requiert des calculs qui peuvent être longs. Si cela n'est pas un désavantage dans le cas d'un code malveillant simple – le travail est effectué préalablement et une fois pour toutes –, c'est en revanche un inconvénient rédhibitoire pour des codes polymorphes ou métamorphes. C'est là une faiblesse conceptuelle majeure vis-à-vis de la technique BRADLEY. Un code malveillant ne pourra en effet jamais recalculer les valeurs de graines s_i et lb_i, ub_i pour blinder une variante polymorphe, avec un temps et une puissance de calcul limités.

Illustrons la recherche de ces données par un exemple tiré de [6].

Exemple 8.2 *Considérons le bloc d'exécution (en hexadécimal) suivant*

$$e9\ 74\ 56\ 34\ 12.$$

Ce bloc correspond, sur un Intel x86 à un saut à l'adresse 12345678_{16}*, en supposant que l'instruction de saut démarre à l'adresse* 0[23].

La valeur $s_i = 0x07e9717a09$ *concaténée avec la clef* $K = aycock$ *(en hexadécimal) en entrée de la fonction de hachage SHA-1, l'empreinte numérique est*

$$ef\ 6d\ f4\ ed\ 3b\ a1\ ba\ 66\ 27\ fe$$
$$e9\ 74\ 56\ 34\ 12\ a2\ d0\ 4f\ 48\ 91.$$

[23] Une autre difficulté qui limite singulièrement la portée pratique de la technique Aycock *et al.* tient à la grande variabilité des adressages qui obligerait un code lors de sa propagation ou de son installation à recalculer un nombre non négligeable de blocs d'exécution.

Fonction de Hachage	*Clef*	*Graine*	*# graines testées*	*Temps de calcul (sec.)*
MD5 (128 bits)	aycock	55b7d9ea16	96915712675	80262
	degraaf	a1ddfc1910	68082987191	58356
	foo	e6500e0214	84599106230	73206
	jacobs	9ac1848109	40201885669	34557
	ucalgary.ca	4d21abe205	24899771642	23059
	Average		62939892681	53888
SHA-1 (160 bits)	aycock	07e9717a09	40084590622	38795
	degraaf	0d928a260e	59834611693	57907
	foo	2bc680de1e	130536957733	125537
	jacobs	ca638d5e06	26937346972	26314
	ucalgary.ca	585cc614	344525998	339
	Average		51547606603	49778
SHA-256 (256 bits)	aycock	7cad4d4807	30796664539	46360
	degraaf	dd72e2380a	43225788191	64625
	foo	c17a8c3629	174262804678	260641
	jacobs	effa7fc07	33787185089	51744
	ucalgary.ca	48343fa40f	66147214782	101823
	Average		69643931455	105039

Table 8.1 – Recherche par force brute des graines s_i pour un bloc d'exécution donné i de cinq octets

En numérotant les octets de cette empreinte à partir de 0, le run d'exécution commence à l'indice $lb = 10$ *et se termine à l'indice* $ub = 14$.

Le précalcul consiste, pour chaque bloc d'exécution i , à mener une recherche exhaustive sur les valeurs de graine s_i et à retenir celles qui produisent une empreinte numérique contenant le bloc i. Le tableau tbl :reljmp donne les résultats d'une telle recherche pour les trois principales fonctions de hachage communément utilisées. Le lecteur remarquera que le temps de calcul pour un bloc d'exécution est loin d'être négligeable. Les auteurs de cette méthode de blindage ont proposé de paralléliser le calcul de la graine au moyen de *botnets*. Outre une puissance de calcul permettant de gérer des valeurs plus importantes, cela limite les possibilités, déjà réduites, de tout analyste [6].

8.5.2 Analyse de la méthode

Plaçons-nous à présent du côté de l'analyste qui doit analyser un code protégé par la technique Aycock *et al.* Il dispose des informations suivantes, contenues dans le code :

- la ou les fonctions de hachage utilisées ;
- les valeurs de graines s_i ;

- les valeurs d'indices lb_i et ub_i ;
- éventuellement une information sur l'ensemble d'appartenance de la clef secrète K.

En revanche, deux informations capitales, sur lesquelles reposent la sécurité du blindage, sont et doivent rester inconnues de l'analyste, à savoir :

1. la clef secrète K. Sa taille doit interdire toute approche de recherche exhaustive. Le résultat de la proposition 8.3 (cas des codes BRADLEY) s'applique ;
2. les blocs d'exécution. Ils doivent être assez nombreux à être protégés. Il faut en effet que l'analyste ne puisse pas deviner par une simple analyse du flot d'exécution la valeur de blocs d'exécution protégés çà et là. La situation idéale est de blinder la totalité du code mais la phase de précalcul sera beaucoup plus longue.

Ces considérations montrent que, comme la technique BRADLEY, la technique Aycock *et al.* offre une sécurité intéressante en terme de blindage de code. Les contraintes sont cependant plus importantes. Toutefois, cette technique est facilement détectable et un simple filtrage permettra de contrer la propagation ou l'échange de codes ainsi blindés. En effet, si la technique Aycock *et al.*, offre un aspect COMSEC, elle n'offre aucun aspect TRANSEC (voir la section 7.4). Expliquons pourquoi.

Pour les fonctions de hachage usuelles, dont la taille d'empreinte est de 128, 160 ou 256 bits (voir tableau 8.1), les valeurs d'indices possibles sont donc limitées pour les valeurs lb_i (sur la base d'un bloc d'exécution de cinq octets) à respectivement 123, 155 et 251, tandis que celles des valeurs ub_i ne dépassent pas 127, 159 et 255 respectivement. Cela signifie que dans chaque version blindée du code, la répartition statistique des valeurs lb_i et ub_i sera suffisamment caractéristique pour permettre une identification certaine et par là même de bloquer tout code ayant ces caractéristiques (voir exercice en fin de chapitre). Une solution consiterait d'une part à utiliser des fonctions de hachage ayant une taille d'empreinte plus grande et d'autre part à considérer des tailles de blocs d'exécution plus petites. Le prix à payer serait alors une complexité calculatoire plus importante pour la phase de blindage.

Si la technique Aycock *et al.* est conceptuellement intéressante, elle est difficilement applicable en pratique. En outre, elle n'est pas viable pour des codes polymorphes/métamorphes ou qui doivent s'adapter à l'environnement cible (recalcul d'adresses par exemple). Insistons sur le fait que, comme pour la technique BRADLEY dans sa version de base, la gestion de la clef est un point critique. Nous allons voir comment cette gestion peut être rendue beaucoup plus simple.

8.6 Obfuscation et blindage probabilistes

Dans la section 8.2.3, une variété particulière de technique d'obfuscation a été définie : la τ-obfuscation. L'intérêt principal de la τ-obfuscation réside

dans le fait que celui qui analyse le code d'une part et les programmes obfuscateurs/desobfuscateurs d'autre part peut être confronté à des contraintes susceptibles de grandement différer. Afin de préciser les choses, considérons la τ-obfuscation dans un contexte viral. Un code malveillant – typiquement un ver ou un cheval de Troie – peut parfaitement réaliser son action en un temps qui serait totalement inacceptable pour un logiciel d'analyse ou de sécurité comme un antivirus. D'un point de vue pratique, en effet, quel utilisateur accepterait que son antivirus monopolise les ressources de son ordinateur pendant plusieurs minutes, voire dizaines de minutes à des fins d'analyse antivirales[24] ?

En revanche, un code malveillant, lui, peut parfaitement « accepter » de finaliser son action (propagation, charge offensive...) après un temps plus ou moins long mais néanmoins significatif. En conséquence, la τ-obfuscation repose sur deux défis :

- protéger le code contre l'analyse, c'est le but de l'obfuscation elle-même ;
- rendre le processus de désobfuscation suffisamment long pour empêcher, dans un contexte malveillant, un antivirus (ou tout autre programme d'analyse) de désobfusquer et d'identifier un code malveillant en un temps inférieur ou égal à τ.

Le premier objectif peut être réalisé indépendamment du second. En effet, en aucune manière, il n'est requis qu'un code malveillant se désobfusque lui-même – pour agir – en moins de temps que ne le ferait un antivirus à des fins d'analyse. Aussi, plutôt que d'imposer à un tel code de travailler en un temps très court, pourquoi ne pas imaginer un code qui n'a aucune information sur la manière de se désobfusquer lui-même ? Ce code devra alors essayer un certain nombre de méthodes à cette fin, ce qui est susceptible de nécessiter en moyenne un temps important (par exemple, quelques dizaines de minutes). Dans ce cas, l'antivirus, lui, sera condamné à passer un temps au moins équivalent pour la désobfuscation et l'analyse du code malveillant. Comme ce dernier est polymorphe/métamorphe, cette analyse doit être reconduite à chaque accès au fichier. Cette contrainte est commercialement inacceptable. L'attaquant a gagné.

Mentionnons en outre le fait que Zuo et Zhou [150] ont présenté des résultats nouveaux concernant la complexité en temps des virus informatiques (temps d'exécution du virus, temps de la détection). Leurs principaux résultats sont les suivants :

- pour chaque catégorie de virus, il existe un virus v dont la complexité en temps de la procédure d'infection est aussi grande que l'on veut ;
- pour chaque catégorie de virus, il existe un virus v dont l'implémentation de la procédure d'infection possède une complexité en temps aussi grande que l'on veut.

[24] Rappelons que la protection la plus couramment utilisée est celle dite « à l'accès » (moniteur temps réel) : par exemple à l'ouverture d'une pièce jointe d'un courrier électronique ou à l'exécution d'un programme. Alors qu'un utilisateur accepte plus ou moins facilement le moindre ralentissement ou interférence par son antivirus, il est certain qu'il changera d'antivirus si ce dernier ralentit la machine par ses analyses.

Le concept de τ-obfuscation s'inscrit directement dans la perspective générale de ces résultats.

Notons que la stratégie de lutte consistant à surveiller les temps d'exécution et à instaurer une limite est inapplicable. Elle incriminerait à tort un grand nombre de logiciels légitimes. Enfin, une fois un code lancé, si l'antivirus a renoncé à son analyse et n'a finalement émis aucune alerte, le code pourra ultérieurement agir en toute impunité.

8.6.1 Une approche unifiée de la protection de code

L'obfuscation et le blindage sont les deux techniques principales de protection de code mais elles partagent autant de points communs que de points sur lesquelles elles diffèrent. C'est la raison pour laquelle nous allons maintenant considérer une approche plus générale qui permettra de prendre en compte toutes les techniques possibles de protection de code : blindage léger[25], polymorphisme, métamophisme, obfuscation... En d'autres termes, le but est d'envisager une approche unifiée de toutes ces techniques.

Considérons un ensemble $\mathcal{S}$ de méthodes de protection (obfuscation, chiffrement, blindages...) qui peuvent ou non dépendre d'une ou plusieurs informations secrètes (typiquement une clef). Alors pour tout $s \in \mathcal{S}$ et pour tout programme P, $s(P)$ est un code qui ne peut être analysé, sauf en y consacrant une quantité suffisante de temps τ. Considérons à présent l'ensemble $\mathcal{S}'$ de méthodes définies de la manière suivante : $\forall s \in \mathcal{S}, \exists s' \in \mathcal{S}'$ telle que $(s' \circ s)(P) \equiv P$, où $\circ$ décrit la composition fonctionnelle[26] (non nécessairement commutative). En d'autres termes, l'ensemble $\mathcal{S}'$ représente l'ensemble des méthodes de déprotection relativement à $\mathcal{S}$. Il est essentiel de noter que $|\mathcal{S}|$ et $|\mathcal{S}'|$ peuvent ne pas être égaux. Comme les techniques de protection de codes doivent être nécessairement réversibles, nous considérerons que $|\mathcal{S}| \leq |\mathcal{S}'|$. Ainsi, pour $s \in \mathcal{S}$, il peut exister $\mathcal{S}'' \subset \mathcal{S}'$ tel que $\forall s'' \in \mathcal{S}'', (s'' \circ s)(P) \equiv P$. Considérons la notation suivante :

$$\mathcal{S}'_s = \{s' \in \mathcal{S}' | (s' \circ s)(P) \equiv P\}.$$

Considérons maintenant un programme malveillant P mettant en œuvre des techniques de protection de code regroupées dans une procédure $\mathcal{C}$. Cette routine choisit aléatoirement une méthode $s \in \mathcal{S}$ et calcule $s(P)$, une version protégée de P. Ce formalisme permet, entre autres choses, de distinguer les techniques polymorphes pour lesquelles nous avons $P \cap s(P) = \mathcal{C}$, des techniques métamorphes où $P \cap s(P) = \emptyset$ – voir [150] pour un autre formalisme. Chaque

[25] Le blindage dit léger n'a pour but que de retarder l'analyse automatique de code (par exemple par un antivirus) et non pas de l'interdire définitivement comme dans le cas du blindage total (techniques Bradley).

[26] Notons que pour certaines techniques, comme le chiffrement, nous pouvons avoir

$$(s' \circ s)(P) = P.$$

fois que le code $s(P)$ est exécuté, il se « déprotège » lui-même en appliquant au hasard ou de manière itérative, une méthode $s' \in \mathcal{S}'$ à $s(P)$.

Pour un $s \in \mathcal{S}$ fixé, la complexité de cette déprotection dépend de manière évidente de la probabilité $p_{s'}$ de trouver un $s' \in \mathcal{S}'$ tel que $(s' \circ s)(P) \equiv P$. Nous avons alors

$$p_{s'} = \frac{|\mathcal{S}'_s|}{|\mathcal{S}'|},$$

et la recherche d'une méthode de déprotection adéquate, relativement à un $s \in \mathcal{S}$ fixé, a une complexité en $\mathcal{O}(\frac{1}{p_{s'}})$. Bien sûr, le calcul de cette complexité et de la probabilité correspondante se fait sous l'hypothèse que les méthodes de déprotection sont uniformément et identiquement distribuées dans l'ensemble $\mathcal{S}'$. Cela constitue un premier critère de mise en œuvre pratique de ce formalisme.

Comme aucune condition supplémentaire n'est imposée sur les éléments de $\mathcal{S}$, aucune différence n'est faite sur la nature même des techniques de protection utilisées (blindage, obfuscation, chiffrement...). Nous ne parlerons plus que de protection au sens général du terme. Cette vision unifiée permet de se concentrer uniquement sur les propriétés mathématiques générales ou sur les structures que les ensembles $\mathcal{S}$ et $\mathcal{S}'$ doivent présenter plutôt que sur les propriétés individuelles de leurs éléments.

D'un point de vue pratique, une manière relativement simple d'implémenter ce modèle consiste à choisir une méthode de protection au hasard dans un ensemble (éventuellement infini) de méthodes de protection. La déprotection d'un code protégé nécessite alors d'essayer aléatoirement ou exhaustivement les méthodes inverses. Nous allons à présent décrire deux implémentations de protection de code, selon le modèle unifié. Ces méthodes sont tirées de [10]. Sans restriction conceptuelle, nous supposerons qu'une partie seulement d'un programme P, jugée critique, est protégée. D'un point de vue général, la validation d'une méthode de déprotection peut être réalisée de beaucoup de façons différentes. La plus intuitive et la plus facile consiste à utiliser une fonction de hachage H [48]. Pour chaque méthode de déprotection s' appliquée à $s(S)$, la valeur $H((s' \circ s)(P))$ est calculée et comparée à une valeur de validation stockée à un endroit donné du code.

8.6.2 Chiffrement probabiliste

Supposons que nous souhaitions chiffrer une partie du code. Cela peut être, par exemple, dans le cas du blindage. Si nous n'imposons aucune contrainte sur le temps de déchiffrement mais si nous voulons que tout analyste du code (et, a fortiori, tout logiciel d'analyse comme un antivirus) ne puisse pas le décrypter sinon en un temps minimum donné, alors nous pouvons par exemple choisir une clef de chiffrement aléatoire possédant une entropie minimale fixée n. Lorsque le code doit être déchiffré, le code doit alors tester toutes les clefs de n bits, et donc l'analyste aussi. La seule différence tient au fait que pour l'analyste cela peut nécessiter un temps de recherche important.

Dans le cas général décrit, la complexité moyenne est en $\mathcal{O}(2^n)$. Il est essentiel de noter que l'algorithme de chiffrement doit être choisi avec soin de sorte qu'aucune attaque, en vertu de faiblesses de conception, ne permette de retrouver la clef plus efficacement que par une recherche exhaustive.

Afin de comprendre ce point fondamental, prenons le cas d'un chiffrement par substitution simple. Cela consiste à choisir un alphabet parmi les $27! \approx 2^{93}$ permutations possibles de l'alphabet ordonné usuel. Ici, l'entropie est importante ($n = 93$). Pourtant, un tel système se cryptanalyse en quelques secondes. L'entropie de la clef n'est donc pas une donnée suffisante. Le niveau de sécurité de l'algorithme est également essentiel. Toutefois, rappelons que l'objectif est de retarder une analyse automatique (blindage léger) et non de rendre le code inexpugnable (blindage total).

Considérons, parmi de nombreux autres, plusieurs scénarii.

Scenarii probabilistes simples

Comme première solution, nous considérons une clef de chiffrement aléatoire. Le déchiffrement consistera ensuite à tester successivement toutes les clefs possibles[27]. Si nous supposons que tester une clef prend une micro-seconde (10^{-6}), alors il faudra au code, pour se déchiffrer, légèrement plus d'une heure pour trouver une clef de 32 bits. Dans ces conditions, aucun antivirus ne peut consacrer un temps aussi long à l'analyse d'un code malveillant. Ce scénario n'est pourtant pas optimal. En effet, il est nécessaire de mettre des contraintes, en terme d'entropie, sur la clef, laquelle ne doit pas être trop importante.

Une autre solution consiste à stocker dans le code protégé une version permutée de la clef. Toutefois, à nouveau, l'ensemble des permutations utilisées ne peut être trop important. La déprotection doit nécessiter un temps assez conséquent, mais qu'il est nécessaire de maîtriser tout en ne laissant comme seule solution que la recherche exhaustive. Il est alors indispensable de restreindre l'ensemble des permutations possibles à un sous-ensemble particulier, de taille plus maîtrisable. Nous allons voir comment faire dans la section 8.6.3.

Une autre scénario intéressant consiste à cacher la clef secrète dans les données chiffrées. Une fois que ces dernières ont été produites, la clef est divisée en plusieurs segments, lesquels sont disséminés dans le code chiffré. Dans ce but, nous pouvons choisir des périodes aléatoires pour réaliser cette dissimulation : soit une première période p, la clef est répartie sur tous les p octets, et ce jusqu'à la fin du code. Si des fragments de clefs restent à dissimuler, une seconde période p' est choisie et on continue. Lors de la phase de déprotection (déchiffrement) du code, il faut retrouver la ou les périodes utilisées, p et p', de manière aléatoire. Le nombre de périodes est aléatoire, mais il est cependant limité par la taille de la clef, laquelle peut être stockée en clair dans le code chiffré. L'intérêt de cette approche est qu'elle ne nécessite plus de contrainte sur la taille de la clef, cette dernière pouvant être très importante – une clef privée de 2 048 bits, par exemple.

[27] Cette approche a été utilisée par le virus *Win32/IHSix* de manière assez triviale.

Toutefois, cette dernière méthode ne cache pas encore parfaitement la présence d'un code malveillant – bien qu'aucun antivirus ne soit capable d'identifier un tel code. La présence d'une quantité non négligeable de code chiffré dans le programme peut déclencher la suspicion d'un antivirus[28]. Ce problème peut être facilement résolu en considérant des techniques de chiffrement mettant en œuvre la simulabilité des tests statistiques (chapitre 3) et en particulier des techniques de contournement des tests de détection fondée sur l'entropie (voir le chapitre 6).

Un scénario plus sophistiqué

Pour cette seconde approche, la partie du code qui doit être protégée est chiffrée à l'aide d'un système à clef publique. Un couple de clefs publique/privée (K_{pr}, K_{pu}) est aléatoirement généré avant la phase de chiffrement[29]. La clef K_{pr} est détruite et le chiffrement est réalisé. La clef publique K_{pu} est stockée dans la procédure de déprotection. Pour déchiffrer le code, la clef privée K_{pr} doit initialement être retrouvée par la procédure de déprotection. Dans ce but, plutôt que de tester toutes les clefs K_{pr} possibles – ce qui est calculatoirement inenvisageable, même pour un code malveillant, si on souhaite conserver une taille minimale à la clef –, la procédure va retrouver directement cette clef privée à partir de la clef publique K_{pu}.

Considérons l'exemple suivant :

- nous utilisons le chiffrement à clef publique probabiliste. La raison essentielle motivant le choix de ce type de système tient au fait qu'il est sémantiquement sûr [61,93]. En d'autres termes, ces systèmes représentent une version polynomialement bornée de la notion de secret parfait[30] tel que défini par Claude Shannon [124]. Le chiffrement probabiliste offre une bien plus grande sécurité que son homoloque déterministe (le chiffrement RSA par exemple). Il met en œuvre de la randomisation pour garantir un haut niveau de sécurité démontrable. Ainsi, un attaquant passif disposant de ressources de calcul polynomialement bornées – c'est précisément le cas pour tout programme d'analyse automatique comme les antivirus ou les IDS – ne peut obtenir aucune information concernant le texte clair à partir du texte chiffré. Dans notre exemple, nous utiliserons le chiffrement probabiliste de Goldwasser-Micali [61] dont la sécurité sémantique est conditionnée par la difficulté de résolution du problème de résiduosité quadratique[31] ;

[28] Notons que la détection de codes malveillants utilisant la présence de données chiffrées est susceptible de conduire à un nombre élevé de fausses alarmes. Les codes protégés par des *packers*, dans un but de protection de *copyright*, seraient détectés comme malveillants.

[29] Cela prend un temps négligeable avec une bonne implémentation en multi-précision, de l'algorithme de Miller-Rabin [92, p. 139].

[30] Pour expliquer ce qu'est le secret parfait (réalisé par le chiffrement de Vernam, par exemple), il suffit de savoir que même disposant de l'éternité et d'une puissance de calcul infinie – conditions on ne peut plus favorables –, l'attaquant sera incapable de déterminer quelle clef et donc quel texte clair ont été choisis par l'émetteur.

[31] Soit un nombre composite (c'est-à-dire non premier) n et un nombre a dont le symbole de

- nous ne décrirons pas en détail le procédé de chiffrement lui-même mais juste la partie consacrée à la génération et à la gestion de clefs. Avant la phase de protection (chiffrement) du code, la procédure de protection doit générer un couple de clefs publique/privée (K_{pr}, K_{pu}). L'algorithme général est le suivant :
  1. choisir deux nombres premiers p et q aléatoires et de grand taille, tous deux ayant à peu près la même taille (utilisation de l'algorithme de Miller-Rabin [92, pp. 146]) ;
  2. calculer $n = pq$;
  3. choisir un élément $y \in \mathbb{Z}_n$ tel que y n'est pas un résidu quadratique modulo n et tel que le symbole de Jacobi $(\frac{y}{n})$ vaut 1 ;
  4. la clef publique K_{pu} est constituée de la paire (n, y) et la clef privée K_{pr} est le couple (p, q).
- la clef K_{pr} est détruite. La routine de protection chiffre le code par chiffrement probabiliste, à l'aide de la clef K_{pu}. La clef K_{pr} est détruite.

La sécurité du chiffrement probabiliste de Goldwasser-Micali repose sur la difficulté de résolution du problème de résiduosité quadratique (QRP). Le probème QRP est aussi difficile à résoudre que celui de la factorisation. En conséquence, et avec les considérations précédentes, la procédure de déprotection doit factoriser le nombre n. Pour cela, il existe quelques algorithmes efficaces [83, 95], dont la meilleure complexité est en $\exp(c \log \alpha.n.(\log(\log(n))^{1-\alpha})) = L_n[\alpha, c]$ où α et c sont des constantes fixées (pour les méthodes les plus efficaces, $\alpha = \frac{1}{3}$ et $1,5 < c < 2$). Toutefois, la plupart de ces méthodes exigent une quantité de mémoire rédhibitoire pour toute utilisation par un logiciel antivirus. De plus, des contraintes mathématique et algorithmiques sont également à considérer pour parvenir à un résultat presque opérationnel.

C'est la raison pour laquelle la procédure de déprotection – que ce soit celle du code malveillant ou celle de l'antivirus – ne peut utiliser ces algorithmes de factorisation. Le meilleur compromis peut être réalisé avec la méthode du crible quadratique (QS) [102]. Notons $L_n[\alpha] = \exp(\sqrt{\log n \log \log n})^{\alpha + \mathcal{O}(1)}$. Alors, la complexité en temps de méthode QS pour factoriser un entier n est $L_n[1]$. La complexité mémoire est quant à elle en $L_n[\frac{1}{2}]$. Ainsi, le choix de la clef publique K_{pu} aura un impact direct sur la méthode de déprotection. Des simulations ont montré qu'une clef de 50 digits pouvait être factorisée en environ 30 minutes avec moins de 100 Ko de mémoire.

Remarque Le chiffrement par clef publique peut également être considéré pour protéger une clef secrète K utilisée dans un chiffrement de type symétrique (autrement dit, une clef de session comme celle utilisée dans PGP). En accord avec ce qui a été dit dans la section 8.6.2, l'intérêt est que la valeur $E_{K_{pu}}(K)$ dissimule l'entropie exacte de K.

Jacobi vaut 1. Le problème de la résiduosité quadratique consiste à déterminer s'il existe ou non un nombre x tel que a est le reste de la division euclidienne de x^2 par n. Ce problème est d'une complexité équivalente à celle du problème de la factorisation.

8.6.3 Transformation de données

Considérons à présent un ensemble de méthodes opérant des transformations sur les données d'un code malveillant (en incluant le code proprement dit et les données). Protéger le code par obfuscation signifie appliquer une de ces méthodes au code exécutable, tandis que la désobfuscation consiste à trouver et à appliquer la transformation inverse. Soit $\mathcal{S}$ cet ensemble de méthodes par transformation. Parmi de nombreuses approches possibles, considérons que $\mathcal{S}$ a une structure de groupe pour la loi de composition fonctionnelle. Alors pour toute transformation $\pi \in \mathcal{S}$, son inverse π^{-1} est aussi dans $\mathcal{S}$, de même que toute composée de transformations de $\mathcal{S}$.

Le groupe symétrique σ_n – autrement dit, le groupe des permutations sur l'ensemble $\{1, \ldots, n\}$ – est l'exemple de référence pour décrire un tel ensemble de transformations. Le principal intérêt de considérer des transformations par permutation de code tient au fait que si du code chiffré par des méthodes classiques de subsitution possède un profil statistique très caractéristique, ce n'est pas le cas des données permutées. Alors qu'un logiciel de détection ou d'analyse sera susceptible de déclencher une alerte en cas de contenu chiffré[32], cela ne sera généralement pas le cas pour du code permuté.

L'obfuscation du code sera assurée en permutant les octets du programme. Nous considérons un *sous-groupe cyclique* G de σ_n, pour restreindre la taille de l'ensemble de référence (en effet $|\sigma_n| = n!$). Ce sous-groupe présente d'intéressantes propriétés mathématiques. En particulier, pour une permutation $\pi \in \sigma_n$, la permutation inverse est π^k, pour une valeur $k \in [1..n]$. Ainsi, la procédure de protection tire une permutation π au hasard alors que la routine de déprotection choisit aléatoirement une permutation π' et cherche de manière exhaustive la bonne valeur k' telle que $(\pi')^{k'} \circ \pi = Id$. Dans ce cas, comme la complexité calculatoire de la déprotection est en $\mathcal{O}(n)$, nous devons choisir des valeurs importantes pour la valeur n afin que le temps de déprotection soit suffisamment long pour contrarier un logiciel d'analyse.

D'un point de vue algorithmique, la permutation π peut être un générateur du sous-groupe cyclique G de σ_n. Trouver la permutation inverse est alors facile à implémenter au moyen d'une simple boucle allant de 1 à l'ordre de G. De plus, l'ordre de G sera choisi en premier afin de maximiser le nombre de générateurs de G.

Dans le cas précédent, comme toute permutation possède un unique inverse, l'ensemble $\mathcal{S}'$ est un singleton. Mais on peut souhaiter que pour une méthode de protection donnée $s \in \mathcal{S}$, on ait $|\mathcal{S}'_s| > 1$. Cela autorise plus de flexibilité pour l'implémentation de la protection/déprotection de code. Une solution – il en existe de très nombreuses autres – est de considérer un ensemble ordonné donné $\mathcal{F}$ d'octets (non nécessairement contigus) $\{b_{i_1}, b_{i_2}, \ldots, b_{i_k}\}$ et un ensemble d'indices dans le code $\{i_1, i_2, \ldots, i_k\}$. Chaque fois qu'une tentative de déprotection d'un programme P (application d'une permutation que l'on

[32] Certains fournisseurs d'accès Internet bloquent par exemple les pièces jointes chiffrées des courriers électroniques.

espère être la permutation inverse) réalise

$$\forall i \in \{i_1, i_2, \ldots, i_k\}, \quad \mathcal{P}[i] = b_i,$$

alors le code est considéré comme déprotégé[33]. Dans ce cas, l'ensemble $\mathcal{F}$ décrit la part critique du programme devant être systématiquement protégée tandis que le reste du code peut contenir du code mort ou factice. Considérons la permutation suivante sur 9 octets : $\pi = (3\ 8\ 4\ 7\ 2\ 9\ 1\ 5\ 6)$. Si $\mathcal{F}$ contient les octets initiaux localisés aux indices $\{2, 5, 8\}$ alors $\mathcal{S}' = \{\pi^3, \pi^6, \pi^9\}$.

D'un point de vue général, il est possible de partitionner le code en k ensembles différents. Nous choisissons $k - j$ ensembles pour décrire la part critique du code à protéger tandis que les j ensembles restants constituent la part non critique du code, laquelle peut également être formée de code factice. Nous considérons alors k groupes cycliques (de permutations) différents, chacun d'entre eux agissant sur l'une des k parties du code. Cette approche nous permet de considérer des groupes de permutations de bien plus grande taille. En effet, l'ordre d'un sous-groupe cyclique a le même ordre de grandeur que la longueur d'une permutation. En d'autres termes, si l'on considère un unique sous-groupe cyclique d'ordre n, tout antivirus devra essayer au plus $\mathcal{O}(n)$ permutations avant de trouver la permutation inverse recherchée. Au contraire, si nous considérons k groupes cycliques différents, chacun d'entre eux d'ordre $n_i, \quad i = 0, \ldots, k-1$, la complexité augmente significativement pour atteindre $\mathcal{O}(\mathrm{lcm}(n_0, n_2, \ldots, n_{k-1}))$ avec un maximum si les différents n_i sont premiers entre eux.

Il est bien sûr évident qu'une implémentation efficace considérera à la fois des valeurs k aléatoires, pour chaque étape de protection/déprotection, ainsi que des n_i co-premiers aléatoires. De plus, afin d'interdire à l'analyste (humain ou logiciel) de deviner quelle est la portion critique de code, et ainsi de pouvoir restreindre la recherche de la permutation inverse à cette seule portion critique, une implémentation efficace de ce schéma de protection agira ainsi : la partie critique du code permutée – notons-la $\sigma(\mathcal{C})$ – contient la longueur du code sur laquelle appliquer la fonction de hachage de validation. Si le candidat pour la permutation inverse est incorrect, alors $(\sigma' \circ \sigma)(\mathcal{C})$ contiendra une valeur incorrecte ν, aléatoire, de cette longueur, égarant ainsi l'analyse. Comme ν est aléatoire, il sera impossible d'extraire une quelconque information sur la longueur exacte et ainsi de restreindre la recherche de la permutation inverse à un sous-ensemble plus petit.

8.7 Conclusion

La preuve de concept BRADLEY a été conçue pour illustrer le fait qu'un blindage total était possible et pour sensibiliser au danger de codes ainsi protégés. D'autres techniques (techniques Aycock, protection probabiliste...) existent. Il

[33] Rappelons que la validation s'effectue via la valeur de hachage de ces données et non sur les données elles-mêmes.

est à peu près certain que ce type de protection signifiera, dans un avenir proche, un danger impossible à gérer a posteriori, si ce n'est déjà le cas[34]. Combinée avec les techniques de polymorphisme/métamorphisme qui contribueront, lorsque efficacement mises en œuvre, à empêcher la première détection, la technique de blindage offre un degré supplémentaire de sécurité au cas où malgré tout le code viendrait à être capturé. Les différentes expériences ont largement confirmé cet état de fait. En outre, l'utilisation de virus k-aires (voir chapitre 4) permettra de multiplier à l'infini les possibilités techniques du blindage total.

À moins d'une surveillance permanente et directe de tout réseau, grâce au déploiement d'un vaste réseau de pots de miel, et en équipant toutes les machines de *rootkit* de surveillance, il n'existe aucune protection contre ce genre de codes. Et encore, cette hypothèse est pour le moins irréaliste, tant les ressources à engager sont immenses en termes de coût et d'organisation.

La technique de blindage total de code présentée dans ce chapitre, ainsi que les différents scenarii proposés, montrent clairement qu'il est nécessaire de faire en sorte qu'aucun code de ce type ne parvienne jamais dans un système : il sera alors trop tard et aucune approche technique ne saurait être *a posteriori* efficace pour gérer ce genre d'attaques. Une politique de sécurité draconnienne doit prévenir toute infection par un tel code. Une entreprise qui en serait victime pourrait alors être l'objet d'un vol de données, d'une destruction de ses informations les plus critiques, d'une attaque à la disponibilité de ses propres systèmes sans qu'il soit possible d'en expliquer la cause et surtout sans pouvoir mettre en œuvre une protection et des contre-mesures.

Enfin, la technique de blindage de code total montre, encore une fois, que la maîtrise de la cryptographie – cette fois-ci par l'attaquant – est une garantie de sécurité. Toute expertise antivirale se doit d'inclure des compétences importantes dans ce domaine.

Exercices

1. Programmez la technique de blindage imaginée par Aycock *et al.* (section 8.5), pour chacune des fonctions de hachage présentée dans le tableau 8.1. Vous étudierez ensuite le profil statistique des codes ainsi blindés et imaginerez un ou plusieurs critères de détection et d'identification de cette technique de blindage. Concluez.
2. Programmez la technique de protection de code par algorithme de chiffrement probabiliste à clef publique décrite dans la section 8.6.2. Étudiez ensuite les performances de la procédure de déprotection en fonction de la taille de la clef publique choisie.

[34] L'analyse d'une attaque réelle par l'auteur en 2005 laisse à penser que ce genre d'attaques est déjà d'actualité. Par chance, dans le cas évoqué, et qui concernait un système sensible, si le chiffrement était plus évolué que ce que l'on peut rencontrer jusqu'à présent, la gestion de la clef présentait encore des faiblesses qui ont permis l'analyse. Mais ce cas n'était plus très éloigné du cas idéal, opérationnellement parlant, du blindage total. Précisons qu'un an après les faits, le code en question n'est toujours pas détecté par les antivirus !

3. Implémentez la technique décrite dans le dernier paragraphe de la section 8.6.3. Étudier le comportement statistique des valeurs ν. Expliquez pourquoi il est impossible d'utiliser ce comportement pour extraire une information sur la valeur ν correcte.

Conclusion

Chapitre 9

Conclusion

Cet ouvrage est à présent terminé. L'essentiel a été dit et démontré, même si bien d'autres aspects auraient pu être abordés : quoique importants, les présenter aurait été irresponsable et probablement illégal.

Mais pour le lecteur, les choses ne font que commencer. L'espoir en écrivant ce livre est que tout professionnel impliqué, d'une manière ou d'une autre, dans la sécurité des systèmes d'information et de communication soit convaincu que les solutions face aux attaques ciblées, réalisées ou à venir, qu'il est possible de mener, ne sont plus techniques et ne le seront jamais. Elles ne peuvent jouer qu'un rôle marginal de confort de la pensée : un antivirus, protection indispensable il faut le rappeler, suscite avant tout le sentiment de sécurité plus qu'il ne protège réellement. Un logiciel antivirus, comme un pare-feu ou autre logiciel de sécurité, ne gèrera jamais que les attaques « grand public », connues et d'un niveau de sophistication relativement limité.

L'attaquant peut toujours se placer dans un contexte « non conventionnel », au mépris de toute règle ou de tout académisme, et surtout se situer hors d'action de la défense. Il suffit de considérer pour lui des instances de problèmes généraux, de complexité suffisante. L'attaquant peut également changer le référentiel de temps concernant son action, de manière à créer une distorsion avec celui du défenseur qui, lui, subit nécessairement celui qui lui est imposé par le système.

Comme illustré par la figure 9.1, cette différence de référentiel se résume à deux points :

- la propagation d'un code, de système en système, est de nos jours fulgurante. Il faut à peine quelques minutes à un ver pour infecter un réseau planétaire [7]. La mise à jour de nos antivirus n'est possible qu'au minimum vingt-quatre heures, dans le meilleur des cas, après l'apparition d'un code *« in the wild »*, et seulement si ce code existe en un nombre suffisant de copies pour permettre une première analyse. Dans le cas d'une attaque ciblée – le cauchemar d'une entreprise –, aucune mise à jour ne sera possible ;

- en terme d'exécution, au sein d'un système cette fois, un code malveillant peut diluer son action dans le temps contrairement à un antivirus qui doit agir rapidement s'il veut rester commercialement viable.

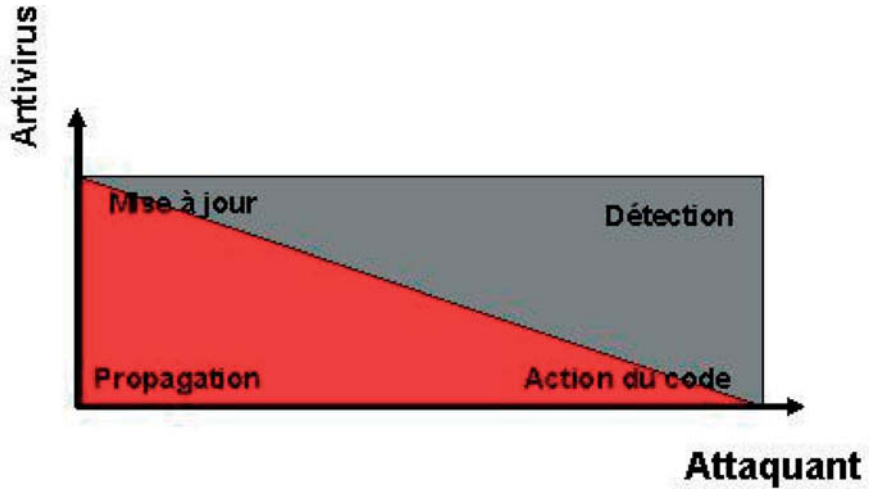


Figure 9.1 – Référentiels attaquant-lutte antivirale

Tout cela illustre le fait que la solution ne peut être technique et que seule une protection fondée sur une politique de sécurite rigoureuse, proactive, reprenant plus ou moins fortement le modèle isolationniste, est viable. Trois aspects, au moins, peuvent être mis en relief :

- la prévention est fondamentale et elle doit s'accompagner d'une véritable politique de veille technologique en amont, et d'un plan de formation et de sensibilisation des utilisateurs – la meilleure arme de l'attaquant sera toujours l'utilisateur et pour reprendre une expression désormais célèbre : « Il n'existe pas de correctif pour la bêtise et le manque de conscience professionnelle. » Une fois qu'un code sophistiqué est dans un système, nous avons montré dans cet ouvrage qu'il était trop tard ;
- de nouvelles dispositions techniques doivent être imaginées pour la détection. Elles doivent de plus devenir extérieures au système, ne serait-ce que pour espérer traiter le cas de certains *rootkits*. De nouveaux modèles et outils doivent également être imaginés et mis au point. La recherche, en virologie informatique, tant théorique que technique, devient une priorité. Le temps des chapelles est révolu. Il n'est plus admissible qu'une corporation décide de ce qu'il est bien de faire ou non et se réserve le droit de mener seule des études, lesquelles produisent des résultats qui au final sont difficilement vérifiables. Le retard accumulé est déjà trop important. Que des sociétés célèbres comme Microsoft participe à des

recherches [75], dans lesquelles l'écriture, l'étude et le test de codes malveillants font partie intégrante du protocole de travail est le signe que les temps changent ;
- enfin, il est devenu nécessaire d'intégrer dans toute politique de sécurité des processus de restauration rapide, pouvant éventuellement concerner une partie importante du réseau. La disponibilité des systèmes et la permanence du métier l'exige. Face à des codes à base de *rootkits* utilisant la virtualisation ou bien des codes k-aires comme le virus PARALLÈLE_4, la seule solution reste la réinstallation du système. Cela implique notamment une politique de sauvegarde rigoureuse – son absence est une grave lacune encore trop fréquemment constatée, notamment dans les entreprises.

L'autre constat est que la technologie évolue trop rapidement et de manière probablement trop dangereuse. La sophistication et l'ergonomie croissantes des systèmes profitent avant tout à l'attaquant. La seule technologie de virtualisation est un exemple dramatiquement patent. La question se pose de savoir si le gain espéré de cette évolution compensera suffisamment les risques croissants qu'elle fait courir à nos systèmes et donc à nos ressources. Il n'est pas sûr que la voix des « marchands du temple » soit la plus pertinente.

Annexes

Chapitre 10

Résultats d'analyse des antivirus

10.1 Les conditions de tests

Nous donnons ici quelques résultats de l'analyse en boîte noire des principaux antivirus, relativement à la famille *Bagle*. Les références des produits testés (version et bases de signatures) sont données dans le tableau 10.1. Toutes

Produits	Version	Base de signatures
Avast	4.6.691	0527-2
AVG	7.0.338	267.9.2/52
Bit Defender	7.2	16 sept. 2005
DrWeb	4.33.2.12231	25 fév. 2006 (104186)
eTrust	7.1.194	11.5.8400 (Vet)
eTrust	7.1.194	23.65.44 (InoculateIT)
F-Secure 2005	5.10-480	15 sept. 2005
G-Data	AVK 16.0.3	KAV-6.818/BD-16.864
KAV Pro	5.0.383	19 sept. 2005
McAfee 2006	-	DAT 4535
NOD 32	2.5	1.1189 (08/08/05)
Norton 2005	11.0.2.4	15 sept. 2005
Panda Titanium 2006	5.01.02	17 jan. 2006
Sophos	5.0.5 R2	3.98
Trend Office Scan	6.5 - 7.100	2.837.00

Table 10.1 – Antivirus analysés (version et base virale)

les variantes de la famille *I-Worm.Bagle* ont été dans un premier temps soumis à l'algorithme E-1, qui a été présenté dans la section 2.4.2. Nous avons repris

la dénomination virale (nommage) utilisée par le site VX Heavens [139], sur lequel les variantes utilisées ont été téléchargées. Dans tout ce qui suit, pour un code de taille n (octets), les octets sont numérotés de 0 à $n-1$.

Quand l'algorithme E-1 a échoué, l'algorithme E-2 a été utilisé, lors d'une seconde étape.

10.2 Résultats : algorithme d'extraction E-1

Pour des raisons de place, nous ne donnons les résultats que pour quelques variantes seulement. Les résultats détaillés sont disponibles en contactant l'auteur.

Les différents tableaux contiennent les indices des octets dans le code et non les octets eux-mêmes. Quand cela a été possible, le motif $\mathcal{S}_{\mathcal{M},\mathcal{M}}$ a été donné en entier. Dans le cas contraire, seuls les premiers indices ont été donnés afin d'illustrer les ressemblances existant entre les différents antivirus, esquissant ainsi une « phylogénie » des ces derniers.

Produit	Taille signature (en octets)	Signature (indices)
Avast	29	14,128 → 14,144, 14,146 → 14,157, 14,159
AVG	7,297	0 - 1 - 60 - 200 - 201 - 220 - 469...
Bit Defender	6	0 - 1 - 60 - 200 - 201 - 206
DrWeb	11	0 - 1 - 60 - 200 - 201 - 206 - 220 222 - 461 - 465 - 469
eTrust/Vet	3,700	0 - 1 - 60 - 200 - 201 - 206 - ...
eTrust/InoculateIT	3,700	0 - 1 - 60 - 200 - 201 - 206 - ...
F-Secure 2005	11	0 - 1 - 60 - 200 - 201 - 206 220 - 240 - 241 - 461 - 469
G-Data	6	0 - 1 - 60 - 200 - 201 - 206
KAV Pro	11	Identique à celle de *F-Secure 2005*
McAfee 2006	7	0 - 1 - 60 - 200 - 201 - 206 - 220
NOD 32	7,449	0 - 1 - 60 - 200 - 201 - 204 - 205...
Norton 2005	0	(pas une fonction ET)
Panda Tit. 2006	9	0 - 1 - 60 - 200 - 201 - 206 220 - 461 - 541
Sophos	28	0 - 1 - 60 - 200 - 201 - 206 - 220...
Trend Office Scan	7	0 - 1 - 60 - 200 - 201 - 206 - 220

Table 10.2 – Motif viral de *I-Worm.Bagle.A*

Produit	Taille signature (en octets)	Signature (indices)
Avast	9	8,162 - 8,166 - 8,170 - 8,173 - 8,175 8,180 - 8,181 - 8,187 - 8189
AVG	13,288	0 - 1 - 60 - 216 - 217 - 222 - 236...
Bit Defender	87	2,831 - 2,964 - ....
DrWeb	1,773	0 - 1 - 60 - 216 - 217 - 222 - 236...
eTrust/Vet	4,306	0 - 1 - 60 - 216 - 217 - 222 - ...
eTrust/InoculateIT	4,306	0 - 1 - 60 - 216 - 217 - 222 - ...
F-Secure 2005	105	0 - 1 - 60 - 216 - 217 - 222 - ...
G-Data	3	2831 - 2964 - 2965
KAV Pro	105	Identique à celle de *F-Secure 2005*
McAfee 2006	12,039	0 - 1 - 60 - 216 - 217 - 222 - ...
NOD 32	4,793	0 - 1 - 60 - 216 - 217 - 220 - 221...
Norton 2005	0	(pas une fonction ET)
Panda Tit. 2006	37	0 - 1 - 59 - 60 - 216 - 217 - 222...
Sophos	15,028	0 - 1 - 2 - 4 - 8 - 12 - 13 - 16 - 24...
Trend Office Scan	146	0 - 1 - 60 - 216 - 217 - 222 - ...

Table 10.3 – Motif viral de *I-Worm.Bagle.E*

Produit	Taille signature (en octets)	Signature (indices)
Avast	8	7,256 → 7,259 7,278 → 7,281
AVG	7,276	5739 - 5864 - 5866 - ...
Bit Defender	3,342	7,385 - 7,386 - ....
DrWeb	2,896	0 - 1 - 60 - 144 - 145 - 150 - 164...
eTrust/Vet	4,320	0 - 1 - 60 - 144 - 145 - 150 - 164...
eTrust/InoculateIT	1,311	0 - 1 - 60 - 144 - 145 - 150 - 164...
F-Secure 2005	3,128	0 - 1 - 60 - 144 - 145 - 150 - 164...
G-Data	2,954	0 - 1 - 60 - 144 - 145 - 150 - 445...
KAV Pro	3,128	Identique à celle de *F-Secure 2005*
McAfee 2006	8,084	0 - 1 - 60 - 144 - 145 - 150 - 164...
NOD 32	9,629	0 - 60 - 144 - 145 - 148 - 149 - ...
Norton 2005	8	0 - 1 - 60 - 144 - 145 150 - 164 - 445
Panda Tit. 2006	437	0 - 1 - 32 → 35 - 60 - 64...
Sophos	3,429	0 - 1 - 60 - 144 - 145 - 150 - 164...
Trend Office Scan	63	7,750 - ...

Table 10.4 – Motif viral de *I-Worm.Bagle.J*

Produit	Taille signature (en octets)	Signature (indices)
Avast	10	14,567 - 14,574 → 14,577 14,581 - 14,585 → 14,588
AVG	13,112	533 → 663 - 665 - ...
Bit Defender	6,093	0 - 1 - 60 - 128 - 129 - 134 - ...
DrWeb	6,707	0 - 1 - 60 - 128 - 129 - 132 - 133...
eTrust/Vet	4,343	0 - 1 - 60 - 128 - 129 - 134 - ...
eTrust/InoculateIT	1,276	0 - 1 - 60 - 128 - 129 - 134 - ...
F-Secure 2005	54	0 - 1 - 60 - 128 - 129 - 548 - ...
G-Data	53	0 - 1 - 60 - 128 - 129 - 548 - ...
KAV Pro	54	Identique à celle de *F-Secure 2005*
McAfee 2006	4,076	0 - 1 - 60 - 128 - 129 - 134 - ...
NOD 32	17,777	0 - 1 - 60 - 128 - 129 - 132 - 133...
Norton 2005	51	0 - 1 - 60 - 128 - 129 - 134 - 429 - ...
Panda Tit. 2006	1659	0 - 1 - 4 - 8 - 12 - 13 - 16 - 24...
Sophos	7,538	0 - 1 - 60 - 128 - 129 - 134 - 148...
Trend Office Scan	44	0 - 1 - 60 - 128 - 129 - ...

Table 10.5 – Motif viral de *I-Worm.Bagle.N*

Produit	Taille signature (en octets)	Signature (indices)
Avast	8	12,916 → 12,919 12,937 → 12,940
AVG	14,575	533 → 536 - 538 - ...
Bit Defender	8,330	0 - 1 - 60 - 128 - 129 - 134 - ...
DrWeb	6,169	0 - 1 - 60 - 128 - 129 - 134 - ...
eTrust/Vet	1,284	0 - 1 - 60 - 128 - 129 - 134 - ...
eTrust/InoculateIT	1,284	0 - 1 - 60 - 128 - 129 - 134 - ...
F-Secure 2005	59	0 - 1 - 60 - 128 - 129 - 546 - ...
G-Data	54	0 - 1 - 60 - 128 - 129 - 546 - ...
KAV Pro	59	Identique à celle de *F-Secure 2005*
McAfee 2006	12,1278	0 - 1 - 60 - 128 - 129 - 134 - ...
NOD 32	21,849	0 - 1 - 60 - 128 - 129 - 132 - 133 - ...
Norton 2005	6	0 - 1 - 60 - 128 - 129 - 134
Panda Tit. 2006	7,579	0 - 1 - 60 - 134 - 148 - 182 - 209...
Sophos	8,436	0 - 1 - 60 - 128 - 129 - 134 - 148...
Trend Office Scan	88	0 - 1 - 60 - 128 - 129 - ...

Table 10.6 – Motif viral de *I-Worm.Bagle.P*

10.3 Résultat d'extraction pour l'algorithme E-2

Là où l'algorithme d'extraction E-1 a échoué – cas où la fonction de détection n'est pas la fonction ET –, l'algorithme E-2 a systématiquement extrait le schéma de contournement complet $(\mathcal{S}_{\mathcal{M},\mathcal{M}}, \overline{f_{\mathcal{M}}})$. C'est, par exemple, le cas pour l'antivirus *Norton 2005*. Nous donnons ici le résultat pour la variante *I-Worm.Bagle.E.*

Afin de limiter le risque d'utilisations « délictuelles », nous ne donnons ici que la forme algébrique normale de $\overline{f_{\mathcal{M}}}$ avant l'étape LOGICALMINIMIZE. Il est cependant possible de remarquer que cette fonction ne dépend que d'un nombre limité d'octets. La DNF est très creuse (faible nombre de termes). Cela signifie que le plagiaire trouvera facilement le moyen de rendre le code indétectable pour l'analyse de forme. Il en a été de même pour les quelques autres cas rencontrés, pour lesquels, en particulier, $15 \leq |\mathcal{S}_{\mathcal{M},\mathcal{M}}| \leq 25$, selon les variantes.

L'algorithme d'extraction E-2, présenté en section 2.4.1, a produit la fonction de non détection suivante (avant l'étape LOGICALMINIMIZE) pour la variante *I-Worm.Bagle.E.* La signature ne dépend que de 15 octets :

$$\mathcal{S}_{\mathcal{M},\mathcal{M}} = \{0, 1, 60, 61, 62, 63, 216, 217, 222, 223, 236, 237, 645, 646, 647, 648\}.$$

Rappelons que la notation x_i signifie que l'octet i de la signature peut ne pas être modifié alors que $\overline{x_i}$ indique qu'il doit être modifié.

Les autres exemples sont disponibles auprès de l'auteur.

$$
\begin{aligned}
&\overline{f_{\text{I-Worm.Bagle.E}}} = \\
&\overline{x_0}.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &\overline{x_0}.\overline{x_1}.\overline{x_{60}}.\overline{x_{61}}.\overline{x_{62}}.\overline{x_{63}}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{646}.x_{646}.x_{647}.x_{648} \\
\vee\ &\overline{x_0}.\overline{x_1}.\overline{x_{60}}.\overline{x_{61}}.\overline{x_{62}}.\overline{x_{63}}.\overline{x_{216}}.\overline{x_{217}}.\overline{x_{222}}.\overline{x_{223}}.\overline{x_{236}}.\overline{x_{237}}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &\overline{x_0}.\overline{x_1}.\overline{x_{60}}.\overline{x_{61}}.\overline{x_{62}}.\overline{x_{63}}.\overline{x_{216}}.\overline{x_{217}}.\overline{x_{222}}.\overline{x_{223}}.\overline{x_{236}}.\overline{x_{237}}.\overline{x_{645}}.\overline{x_{646}}.\overline{x_{647}}.\overline{x_{648}} \\
\vee\ &x_0.\overline{x_1}.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.\overline{x_{60}}.\overline{x_{61}}.\overline{x_{62}}.\overline{x_{63}}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.\overline{x_{60}}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.\overline{x_{60}}.\overline{x_{61}}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.\overline{x_{60}}.\overline{x_{61}}.\overline{x_{62}}.\overline{x_{63}}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.\overline{x_{61}}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.\overline{x_{62}}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.\overline{x_{63}}.x_{216}.x_{217}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.\overline{x_{648}} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.\overline{x_{216}}.\overline{x_{217}}.\overline{x_{222}}.\overline{x_{223}}.\overline{x_{236}}.\overline{x_{237}}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.\overline{x_{216}}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.\overline{x_{216}}.\overline{x_{217}}.\overline{x_{222}}.\overline{x_{223}}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.\overline{x_{217}}.x_{222}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.\overline{x_{222}}.x_{223}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.\overline{x_{223}}.x_{236}.x_{237}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.\overline{x_{236}}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.\overline{x_{237}}.x_{645}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{237}.\overline{x_{645}}.\overline{x_{646}}.\overline{x_{647}} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{237}.\overline{x_{645}}.\overline{x_{646}}.\overline{x_{647}}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{237}.\overline{x_{645}}.x_{646}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{237}.x_{645}.\overline{x_{646}}.x_{647}.x_{648} \\
\vee\ &x_0.x_1.x_{60}.x_{61}.x_{62}.x_{63}.x_{216}.x_{217}.x_{222}.x_{223}.x_{237}.x_{645}.\overline{x_{647}}.x_{648}
\end{aligned}
$$

Bibliographie

[1] Abel R. J. R. (1996), Some new BIBDS with $\lambda = 1$ and $6 \leq k \leq 10$, *Journal of Combinatorial Designs*, Vol. 4, pp. 27-50.

[2] *Advanced Configuration and Power Interface Specification*, septembre 2004. Disponible sur `http://www.acpi.info/DOWNLOADS/ACPIspec30.pdf`

[3] Aho A. V., Hopcroft J. E. et Ulman J. D. (1974), *The Design and Analysis of Computer Algorithms*, Addison Wesley.

[4] AMD64 Architecture Programmer's Manual Vol. 2 : System Programming Rev 3.11. Disponible sur `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf`

[5] Angluin D. (1988), Queries and Concept Learning, *Machine Learning*, Vol. 2-4, pp. 319–342.

[6] Aycock J., de Graaf R. et Jacobson Jr M. (2006), Anti-disassembly using cryptographic hash functions, *Journal in Computer Virology*, EICAR Special Issue, 2 :(1), pp. 79–86.

[7] Balepin I. (2003), *Superworms and Cryptovirology : a Deadly Combination*, `http://wwwcsif.cs.ucdavis.edu/~balepin/new_pubs/worms-cryptovirology.pdf`

[8] Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vadhan S, Yang K. (2001), On the (im)possibility of obfuscation programs. In : *Advances in Cryptology, Crypto 2001*, LNCS 2139, pp. 1–18, Springer Verlag.

[9] Barwise J. (1983), *Handbook of Mathematical Logic*. North-Holland, ISBN 0-444-86388-5.

[10] Beaucamps P., Filiol E. (2006), On the possibility of practically obfuscating programs - Towards a unified perspective of code protection, *Journal in Computer Virology*, (2)-4, WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds.

[11] Beauchamp K. G. (1984), *Applications of Walsh and related functions*, Microelectronics and Signal Processing Series, Academic Press, ISBN 0-12-084180-0.

[12] Beth T., Jungnickel D. and Lenz H. (1999), *Design Theory*, Cambridge University Press, ISBN 0-5214-4432-2 (Vol. 1) and ISBN 0-5217-7231-1.

[13] Borovkov A. (1987), *Statistique mathématique*, Éditions Mir.

[14] Boyer R. S. et Moore J. S. (1977), A fast string-searching algorithm. *Communications of the ACM*, (20), 10, pp. 762–772.

[15] Brulez N., Raynal F. (2005), Le polymorphisme cryptographique : quand les opcodes se mettent à la chirurgie esthétique, *Journal de la sécurité informatique MISC*, 20, juillet 2005.

[16] Cachin C. (2004), An Information-theoretic Model for Steganography, *Information and Computation*, vol. 192, pp. 41–56.

[17] Cachin C. (2005), Digital Steganography. In : *Encyclopedia of Cryptography and Security*, Springer Verlag.

[18] Carton O. (2006), *Langages formels, calculabilité et complexité*, Cours de l'École Normale Supérieure. Disponible sur `http://www.jussieu.fr/~carton/Enseignement/Complexite/ENS/Support/`

[19] Chakrabarty K. et Hayes J. P. (1998), Balanced Boolean Functions. *IEE Proc.-Comput. Digit. Tech.*, Vol. 145, No. 1.

[20] Chausson C. (2006), L'anti-virus McAfee prend les fichiers Windows pour des virus, Le monde informatique, édition du 15 mars 2006. Publié sur `http://www.lemondeinformatique.fr`

[21] Chomsky N. (1956), Three models for the description of languages, *IRE Transactions on Information Theory*, 2, pp. 113–124.

[22] Chomsky N. (1969), On certain formal properties of grammars, *Information and Control*, 2, pp. 137–167.

[23] Chow S., Eisen P. Johnson H., Zakharov V.A. (2001), An approach to the obfuscation of control-flow of sequential computer programs, *Information Security, ISC 2001*, LNCS 2200, pp. 144–155, Springer Verlag.

[24] Christodorescu M. and Jha S. (2004), Testing Malware Detectors. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA'04)*.

[25] Ciubotariu M. (2003), *Virus Cryptoanalysis*, Virus Bulletin, november. Disponible sur `http://www.virusbtn.com/magazine/archives/200311/cryptoanalysis.xml`

[26] Cohen F. (1986), *Computer viruses*, Ph. D thesis, University of Southern California, Janvier 1986.

[27] Colbourn C. J. and Dinitz J. H. (1996), *Handbook of Combinatorial Designs*, CRC Press, ISBN 0-8493-8948-8.

[28] Collberg C.S., Thomborson C. (2002), Watermarking, tamper-proofing and obfuscation - Tools for software protection, *IEEE Transactions on Software Engineering*, Vol. 28, No. 8, August 2002, pp. 735-746.

[29] `http://www.cryptovirology.com/`

[30] De Drézigué D., Fizaine J.-P. et Hansma N. (2006), In-depth Analysis of the Viral Threats with OpenOffice.org Documents, *Journal in Computer Virology*, (2), 3.

[31] Denis-Papin M., Kaufmann A. et Faure R. (1963), *Cours de calcul booléien appliqué*, Coll. Bibliothèque de l'ingénieur électricien-mécanicien, Albin Michel éditeur.

[32] Dodge Y. (1999), *Premiers pas en statistiques*, Springer Verlag France, ISBN 2-287-59678-X.

[33] Dréo J., Pétrowski A., Siarry P. et Taillard E. (2003), *Métaheuristiques pour l'optimisation difficile*, Eyrolles.

[34] Ferrie P., Zsör P. (2001), Zmist Opportunities, *Virus Bulletin*, March 2001, pp. 6–7.

[35] Filiol E. (2001), *Techniques de reconstruction en cryptographie et en théorie des codes*, Thèse de doctorat, École Polytechnique, Palaiseau.

[36] Filiol E. (2002), A New Statistical Testing for Symmetric Ciphers and Hash Functions. In : *Proceedings of the 4th International Conference on Information and Communication Security 2002*, Lecture Notes in Computer Science, Vol. 2513, pp. 342–353, Springer Verlag.

[37] Filiol E. (2003), Le virus de boot furtif Stealth, *MISC, Le journal de la sécurité informatique*, Numéro 6.

[38] Filiol E. (2003), *Les virus informatiques : théorie, pratique et applications*, Collection IRIS, Springer France.

[39] Filiol E. (2004), Le ver MyDoom, *Journal de la sécurité informatique MISC*, (**13**), May 2004.

[40] Filiol E. (2004), Le ver Blaster/Lovsan, *Journal de la sécurité informatique MISC*, numéro 11, janvier 2004.

[41] Filiol E. (2004), Le chiffrement par flot, *Journal de la sécurité informatique MISC* 16, novembre 2004.

[42] Filiol E (2005), SCOB/PADODOR : quand les codes malveillants collaborent, *Journal de la sécurité informatique MISC*, numéro 17, Janvier 2005.

[43] Filiol E. (2005), Le virus WHALE : le virus se rebiffe, *Journal de la sécurité informatique MISC*, 19, Mai 2005.

[44] Filiol E. (2005), Cryptologie malicieuse ou virologie cryptologique ? *Journal de la sécurité informatique MISC*, 20, juillet 2005.

[45] Filiol E. (2005), Le virus Ymun : la cryptanalyse sans peine, *Journal de la sécurité informatique MISC*, 20, juillet 2005.

[46] Filiol. P. (2005), Le virus PERRUN : méfiez vous des images... et des rumeurs, *Journal de la sécurité informatique MISC*, numéro 18, Mars 2005.

[47] Filiol E. (2005), Le virus Bradley ou l'art du blindage total, *Journal de la sécurité informatique MISC* 20, juillet 2005.

[48] Filiol E. (2005), Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis : the BRADLEY virus, In : *Proceedings of the 14th EICAR Conference*, pp.- 201–217.

[49] Filiol E. (2005), Évaluation des logiciels antiviraux : quand le marketing s'oppose à la technique, *Journal de la sécurité informatique MISC*, 21, septembre 2005.

[50] Filiol E. (2005), La simulabilité des tests statistiques, *Journal de la sécurité informatique MISC*, 22, novembre 2005.

[51] Filiol E., Helenius M. et Zanero S. (2005), Open Problems in Computer Virology, *Journal in Computer Virology*, (1), 3-4, pp. 55–66.

[52] Filiol E. (2006), Malware Pattern Scanning Schemes Secure Against Black-box Analysis. In : *Proceedings of the 15th EICAR Conference*, publié dans *Journal in Computer Virology*, EICAR 2006 Special Issue, Vol. 2, Nr. 1, pp. 35–50.

[53] Filiol E., Jacob G. et Le Liard M. (2006), Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies, *Journal in Computer Virology*, (2)-4, WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds.

[54] Filiol E. (2007), Formalisation and Implementation Aspects of k-ary (malicious) codes, à paraître.

[55] Filiol E., Josse S. (2007), A Statistical Model for Viral Detection Undecidability, à paraître.

[56] Filiol E. (2007), Formal Model Proposal for (Malware) Program Stealth, à paraître.

[57] Garey M. R. et Johnson D. S. (1979), *Conputers and Intractability*, Freeman.

[58] Glover F. (1986), Future paths for integer programming and links to artificial intelligence, *Computers and Operations Research*, (13), pp. 533–549.

[59] Goldberg R. P. (1973), *Architectural Principles for Virtual Computer Systems*, PhD thesis, Harvard University.

[60] Goldberg L. A., Goldberg P. W., Phillips C. A. and Sorkin G. B. (1998), Constructing computer virus phylogenies, *Journal of Algorithms*, vol. 26, pp. 188–208.

[61] Goldwasser S. and Micali S. (1984), Probabilistic Encryption, *Journal of Computer and Systems Sciences*, (28), pp. 270–299.

[62] GOST 28147-89 (1980), Cryptographic Protection for Data Processing Systems, White paper, available from `http://www.cisco.com/warp/public/732/netflow`, Government Comittee of the USSR for Standards.

[63] Hoglund G. et Butler J. (2006), *Rootkits : Subverting the Windows Kernel*, Addison Wesley, ISBN 0-321-29431-9.

[64] Hopcroft J. et Ullman J. D. (1979), *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, ISBN 0-201-44124-1.

[65] Hopper N. J., Langford J. et van Ahn (2002), Provably Secure Steganography. In : *Advances in Cryptology - CRYPTO'02*, LNCS 2442, pp. 77-92, Springer.

[66] Ilachinski A. (2001), *Cellular Automata : A Discrete Universe*, World Scientific.

[67] Jacob G., Le Liard M. (2006), Evaluation des méthodes de détection comportementale des virus, Rapport de projet Mastère SSI, Laboratoire de virologie et de cryptologie et Supélec Bretagne.

[68] Jones N. D. (1997), *Computability and Complexity*, MIT Press.

[69] Josse S. (2005), Techniques d'obfuscation de code : chiffrer du clair avec du clair, *Journal de la sécurité informatique MISC*, 20, juillet 2005.

[70] Josse S. (2006), How to measure the effectiveness of an AV. In : *Proceedings of the 15th EICAR Conference*, publié dans *Journal in Computer Virology*, EICAR 2006 Special Issue, Vol. 2, Nr. 1, 51–66.

[71] Karim Md. E., Walenstein A. and Lakhotia A. (2005), Malware Phylogeny Generation using Permutations of Code, *Journal in Computer Virology*, Vol. 1, 1-2, pp. 13–23.

[72] Karp R. M. et Rabin M. O. (1981), Efficient Randomized Pattern-Matching Algorithms, Rapport technique TR-31-81, Ayken Computation Laboratory, Harvard University.

[73] Kearns, M. and Vazirani U. (1994), *An Introduction to Computational Learning Theory*, MIT Press, Cambridge, MA, ISBN 0-262-11193-4.

[74] Kephart J. O. and Arnold W. (1994), Automatic Extraction of Computer Virus Signatures. In : *Proceedings of the 4th Virus Bulletin International Conference*, pp. 179–194, Virus Bulletin Ltd.

[75] King S. T., Chen P. M., Wang Y.-M., Verbowski C., Wang H. J. et Lorch (2006), SubVirt : Implementing Malware With Virtual Machines, Université du Michigan et Microsoft Research.

[76] Kirkpatrick S., Gelatt S. et Vecchi M. (1983), Optimization by simulated annealing, *Science*, (220), 4598, pp. 671–680.

[77] Kortchinsky K. (2003), 0-Days - Recherche et exploitation de vulnérabilités en environnement Win32, *Actes de la conférence SSTIC 2005*, pp. 1–18. Disponible sur le site `http://www.sstic.org`

[78] Kortchinsky K. (2004), Cryptographie et reverse-engineering en environnement Win32, *Actes de la conférence SSTIC 2004*, pp. 129–144. Disponible sur le site `http://www.sstic.org`

[79] Kulesza K. and Kotulski Z. (2003), On Secret Sharing for Graphs, `arxiv.org/cs.CR/0310052`.

[80] Knuth D. E., Morris Jr J. H. et Pratt V. R. (1977), Fast pattern-matching in strings, *SIAM Journal on Computing*, (6), 2, pp. 323–350.

[81] Lakhotia A., Kapoor A. et Kumar E. U. (2004), Are Metamorphic Viruses Really Invincible. Part 1, *Virus Bulletin*, pp. 5–7, décembre 2004.

[82] Lejeune M. (2005), *Statistique : la théorie et ses applications*, Collection Statistique et probabilités appliquées, Springer Verlag, ISBN 2-287-21241-8.

[83] Lenstra A. K. (2000), Integer Factoring, *Designs, Codes and Cryptography*, (19), pp. 101–128.

[84] Lewis H. R. et Papadimitriou C. H. (1981), *Elements of Theory of Computation*, Prentice Hall.

[85] LiTlLe VxW (2003), Polymorphism and Intel Instruction Format, *29A E-zine*, 7, `http://www.29a.net/`.

[86] Loi pour la confiance en l'économie numérique, *Journal Officiel*, 22 Juin 2004. Une présentation détaillée de cette loi peut être consultée dans [38, chapitre 5].

[87] Ludwig M. A. (1991), *The Little Black Book of Computer Viruses*, American Eagle Press.

[88] Ludwig M. A. (2000), *The Giant Black Book of Computer Viruses*, Second edition, American Eagle Press. La traduction française de la première édition a été assurée par Pascal Lointier aux éditions Dunod, sous le titre : *Du virus à l'antivirus.*

[89] McCluskey E. J. (1956), Minimization of Boolean Functions, *Bell System Tech. J.*, Vol. 35, Nr 5, pp. 1417–1444.

[90] MacWilliams F. J. et Sloane N. J. A. (1977), *The Theory of Error-correcting Codes*, North-Holland, Elsevier, ISBN 0-444-85193-3.

[91] Martineau T. (2004), La clef USB : votre nouvel ennemi, *Journal de la sécurité informatique MISC*, 16, novembre 2004.

[92] Menezes A. J., Van Oorschot P. C., Vanstone S. A. (1997), *Handbook of Applied Cryptography*, CRC Press, Boca Raton, New York, London, Tokyo, ISBN 0-8493-8523-7.

[93] Micali S., Rackoff C. and Sloan B. (1988), The Notion of Security for Probabilistic Cryptosystems, *SIAM Journal in Computing*, (17), pp. 412–426.

[94] Michaels J. G. (2000), Algebraic Structures. In : *Handbook of Discrete and Combinatorial Mathematics*, Rosen K. H. editor, pp. 344–354, CRC Press, ISBN 0-8493-0149-1.

[95] Morain F. (2002), Thirty Years of Integer Factorization, F. Chyzak ed., *Proceedings of the Algorithms Seminar 2000 - 2001*, pp. 77–80.

[96] Morales J. A., Clarke P.J., Deng Y. et Golam Kibria B. M. (2006), Testing and Evaluating Virus Detectors for Handheld Devices, *Journal in Computer Virology*, (2), 2.

[97] Morin B. (2006), Intrusion Detection vs Virology. In : *Proceedings of the First Workshop in Theoretical Computer Virology*, G. Bonfante et J.- Y. Marion eds, Nancy, May 2006.

[98] National Institue of Standards and Technology, NIST FIPS PUB 180, *Secure Hash Standard*, U.S. Department of Commerce, May 1993.

[99] Papadimitriou C. H. (1995), *Computational Complexity*, Addison Wesley, ISBN 0-201-53082-1.

[100] Pearce S. (2003), *Viral Polymorphism*, SANS Institute.

[101] Pearl, J. (1990), *Heuristique : stratégies de recherche intelligente pour la résolution de problèmes par ordinateur.* Cepadues éditions, ISBN 2-95428-186-1.

[102] Pomerance C. (1984), The Quadratic Sieve Factoring Algorithm. In : *T. Beth and N. Cot and I. Ingemarsson eds, Advances in Cryptology - Eurocrypt'84*, LNCS 209, pp. 169–182, Springer Verlag.

[103] Post E. (1947), Recursive unsolvability of a problem of Thue, *Journal of Symbolic Logic*, 12, pp. 1–11.

[104] Prins C. (1994), *Algorithmes de graphes*, Eyrolles, ISBN 2-212-09020-X.

[105] Project funded by the Fund for Scientific Research - Flanders (2003), *Coordinated Research of Program Obfuscation*, `http://www.elis.regent.be/~banckaer/obfuscation/proposal.html`

[106] Qozah (1999), Polymorphism and grammars, *29A E-zine*, 4, `http://www.29a.net/`.

[107] Quine V. W. (1952), The Problem of Symplifying Truth Functions, *The American Mathematical Monthly*, Vol. 59, Nr. 8, pp. 521–531.

[108] Quine V. W. (1955), A Way to Simplify Truth Functions, *The American Mathematical Monthly*, Nov. 1955.

[109] Riordan J., Schneier B. (1998), Environmental key generation towards clueless agents, In *Mobile Agents and Security Conference'98*, G. Vigna ed., Lecture Notes in Computer Science, pp 15–24, Springer-Verlag, 1998.

[110] Rivest R.L. (1992), *The RC4 Encryption Algorithm*, RSA Data Security Inc.

[111] Rivest R.L., Robshaw M.J.B. Robshaw, Sidney R. and Yin Y.L. (1998), The RC6 block cipher, *Proc. of the 1st AES Candidate Conference*, August 1998, Ventura.

[112] Robert C. P. (2005), *Le choix bayésien : principes et pratique*, Collection Statistique et probabilités appliquées, Springer Verlag France, ISBN 2-287-25173-1.

[113] Robert F. (1995), *Les systèmes dynamiques discrets*, Collections Mathématiques et Applications, Vol. 19, Springer Verlag France, ISBN 3-540-60086-8.

[114] Russinovitch M. (2005), Sony, Rootkits and Digital Rights Management Gone Too Far, `http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html`, 31 octobre 2005. Ce document présente techniquement le rootkit utilisé par Sony.

[115] Russinovitch M. (2005), Déterrer les Root Kits, *Le mensuel des environnements Windows Server*, pp. 36–46, Octobre 2005.

[116] Rutkowska J. (2003), Detecting Windows Server Compromises. In : *HivenCon Security Conference.* Disponible sur `http://www.invisiblethings.org/papershivercon03_joanna.ppt`

[117] Rutkowska J. (2004), Red Pill... or how to detect VMM using (almost) one CPU instruction, `http://invisiblethings.org/papers/redpill.html`

[118] Rutkowska J. (2006), Subverting Vista Kernel for Fun and Profit, *SysCan'06 Conference*, July 21st, Singapoure.

[119] Schmall M. (1998), *Heuristische Viruserkennung.* Diplom thesis, Universität Hamburg.

[120] Schmall M. (2003), *Classification and identification of malicious code based on heuristic techniques utilizing Meta languages*, PhD thesis, University of Hamburg.

[121] Serazzi G. et Zanero S. (2003), Computer Virus Propagation Models, *Tutorials of the 11th IEEE/ACM Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecom. Systems - MASCOTS 2003*, Maria Carla Calzarossa and Erol Gelenbe eds., Lecture Notes in Computer Science 2965, Springer Verlag.

[122] Shah P. (2002), Code Obfuscation For Prevention of Malicious Reverse Engineering Attacks, `http://islab.oregonstate.edu/koc/ece478/02Reports/S2.pdf`

[123] Shannon C. E (1948), A Mathematical Theory of Communication, *Bell Syst. Tech. J.*, Vol. 27, pp. 379–423 and 623–656.

[124] Shannon C. E. (1949), Communication Theory of Secrecy Systems, *Bell System Technical Journal*, (28), pp. 656–715.

[125] Simova M. (2005), *Stealthy Ciphertext*, Master of Science thesis, San Jose State University. Disponible sur `http://www.cs.sjsu.edu/faculty/stamp/students/cs299report.pdf`.

[126] Sparks S. et Butler J. (2005), Raising the Bar For Windows Rootkit Detection, *Phrack*, 63.

[127] Spinellis D. (2003), Reliable Identification of Bounded-length Viruses is NP-complete, *IEEE Transactions in Information Theory*, Vol. 49, No. 1, pp. 280–284, janvier.

[128] Staniford S., Paxson V. et Weaver N. (2002), How to own the Internet in your spare time. In : *Proceedings of the 11th USENIX Security Symposium.*

[129] Stimms, S., Potter, C. and Beard A. (2004), 2004 Information Security Breaches Survey, UK Department of Trade and Industry (Ministère anglais du commerce et de l'industrie), 2004. Disponible sur `http://www.security-survey.gov.uk`. Une vidéo présentant le rapport à la presse ainsi qu'une synthèse à destination des décideurs sont également disponible sur ce site.

[130] Szor P. (2005), *The Art of Computer Virus Research and Defense*, Symantec Press and Addison Wesley, ISBN 9-780321-304544.

[131] The Mental Driller (2002), Advanced polymorphic engine construction, *29A E-zine* 5, `http://www.29a.net/`.

[132] The Mental Driller (2003), Metamorphism in practice, *29A E-zine* 6, `http://www.29a.net/`. Le code du virus *Win32/Linux.MetaPHOR* est lui dans le numéro 8 (29A E-zine 8).

[133] Tzeitzin G.C (1958), Associative calculus with an unsolvable calculus problem, *Tr. Math. Inst. Steklov Akad. Nauk SSSR*, 52, pp. 172–189.

[134] U.S. Copyright Office Summary (1998), The Digital Millenium Copyright Act of 1998, `http://www.copyright.gov/legislation/dmca.pdf`

[135] US Government Protection Profile, Anti-virus Applications for Workstations in Basic Robustness Environments, Version 1.0, January 2005. Available at `www.iatf.net/protection_profiles/index.cfm`

[136] Veldman F. (1999), Heuristic Anti-Virus Technology, `http://vx.netlux.org/lib/static/vdat/epheurs1.htm`

[137] Ventsel H. (1973), *Théorie des probabilités*, éditions Mir.

[138] Vernam G. S. (1926), Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of The American Institute for Electrical Engineers*, Vol. 55, pp 109–115.

[139] Base de données virales VX Heavens, `vx.netlux.org`.

[140] Wang X., Feng D., Lai X. et Yu H. (2004), Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Disponible sur `http://eprint.iacr.org/2004/199`

[141] Wegener I. (1987), *The complexity of Boolean functions*, Wiley.

[142] Wedrzynowski E. (2003), Des trappes dans les clefs, *Actes de la conférence SSTIC 2003*, pp. 248–260. Disponible sur le site `http://www.sstic.org`

[143] Whitehead (2003), Abstract Algebra in Poly Engines, *29A E-zine*, 6, `http://www.29a.net/`.

[144] Wiley B. (2002), *Curious Yellow : The First Coordinated Worm Design*, `http://blanu.net/curious_yellow.html`.

[145] Xiao G.-Z. et Massey J. L. (1988), A spectral characterization of correlation-immune combining functions, *Transactions on Information Theory*, Vol. IT-34 Nr 3, pp 569–571.

[146] Yoo I., Ultes-Nitsche U. (2006), Non-signature-based Virus Detection : Towards Establishing Unknown Virus Detection Technique Using SOM, *Journal in Computer Virology*, (2), 3.

[147] Young A, Yung M. (1996), Cryptovirology : Extorsion-Based Security Threats and Countermeasures, *IEEE Symposium on Security and Privacy*, Oakland, CA, 1996.

[148] Young A, Yung M. (2004), *Malicious Cryptography : Exposing Cryptovirology*, Wiley & Sons.

[149] Zuo Z. et Zhou M. (2004), Some further theoretical results about computer viruses, *The Computer Journal*, Vol. 47, No. 6.

[150] Zuo Z, Zhou M. (2005), On the Time Complexity of Computer Viruses, *IEEE Transactions in Information Theory*, (51), 8.

Index

Achevé d'imprimer sur les presses de l'Imprimerie BARNÉOUD
B.P. 44 - 53960 BONCHAMP-LÈS-LAVAL
Dépôt légal : décembre 2006 - N° d'imprimeur : 612040
Imprimé en France