Nancy-Université
INPL

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.
Il est soumis à la propriété intellectuelle de l'auteur au même titre que sa version papier. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.
D'autre part, toute contrefaçon, plagiat, reproduction illicite entraîne une poursuite pénale.

Contact SCD INPL: mailto:scdinpl@inpl-nancy.fr

LIENS

Code de la propriété intellectuelle. Articles L 122.4
Code de la propriété intellectuelle. Articles L 335.2 – L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Nancy-Université

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Analyse de codes auto-modifiants pour la sécurité logicielle

THÈSE

présentée et soutenue publiquement le 15 octobre 2010

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine

(spécialité informatique)

par

Daniel Reynaud

Composition du jury

Rapporteurs :	Marc Dacier	Professeur, EURECOM
	Marc Pouzet	Professeur, Université Paris-Sud
Examinateurs :	José M. Fernandez	Professeur, École Polytechnique Montréal
	Hélène Kirchner	Directeur de Recherche, INRIA
	Jean-Yves Marion	Professeur, École Nationale Supérieure des Mines de Nancy
	Frédéric Raynal	Ingénieur, Sogeti-CapGemini
Invités :	Cédric Blancher	Ingénieur, EADS IW
	Danilo Bruschi	Professeur, Université de Milan

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

Loria

Table des matières

Introduction vii

Fondements théoriques 1

1 Évolution de la virologie informatique 3
- 1.1 Évolution de la menace virale . . . 3
  - 1.1.1 Années 1970 : la genèse . . . 3
  - 1.1.2 Années 1980 : les premières infections . . . 3
  - 1.1.3 Années 1990 : une sophistication croissante et l'apparition des anti-virus . . . 4
  - 1.1.4 Années 2000-2005 : l'âge d'or des vers . . . 4
  - 1.1.5 Années 2005-2010 : essor de la cybercriminalité, émergence de l'espionnage politique . . . 5
  - 1.1.6 Bilan . . . 6
- 1.2 Évolution de la recherche en virologie informatique . . . 7
  - 1.2.1 Années 1940-1980 : modélisation du vivant . . . 8
  - 1.2.2 Années 1980-2000 : virologie abstraite . . . 8
  - 1.2.3 Années 2000-2010 : des virus aux malwares . . . 9
  - 1.2.4 Bilan . . . 10
- 1.3 Définition informelle des programmes malveillants . . . 10
- 1.4 Conclusion . . . 11

2 Notations et langages de programmation 13
- 2.1 Un langage de haut niveau : `WHILE` . . . 13
  - 2.1.1 Domaine de calcul . . . 13
  - 2.1.2 Interactions avec l'environnement . . . 14
  - 2.1.3 Syntaxe abstraite . . . 14
  - 2.1.4 Syntaxe concrète . . . 15
  - 2.1.5 Sémantique . . . 15
  - 2.1.6 Exemple . . . 17
- 2.2 Un langage de bas niveau : `Machine-GOTO` . . . 17
  - 2.2.1 Domaine de calcul . . . 17

2.2.2 Syntaxe abstraite . . . 18
2.2.3 Syntaxe concrète . . . 18
2.2.4 Sémantique . . . 19
2.2.5 Exemple . . . 21
2.3 Conclusion . . . 21

3 Définition et propriétés des programmes auto-modifiants 23
3.1 Introduction aux programmes auto-modifiants . . . 23
3.1.1 Exemple informel de polymorphisme . . . 23
3.1.2 Autres applications des programmes auto-modifiants . . . 25
3.2 Auto-modifications internes . . . 25
3.2.1 Définition . . . 26
3.2.2 Exemple en `Machine-GOTO` . . . 26
3.3 Auto-modifications externes . . . 27
3.3.1 Définition . . . 27
3.3.2 Exemple en `WHILE` . . . 27
3.4 Propriétés fondamentales des programmes auto-modifiants . . . 28
3.4.1 Équivalence entre auto-modifications internes et externes . . . 28
3.4.2 Calculabilité de la détection des programmes auto-modifiants 28
3.4.3 Différences entre programmes statiques et auto-modifiants . 29
3.5 Transformation des programmes . . . 30
3.5.1 Des statiques en auto-modifiants . . . 30
3.5.2 Des auto-modifiants externes en statiques . . . 31
3.5.3 Des auto-modifiants internes en statiques . . . 31
3.6 Conclusion . . . 33

4 Analyse fonctionnelle des programmes auto-modifiants 35
4.1 Comportement des programmes et politiques de sécurité . . . 35
4.1.1 Définition du comportement des programmes . . . 35
4.1.2 Définition des politiques de sécurité . . . 36
4.1.3 Application aux programmes malveillants . . . 37
4.2 Conformité à une politique . . . 37
4.2.1 Indécidabilité de la détection . . . 37
4.2.2 Approximations décidables . . . 39
4.3 Analyse par instrumentation . . . 40
4.3.1 Définition . . . 40
4.3.2 Problèmes liés à la transparence . . . 41
4.3.3 Construction d'un programme d'instrumentation . . . 42
4.4 Conclusion . . . 44

Analyse opérationnelle 45

5 Typage dynamique et comportements auto-modifiants 47
5.1 Exemple en x86 . . . 47
5.2 Typage dynamique de la mémoire . . . 48
5.2.1 Définition préliminaire . . . 48
5.2.2 Système de typage classique . . . 49
5.2.3 Système de typage monotone . . . 51
5.3 Vagues de code . . . 52
5.3.1 Définition . . . 52
5.3.2 Codes auto-modifiants en profondeur et en largeur . . . 53
5.4 Comportements auto-modifiants . . . 55
5.4.1 Vérification d'intégrité et écrasement de code en C . . . 55
5.4.2 Signatures logiques . . . 57
5.5 Non-interférence . . . 59
5.5.1 Pour une commande . . . 59
5.5.2 Pour une vague . . . 60
5.6 Typage statique . . . 62
5.6.1 Modèle avec variables . . . 62
5.6.2 Indécidabilité dans le cas général . . . 63
5.7 Conclusion . . . 64

6 Applications à la sécurité logicielle 65
6.1 TraceSurfer . . . 65
6.2 Visualisation . . . 66
6.2.1 Définition des graphes d'auto-référence . . . 66
6.2.2 Relation avec les signatures logiques . . . 68
6.2.3 Utilisation comme signature . . . 68
6.3 Analyse de programmes packés . . . 69
6.3.1 Protection par packing . . . 69
6.3.2 Approche classique pour l'unpacking générique . . . 71
6.3.3 Contribution . . . 72
6.4 Détection de vulnérabilités . . . 76
6.4.1 Introduction aux attaques par débordement de tampon . . . 76
6.4.2 Exemple en langage C . . . 76
6.4.3 Analyse par TraceSurfer . . . 78
6.5 Conclusion . . . 79

7 Résultats expérimentaux 81
7.1 Analyse de programmes packés . . . 81
7.1.1 Jeu de test . . . 81
7.1.2 Évaluation des performances . . . 81

7.1.3 Résultats du typage 82
7.2 Analyse de compilateurs à la volée 82
7.2.1 Introduction aux compilateurs à la volée 82
7.2.2 Jeu de test 85
7.2.3 Résultats 85
7.3 Analyse de programmes malveillants 86
7.3.1 Introduction aux techniques anti-virtualisation 86
7.3.2 Jeu de test et protocole expérimental 87
7.3.3 Résultats 87
7.4 Analyse de programmes sains 88
7.4.1 Jeu de test et protocole expérimental 88
7.4.2 Résultats 88
7.5 Conclusion 90

Conclusion **93**

Introduction

Société de l'information. La révolution technologique liée à la mondialisation de l'informatique personnelle et de l'Internet a induit des changements profonds dans la société qui entre dans l'ère de l'information. Au delà des évolutions dans la vie quotidienne, on assiste également à des changements majeurs du fonctionnement des entreprises, des médias et des États. Cette société se nourrit de la numérisation et du traitement automatique des informations.

À bien des égards, les technologies qui ont permis ces évolutions manquent de robustesse et le problème de la confiance dans les systèmes d'information est loin d'être résolu. Ainsi, un nombre croissant de disciplines scientifiques traitent les différents aspects de la sécurité des systèmes d'information. On distingue par exemple :

- la cryptographie, ou science du secret, qui vise à protéger les données ;
- la sécurité des protocoles qui s'intéresse à la protection des échanges entre agents autonomes ;
- la sécurité des programmes qui cherche à garantir le bon fonctionnement des programmes et à protéger les systèmes informatiques contre les programmes malveillants.

Analyse de codes binaires. Cette thèse s'inscrit dans le domaine de la sécurité des programmes et, plus particulièrement, dans celui de l'analyse de codes binaires. Il s'agit de déterminer les propriétés de sécurité de programmes directement exécutables par des processeurs matériels – on parle de programmes en langage machine ou codes binaires. Ces langages, de très bas niveau, ne comportent plus les abstractions utilisées par les créateurs de programmes qui pourraient faciliter leur compréhension. De plus, ils ont été conçus et optimisés pour être exécutés efficacement et non pour être analysés. En conséquence, de nombreuses propriétés que l'on peut déduire des programmes de haut niveau sont indécidables pour des codes binaires. Or, l'analyse binaire est l'unique moyen disponible pour lutter contre les programmes malveillants.

Lutte contre les programmes malveillants. Les programmes malveillants, ou *malwares*, comprennent toutes les formes de programmes indésirables destinés à corrompre l'intégrité d'un système informatique. Ils incluent les virus et les vers, mais également les programmes espions. On peut actuellement diviser la lutte contre les programmes malveillants en deux grandes phases. Dans un premier temps, il s'agit

d'*analyser* un programme dont on ne connaît pas le comportement et de déterminer s'il semble être malveillant. Vient ensuite la phase de *détection* qui consiste à lever une alerte si un programme inconnu présente des similarités avec une liste de programmes reconnus comme malveillants au cours de la phase d'analyse.

La détection est la partie visible de la lutte contre les programmes malveillants, car elle est réalisée par les logiciels antivirus du commerce. Cette phase de détection souffre toutefois de nombreuses limitations bien documentées, dues en grande partie au contexte technique et à l'architecture des codes binaires. En effet, la détection est actuellement essentiellement basée sur la reconnaissance de signatures, c'est-à-dire de motifs censés caractériser les programmes reconnus dans la phase d'analyse. Or, cette recherche de signatures, peu fiable par nature, ne peut fonctionner correctement que si le corps des programmes est statique. Cette hypothèse n'est pas vraie pour une classe particulière de programmes que l'on appelle *programmes auto-modifiants*.

Codes auto-modifiants. Les programmes auto-modifiants fonctionnent de manière singulière car ils sont capables de réécrire leur propre code en cours d'exécution. Absents des modèles de calcul théoriques, ils sont pourtant omniprésents dans les ordinateurs et les systèmes d'exploitations actuels. Ils sont en effet utilisés par les chargeurs d'amorçages (ou *bootloaders*), pour la compilation à la volée ou encore l'optimisation dynamique de code. Ils sont également omniprésents dans les programmes malveillants, dont les auteurs ont bien compris qu'ils constituaient des objets complexes à analyser. Ils sont également virtuellement présents dans tous les autres programmes mais de manière non-intentionnelle. En effet, on peut voir certaines classes de vulnérabilités, par exemple les failles par débordement de tampon, comme la possibilité d'exécuter accidentellement des données – ce qui est un comportement caractéristique des programmes auto-modifiants.

Nous sommes donc face à une situation paradoxale : les programmes auto-modifiants sont indispensables au bon fonctionnement et à la performance des systèmes informatiques, mais ils font également le jeu des cybercriminels. Dans le même temps, ce comportement est souvent indésirable mais on ne sait pas garantir son absence.

Résumé du manuscrit

Cette thèse est consacrée à l'analyse des programmes auto-modifiants dans le contexte de la sécurité des programmes et, plus particulièrement, de la lutte contre les programmes malveillants. Elle se divise en deux grandes parties consacrées respectivement aux propriétés fondamentales des programmes auto-modifiants et à une méthode d'analyse orientée vers l'expérimentation.

Fondements théoriques

Chapitre 1. Ce chapitre propose une perspective historique sur les programmes malveillants et la recherche théorique en virologie informatique. Nous verrons que la notion d'auto-reproduction a perdu sa place centrale au profit de la notion plus floue de malveillance. De plus, les motivations des auteurs de malwares ont énormément évolué avec l'essor de la cybercriminalité et l'émergence du concept de guerre informatique.

Chapitre 2. Nous introduisons dans ce chapitre deux modèles de calcul munis d'une sémantique formelle utiles pour la suite. Il s'agit d'un langage simple de haut niveau et d'un langage proche des langages machines, étendus avec la notion d'environnement. À chaque fois, nous introduisons également une syntaxe concrète qui permet de travailler sur les programmes comme des données du langage.

Chapitre 3. Ce chapitre présente une définition de l'auto-modification naturelle pour chaque modèle de calcul que l'on qualifiera d'interne ou d'externe selon qu'elle repose sur l'environnement ou uniquement sur des opérations du langage. Nous unifions ensuite ces définitions qui caractérisent un même phénomène en montrant des propriétés fondamentales vraies dans les deux cas.

Chapitre 4. Afin de raisonner sur les programmes malveillants, il est nécessaire d'introduire de manière formelle la notion de comportement et de politique de sécurité. Nous montrons que toute politique non-triviale est indécidable, ce qui englobe en particulier le problème de la détection virale.

Analyse opérationnelle

Chapitre 5. Nous présentons ici une contribution majeure de cette thèse : un système de typage dynamique de la mémoire qui permet de caractériser des signatures de comportements auto-modifiants. Ce système de typage est particulièrement adapté aux programmes en langage machine, et a été implémenté dans un prototype, TraceSurfer.

Chapitre 6. Ce chapitre propose différents scénarios d'utilisation du système de typage décrit précédemment pour la sécurité des programmes : l'utilisation de graphes d'auto-référence comme signatures, l'analyse de comportements complexes et la détection de vulnérabilités.

Chapitre 7. Enfin, nous présentons une validation par l'expérience du système de typage et du prototype associé qui ont été testés sur différents ensembles de programmes. Nous donnons également des mesures obtenues lors d'une expérience à grande échelle sur des malwares réels.

Fondements théoriques

1 Évolution de la virologie informatique

En l'espace de quelques décennies, la virologie informatique s'est complètement métamorphosée. À l'origine une curiosité scientifique et technique centrée sur les mécanismes d'auto-reproduction, la présence de codes viraux s'est généralisée au point de devenir une nuisance économique majeure. L'importante croissance du réseau Internet s'est accompagnée d'une criminalisation de la menace et d'une implication plus forte d'acteurs étatiques, notamment militaires, dans le domaine de la sécurité des systèmes d'information.

Nous proposons de retracer l'évolution de cette menace ainsi que celle de la recherche sur les programmes malveillants. En particulier, nous allons mettre en évidence le fait que la notion d'auto-reproduction a perdu sa position centrale dans ce domaine au profit de la notion plus générale de comportement. Enfin, nous proposerons une définition informelle des programmes malveillants qui place l'utilisateur et le créateur du programme au coeur du problème.

1.1 Évolution de la menace virale

Sur une période d'environ 40 ans, la menace virale est passée du statut de problème technique et économique à celui d'un véritable enjeu de société.

1.1.1 Années 1970 : la genèse

Les premiers programmes auto-répliquants expérimentaux sont créés, comme le ver Creeper qui s'est répandu sur ARPANET [CR04], l'ancêtre d'Internet. Le terme "ver" n'est toutefois apparu que plus tard pour désigner un programme se répliquant de manière autonome sur un réseau. On en attribue généralement la paternité à un roman de science-fiction de John Brunner [Bru75], publié en 1975, repris dans le contexte de la virologie informatique par deux chercheurs de Xerox PARC [SH82].

1.1.2 Années 1980 : les premières infections

Les premiers programmes auto-répliquants apparaissent « dans la nature », c'est-à-dire ciblant des ordinateurs du commerce (Apple II et MS-DOS) et en dehors de laboratoires spécialisés.

Ils se propagent via :
- l'échange de disquettes (Elk Cloner, 1982) ;
- l'infection de fichiers exécutables (Virdem, 1986 et Lehigh, 1987) ;
- l'infection du secteur de démarrage (Brain, 1986) ;
- le réseau : c'est le cas du célèbre ver Morris en 1988 qui a infecté 6 000 machines en quelques heures, soit environ 10% d'Internet à l'époque [Spa89].

1.1.3 Années 1990 : une sophistication croissante et l'apparition des anti-virus

Au cours de cette période, la menace virale s'est généralisée pour le système d'exploitation Windows de Microsoft, provoquant une importante couverture médiatique et l'apparition des premiers antivirus commerciaux.

Les infecteurs de fichiers se sophistiquent : le polymorphisme fait son apparition avec le virus Chameleon en 1990 et le moteur de mutation de Dark Avenger en 1992. Il s'agit de changer la syntaxe concrète du virus lorsqu'il se propage afin d'éviter d'être reconnu par une signature d'antivirus.

A partir de 1995, les macro-virus font leur apparition et deviennent prédominants : ce sont des programmes associés à des fichiers de données ciblant notamment les programmes de traitement de texte et les tableurs (ces virus sont donc compatibles avec différents systèmes d'exploitation).

On voit également de nombreux chevaux de Troie sophistiqués permettant de prendre le contrôle d'un ordinateur à distance (BackOrifice, 1998 et SubSeven, 1999).

1.1.4 Années 2000-2005 : l'âge d'or des vers

De nombreuses infections spectaculaires ont lieu :
- Melissa, mars 1999 : ce macro-virus établit un nouveau record d'infection (1 200 000 ordinateurs), provoque des congestions réseau et des pertes qui se comptent en centaines de millions de dollars [Gar99] ;
- ILOVEYOU, mai 2000 : un des vers les plus coûteux de l'histoire, avec des millions de machines infectées et des pertes s'élevant à plusieurs milliards de dollars [1] ;
- Code Red, juillet 2001 : un ver se propageant automatiquement sur les serveurs web via une faille du serveur IIS de Microsoft, infectant 359 000 serveurs en 13 heures [MS01] ;
- Blaster et Sobig, 2003 : des attaques ciblant les postes de travail sous Windows ;

1. Source : Time Magazine du 15 mai 2000, http://www.time.com/time/europe/magazine/2000/0515/cover.html

– MyDoom, janvier 2004 : un ver se propageant par email particulièrement rapidement [2] ;
– Sasser, mai 2004 : un ver se propageant à des millions d'ordinateurs [3] de manière automatique, en exploitant une faille du système d'exploitation Windows.

On notera également l'apparition de virus ciblant les téléphones mobiles qui semble avoir été une tendance éphémère [Hyp10].

1.1.5 Années 2005-2010 : essor de la cybercriminalité, émergence de l'espionnage politique

Les logiciels malveillants actuels n'ont plus beaucoup de points communs avec leurs ancêtres expérimentaux. Ils sont développés par des cybercriminels dont l'objectif est de dégager un maximum de profits avec ce que l'on appelle des botnets, ou réseaux de machines zombies. Ils ont les caractéristiques suivantes :

– l'auto-réplication n'est pas un trait caractéristique de leur fonctionnement ;
– ils cherchent à infecter un grand nombre de machines tout en restant discrets ;
– ils sont contrôlés à distance par un individu ou un groupe d'individus, que l'on identifie très rarement ;
– ils servent à relayer du spam, mener des attaques par déni de service, voler des informations privées [MK10] ;
– ils ont un degré de sophistication élevé (protocoles réseaux robustes, utilisation de la cryptographie, mises à jour régulières) ;
– ils utilisent plusieurs vecteurs de propagation (spam, sites webs malveillants, failles dans les navigateurs ou les lecteurs de documents, installation par d'autres logiciels malveillants, clés USB, échange de logiciels piratés...).

Parmi les botnets les plus marquants, on compte :

– Storm, janvier 2007 : un des premiers botnets à but clairement lucratif, infectant environ 50 000 machines [HSD+08] ;
– Conficker, novembre 2008 : un botnet particulièrement virulent (de l'ordre de 9 millions de machines infectées [4]) et sophistiqué, il utilise notamment des signatures électroniques pour s'assurer que les mises à jour proviennent bien des auteurs du malware [PSY09]. Se propageant automatiquement aux machines vulnérables en réseau et par les clés USB, il a infecté des systèmes critiques non connectés à Internet comme ceux des armées françaises [5] et anglaises [6] ;

2. Source : CNN, http://edition.cnn.com/2004/TECH/internet/01/28/mydoom.spreadwed/
3. Source : BBC News, http://news.bbc.co.uk/2/hi/technology/3682537.stm
4. Source : F-Secure, http://www.f-secure.com/weblog/archives/00001584.html
5. Source : Libération, http://secretdefense.blogs.liberation.fr/defense/2009/02/les-armes-attaq.html
6. Source : The Register, http://www.theregister.co.uk/2009/01/20/mod_malware_still_going_strong/

– Mariposa, mars 2010 : un botnet d'environ 13 millions de machines [7], dont les trois auteurs espagnols ont été récemment arrêtés.

Enfin, on notera une infection remarquable nommée GhostNet [DMR+09], dont la finalité n'était pas le profit mais l'espionnage politique. Le but n'était donc pas d'infecter un maximum de machines à l'aveugle, mais un nombre restreint de cibles de haute valeur. Ce réseau d'espionnage avait les caractéristiques suivantes :

– au moins 1 295 ordinateurs infectés dans 103 pays différents, essentiellement en Asie du sud-est ;
– les attaques ciblaient des journalistes, des organisations non-gouvernementales, des militants des droits de l'homme, des ministères des affaires étrangères, des missions diplomatiques et les personnes liées au gouvernement tibétain en exil à Dharamsala ;
– les ordinateurs utilisés pour contrôler le botnet étaient situés sur l'île de Hainan en Chine ;
– certaines informations exfiltrées par le malware ont été utilisées par des diplomates et des services de renseignement chinois [8], certains en concluent que des agents du gouvernement chinois ont activement participé à l'attaque [NA09].

1.1.6 Bilan

D'expériences sur les mécanismes d'auto-reproduction, les logiciels malveillants sont donc passés au stade d'outils pour les cybercriminels. Malgré l'évolution et la diversification des menaces, les technologies utilisées par les antivirus commerciaux ne semblent toutefois pas avoir évolué de manière significative depuis leur apparition.

Outre la cybercriminalité, les logiciels malveillants sont susceptibles d'être utilisés dans d'autres scénarios d'attaques [Lew10], tels que :

– l'espionnage industriel ou militaire, une extension du renseignement sur le terrain informatique qui se produit probablement quotidiennement ;
– le cyberterrorisme, c'est-à-dire l'utilisation à des fins idéologiques d'attaques informatiques afin de provoquer un sentiment de peur dans la population civile ;
– la guerre informatique, impliquant des attaques entre acteurs étatiques.

Si la cybercriminalité et l'espionnage sont des réalités, il n'y a pas eu à ce jour d'attaques que l'on puisse qualifier de cyberterrorisme ou de guerre informatique. Même si certains doutent du réalisme d'un scénario de type guerre informatique [Ran08], les politiques et les militaires se sont saisis de la question et de nombreux États adoptent une doctrine de cyberdéfense [Bus03], y compris la France [Liv08].

Les incidents indiquant des tensions entre entités géopolitiques dans le cyberespace se produisent de plus en plus fréquemment. On retiendra notamment les

7. Source : ZDNet UK, http://www.zdnet.co.uk/news/security-threats/2010/03/16/how-the-butterfly-botnet-was-broken-40088328/

8. Source : The New York Times de mars 2009, http://www.nytimes.com/2009/03/29/technology/29spy.html

évènements suivants :

- Estonie/Russie, avril 2007 [Les07] : les infrastructures estoniennes (les sites gouvernementaux et bancaires) sont la cible d'attaques d'une portée limitée, mais qui provoquent une prise de conscience du risque. Une des conséquences directes de ces attaques est l'inauguration par l'OTAN en 2008 d'un centre de recherche sur la cyberdéfense en Estonie [9].
- Géorgie/Russie, juillet 2008 : des sites gouvernementaux géorgiens sont la cible d'attaques coordonnées [10] peu de temps avant que la guerre éclate entre la Géorgie et la Russie en Ossétie du Sud. Le gouvernement russe nie toute implication dans les attaques informatiques précédent l'affrontement armé.
- Tibet/Chine, mars 2009 : déploiement du réseau d'espionnage GhostNet, détaillé en section 1.1.5.
- États-Unis/Corée du Sud/Corée du Nord, juillet 2009 : des sites gouvernementaux aux États-Unis et en Corée du Sud sont la cible d'attaques par déni de service, semblables aux attaques contre l'Estonie et la Géorgie. De nombreux éléments laissent penser que la Corée du Nord est derrière les attaques [11], utilisant une variante du ver MyDoom (voir section 1.1.4).
- Google/Chine, décembre 2009 : Google (entreprise de droit américain) annonce l'arrêt de la censure des résultats de son moteur de recherche en Chine [12], suite à des attaques informatiques en provenance de ce pays visant à intercepter des messages des militants chinois des droits de l'homme. Depuis le 23 mars 2010 [13], le site `google.cn` renvoie vers une version non censurée du moteur de recherche, hébergée à Hong Kong (`google.hk`).

1.2 Évolution de la recherche en virologie informatique

La virologie informatique a pour origine la recherche sur les mécanismes d'autoreproduction et la modélisation du vivant. Ce n'est qu'assez récemment avec les travaux de Cohen [Coh86] et Adleman [Adl88] que les virus informatiques sont étudiés en tant que programmes spécifiquement malveillants, et pas seulement autoreproducteurs. La recherche contemporaine s'est ensuite diversifiée et considère le

9. Cooperative Cyber Defence Centre of Excellence, `http://www.ccdcoe.org/`
10. Source : The New York Times du 12 août 2008, `http://www.nytimes.com/2008/08/13/technology/13cyber.html`
11. Source : Wall Street Journal, `http://online.wsj.com/article/SB124701806176209691.html`
12. Source : blog officiel de Google, `http://googleblog.blogspot.com/2010/01/new-approach-to-china.html`
13. Source : Le Monde du 23 mars 2010, `http://www.lemonde.fr/technologies/article/2010/03/23/google-se-replie-a-hongkong-pour-ne-pas-ceder-face-a-pekin_1323236_651865.html`

problème sous des aspects sémantiques [CJS+05], comportementaux [Jac09] ou opérationnels [Fil05, Fil07].

1.2.1 Années 1940-1980 : modélisation du vivant

Bien que ne traitant pas explicitement des virus en tant que programmes malveillants, von Neumann [Neu66] a étudié pendant les années 1940 les mécanismes d'auto-reproduction des automates cellulaires, lesquels présentent des points communs remarquables avec les virus. Un automate cellulaire est un modèle de calcul comportant :

- une grille de cellules à deux dimensions dans une configuration initiale ;
- un ensemble de règles de transition, chaque cellule changeant d'état en fonction de son propre état et de l'état de ses quatre cellules adjacentes.

En appliquant itérativement les règles de transition, on observe une évolution globale de l'automate qui n'est dictée que par la configuration initiale et les règles de transition locales à chaque cellule. Von Neumann a démontré l'existence d'automates cellulaires produisant une copie d'eux-mêmes à l'identique à l'aide d'un constructeur universel (capable de produire n'importe quel automate à partir de sa description) et d'une description de lui-même.

On peut comparer ce mécanisme à un compilateur auquel on demanderait de compiler son propre code source. Il produirait alors une copie de lui même. Par la suite, on a montré que la présence d'un constructeur universel n'était pas une condition nécessaire à l'auto-reproduction [Cod68, Lan84].

1.2.2 Années 1980-2000 : virologie abstraite

Les virus informatiques apparaissent comme objet d'étude sous ce nom avec les travaux de Cohen [Coh86]. Il s'intéresse à l'aspect auto-reproducteur de ces programmes et à l'impact concret qu'ils peuvent avoir sur les systèmes informatiques. On lui doit donc la notion de programme auto-reproducteur malveillant (ou simplement indésirable) qui domine encore aujourd'hui, bien qu'il se soit également intéressé aux virus utiles.

Cohen définit les virus à l'aide de machines de Turing. On rappelle que ces machines offrent le modèle de calcul suivant :

- une bande mémoire infinie, divisée en cases ;
- une tête de lecture dans un état initial, pouvant lire ou écrire un symbole dans la case devant elle, ou se déplacer d'une case ;
- des règles de transition, qui définissent un symbole à écrire et un mouvement à effectuer en fonction de l'état actuel de la tête de lecture et de l'état lu sur la bande.

Un programme (c'est-à-dire une suite de symboles sur la bande mémoire) est un virus si et seulement si son interprétation par une machine de Turing conduit à la

modification d'autres symboles sur la bande (qui modélisent d'autres programmes) pour contenir des copies potentiellement différentes du virus initial.

Ce formalisme conserve la notion d'auto-reproduction au sens de von Neumann, et y intègre les éléments suivants :

- mutation : des modifications peuvent se produire au cours de la copie ;
- infection : un virus modifie des programmes existant dans l'environnement pour qu'ils deviennent des virus.

Cohen montre également le résultat marquant suivant : il n'existe pas de programme qui permette de décider si un programme arbitraire est un virus. Concrètement, cela a pour conséquence qu'il n'existe pas d'antivirus parfait pour les ordinateurs courants.

Par la suite, Adleman [Adl88] s'abstrait de la mécanique des machines de Turing et propose une définition des virus avec des fonctions récursives. Il classifie les infections selon leur comportement fonctionnel, c'est-à-dire uniquement en étudiant la relation entre l'entrée et la sortie des programmes. Ces travaux sont importants car, pour la première fois, ils permettent d'envisager les infections informatiques de manière indépendante de tout modèle de calcul particulier.

Enfin, le formalisme d'Adleman a été généralisé par Bonfante, Kaczmarek et Marion [BKM06] qui ont montré le rôle central et constructif du deuxième théorème de récursion de Kleene [Kle52] pour l'auto-reproduction, celui-ci pouvant être vu comme un compilateur de virus. Ils ont donc répondu à une question soulevée par Cohen en montrant que tout modèle de calcul satisfaisant le théorème de récursion pouvait produire des virus. En particulier, même des modèles de calculs très limités, non Turing-complets, vérifient le théorème de récursion et permettent donc des constructions virales [Kac08].

1.2.3 Années 2000-2010 : des virus aux malwares

La question de l'auto-reproduction ne joue plus un rôle central en virologie informatique, celle-ci s'étant adaptée au contexte technique pour s'intéresser avant tout à la sécurité des systèmes d'information et au comportement des programmes. En effet, la notion d'auto-reproduction est en principe indépendante de la notion de malveillance même si, en pratique, les programmes auto-reproducteurs sont généralement considérés comme indésirables. Pour des raisons opérationnelles, il est devenu primordial de se protéger contre l'ensemble des programmes malveillants.

On assiste également à l'émergence de l'analyse binaire qui est la convergence de l'analyse de programmes malveillants et de la recherche de vulnérabilités [SBY+]. Ces deux domaines ont pour point commun de fonctionner sur du code machine, et les algorithmes développés pour un domaine sont généralement applicables dans l'autre domaine. Par exemple :

- la résolution de contraintes peut servir à la fois à forcer des logiciels à exhiber un comportement malveillant[MKK07] et à générer automatiquement des tests

[GLM+08] ;
- le calcul d'une distance entre les programmes sert à la fois à obtenir des phylogénies de logiciels malveillants [GGPS96] et à mettre en évidence des failles à partir d'un logiciel et d'une mise à jour corrigeant la faille[BPSZ08] ;
- la spécification de la sémantique du code machine permet de normaliser les programmes malveillants [CKJ+05] mais également de prouver qu'un programme compilé est conforme à sa spécification [Myr09], ou d'exhiber un contre-exemple si ce n'est pas le cas [Hee09].

1.2.4 Bilan

L'évolution de la menace virale a façonné la production scientifique liée à la sécurité des systèmes d'information. Cette thèse vérifie la tendance actuelle, et s'inscrit dans le cadre de l'analyse de programmes malveillants (ou malwares), plutôt que celui de l'analyse des programmes auto-reproducteurs (qui peuvent être malveillants ou non).

1.3 Définition informelle des programmes malveillants

Il n'existe pas dans la littérature de définition des programmes malveillants faisant autorité, probablement car ils se prêtent moins à la formalisation que les programmes auto-reproducteurs. Nous pensons toutefois qu'il existe un certain nombre de propriétés permettant de caractériser les programmes malveillants. Pour cela, définissons d'abord définir ce qu'est une *action sensible* :

Définition 1. Une action sensible sur un système informatique est une action ayant des conséquences sur la confidentialité, l'intégrité ou la disponibilité des données de l'utilisateur du système.

Nous définissons ensuite deux principes d'interaction des programmes avec l'utilisateur :

Définition 2. Un programme réalisant une action sensible doit vérifier les deux principes suivants :
- principe d'information : le programme doit alerter l'utilisateur avant de réaliser une action sensible, notamment sur ses effets potentiellement indésirables ;
- principe de contrôle : le programme doit permettre à l'utilisateur de contrôler son fonctionnement (interruption, désinstallation).

Nous disposons maintenant de tous les critères pour définir un programme malveillant :

Définition 3. Un programme malveillant (ou malware) est un programme créé intentionnellement pour réaliser des actions sensibles en violation des principes de contrôle et d'information.

On remarquera que pour avoir le statut de programme malveillant, il est nécessaire que le programme ait été spécialement créé pour être malveillant. En effet, la malveillance est une notion subjective qui implique une volonté de nuire. Ce point est important et fait explicitement référence au fait que les programmes malveillants n'apparaissent pas "naturellement" et ne sont pas le fruit du hasard. Ils sont créés par des personnes conscientes de leur caractère nuisible. Cela exclut les erreurs de programmation qui pourraient avoir un effet indésirable, mais non-intentionnel.

On remarquera également la présence de l'utilisateur dans la définition, non seulement avec l'impact objectif des actions sensibles sur ses données, mais aussi avec ses attentes subjectives. À l'origine de toute exécution d'un programme informatique, il y a un utilisateur qui attend un résultat (en particulier, il s'attend tacitement à ce que le programme respecte les principes d'information et de contrôle). Bien qu'il n'ait pas formulé explicitement les principes d'information et de contrôle, Cohen [Coh86] avait déjà prévu le cas d'un virus qui respectait ces principes, sous la forme d'un virus demandant la permission avant de se propager. Il était arrivé à la conclusion qu'un tel virus n'était pas malveillant.

Selon cette définition, la problématique des programmes malveillants est donc avant tout une problématique humaine, puisqu'il s'agit de confronter l'intention du créateur avec les attentes de l'utilisateur. L'intention et les attentes étant des notions subjectives, il n'y a donc pas de programme objectivement malveillant (c'est-à-dire abstraction faite du sujet observant). Toute approche purement technique pour la détection des programmes malveillants est donc vouée à l'échec.

1.4 Conclusion

Dans ce chapitre, nous avons vu que la notion d'auto-reproduction n'était plus fondamentale pour la compréhension des programmes malveillants. Nous sommes amenés à placer le comportement au centre de l'analyse des programmes et donc à confronter deux acteurs avec des intentions et des attentes potentiellement incompatibles. Cela rend fondamentalement impossible toute approche purement automatique pour la détection des programmes malveillants.

Dans les chapitres suivants, nous introduirons deux modèles de calcul permettant de modéliser les interactions avec le système, puis nous nous baserons sur ces interactions pour définir le comportement des programmes.

2 Notations et langages de programmation

Nous proposons ici deux modèles de calcul que nous réutiliserons dans les chapitres suivants. Le premier modèle est un langage de haut niveau classique avec un contrôle de flot structuré et des variables arbitrairement grandes, étendu avec la notion d'appel système. Le deuxième modèle est plus proche des langages machine, avec un contrôle de flot non structuré et des cases mémoires finies à la place des variables.

Dans les deux cas, nous détaillerons une syntaxe concrète permettant d'encoder les programmes dans leur propre domaine de calcul. On pourra ainsi travailler sur les programmes qui prennent des programmes en entrée et en sortie.

2.1 Un langage de haut niveau : `WHILE`

On considère qu'un langage est de haut niveau lorsque :
- il est préfixe (c'est-à-dire qu'aucun programme du langage n'est préfixe d'un autre programme) ;
- le contrôle de flot est structuré, la distinction entre programme et donnée étant alors imposée par la grammaire du langage (typiquement, les données doivent être affectées à une variable ou apparaître dans le calcul d'une expression).

Dans tous les cas, la transformation de la syntaxe concrète du programme en syntaxe abstraite (le *parsing*) doit permettre de distinguer de manière non ambigüe ce qui constitue une donnée et ce qui constitue une instruction du langage.

Comme exemple de langage de haut niveau, nous réutilisons le langage `WHILE` défini dans [Jon97], étendu avec la notion d'interaction avec l'environnement (ou appel système).

2.1.1 Domaine de calcul

Les programmes `WHILE` travaillent sur l'ensemble des arbres binaires construits à partir des éléments `cons` et `nil` d'arité respective 2 et 0. Il est défini comme le plus petit ensemble satisfaisant :

$$\texttt{nil} \in \mathbb{D} \qquad\qquad \forall d_1, d_2 \in \mathbb{D} : \texttt{cons}(d_1, d_2) \in \mathbb{D}$$

Cet ensemble permet d'encoder les n-uplets de manière directe. En effet, pour tous $d_1, \ldots, d_n$, nous définissons le n-uplet $(d_1, \ldots, d_n)$ ainsi :

$$(d_1, \ldots, d_n) = \texttt{cons}(d_1, \ldots, \texttt{cons}(d_n, \texttt{nil}) \ldots)$$

Pour les tests booléens, on assimilera `nil` à `false` et $\texttt{cons}(\texttt{nil}, \texttt{nil})$ à `true`.

2.1.2 Interactions avec l'environnement

On étend le langage `WHILE` standard avec la notion d'interaction avec l'environnement. Les interactions sont réalisées via des appels à des fonctions de l'interpréteur, définies de la manière suivante :

Définition 4. L'*interface* est la fonction Σ telle que :
- Σ : `WHILE-symbols` $\to (\mathbb{D} \to \mathbb{D})$ associe à un symbole du langage `WHILE` une fonction partielle de $\mathbb{D}$ dans $\mathbb{D}$;
- le domaine de Σ est fini.

On appelle les éléments de $dom(\Sigma)$ les *appels système*.

La sémantique des appels système est externe à celle du langage `WHILE`. Cela permet de modéliser les systèmes informatiques réels qui sont hiérarchisés en niveaux, et pour lesquels les appels au niveau inférieur se comportent effectivement comme des boîtes noires qui prennent un paramètre et renvoient un résultat.

Par exemple, sur l'architecture x86 le langage `WHILE` correspondrait à l'espace utilisateur (ou ring 3) alors que le niveau inférieur serait l'espace noyau (ou ring 0).

Remarque 5. Le niveau utilisateur et le niveau système/interpréteur peuvent être implémentés dans le même langage (comme c'est le cas en x86), ou être des langages complètement différents.

Remarque 6. Les programmes `WHILE` classiques sont écrits pour l'environnement $\Sigma = \{\texttt{read}, \texttt{write}\}$ qui leur permet de lire leur entrée et de retourner un résultat.

2.1.3 Syntaxe abstraite

On utilise les conventions suivantes :
- les variables sont notées *Var* ;
- les éléments de $\mathbb{D}$ sont notés *Val* ;
- les appels système sont notés *Call*.

Les programmes WHILE ont la syntaxe EBNF suivante :

$$
\begin{aligned}
\textit{expression} &::= \textit{Var} \\
&\mid \textit{Val} \\
&\mid \texttt{cons}(\textit{expression},\ \textit{expression}) \\
&\mid \texttt{head}(\textit{expression}) \\
&\mid \texttt{tail}(\textit{expression}) \\
&\mid \texttt{=?}\ \textit{expression}\ \textit{expression} \\
&\mid \textit{Call(expression)} \\
\textit{commande} &::= \textit{expression} \\
&\mid \textit{Var}\ \texttt{:=}\ \textit{expression} \\
&\mid \textit{commande}\texttt{;}\ \textit{commande} \\
&\mid \texttt{while}\ \textit{expression}\ \texttt{do}\ \textit{commande} \\
\textit{programme} &::= \textit{commande}
\end{aligned}
$$

On emploiera la notation simplifiée `Call()` à la place de `Call(nil)` pour désigner un appel système d'arité zéro.

2.1.4 Syntaxe concrète

La syntaxe concrète des commandes est donnée par une fonction totale et calculable $c \to \underline{c} \in \mathbb{D}$. On étend de manière canonique cette fonction aux programmes :

$$p = (c_1; \ldots; c_n) \to \underline{p} = (\underline{c_1}, \ldots, \underline{c_n})$$

Comme tout programme WHILE peut être encodé comme un élément de $\mathbb{D}$, cela permet de lire et générer des programmes comme n'importe quelle autre donnée.

2.1.5 Sémantique

Valuations

On note les valuations de variables $\sigma : \texttt{Var} \to \mathbb{D}$. Ce sont des fonctions à domaine fini qui associent une valeur à un nom de variable. Pour $\texttt{V}_1,\ \ldots,\ \texttt{V}_n \in \mathit{dom}(\sigma)$ et les valeurs correspondantes $(d_1, \ldots, d_n) \in \mathbb{D}$, on écrit $\sigma = [\texttt{V}_1 \mapsto d_1,\ \ldots,\ \texttt{V}_n \mapsto d_n]$.

Il est courant d'obtenir une nouvelle valuation en changeant seulement quelques éléments de l'ancienne valuation. Ainsi, on écrit $\sigma' = \sigma[\texttt{V} \mapsto d]$ la valuation définie de la manière suivante :

$$\sigma'(x) = \begin{cases} d & \text{si } x = \texttt{V} \\ \sigma(x) & \text{sinon} \end{cases}$$

Évaluation des expressions

On note $\mathsf{E}_{\Sigma,\sigma} : \textit{expression} \to \mathbb{D}$ la fonction d'évaluation des expressions pour un environnement et une valuation donnée.

La sémantique de `WHILE` est la sémantique usuelle, étendue avec l'évaluation des appels système :

$$
\begin{aligned}
\mathsf{E}_{\Sigma,\sigma}[\![\texttt{Var}]\!] &= \sigma(\texttt{Var}) \\
\mathsf{E}_{\Sigma,\sigma}[\![\texttt{Val}]\!] &= \texttt{Val} \\
\mathsf{E}_{\Sigma,\sigma}[\![\texttt{cons(expr1, expr2)}]\!] &= \texttt{cons}(\mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr1}]\!],\ \mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr2}]\!]) \\
\mathsf{E}_{\Sigma,\sigma}[\![\texttt{head(expr)}]\!] &= \begin{cases} e & \text{si } \mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr}]\!] = \texttt{cons}(e,\ f) \\ \texttt{nil} & \text{sinon} \end{cases} \\
\mathsf{E}_{\Sigma,\sigma}[\![\texttt{tail(expr)}]\!] &= \begin{cases} f & \text{si } \mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr}]\!] = \texttt{cons}(e,\ f) \\ \texttt{nil} & \text{sinon} \end{cases} \\
\mathsf{E}_{\Sigma,\sigma}[\![\texttt{=? expr1 expr2}]\!] &= \begin{cases} \texttt{true} & \text{si } \mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr1}]\!] = \mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr2}]\!] \\ \texttt{false} & \text{sinon} \end{cases} \\
\mathsf{E}_{\Sigma,\sigma}[\![\texttt{Call(expr)}]\!] &= \Sigma(\texttt{Call})(\mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr}]\!])
\end{aligned}
$$

Exécution des commandes

Étant donnée une valuation σ, la relation $\texttt{C} \vdash \sigma \to \sigma'$ indique que la nouvelle valuation après exécution de la commande `C` sur σ est σ'.

Cette relation est définie de manière classique et étendue avec l'évaluation des expressions comme des commandes sans effet sur les variables. En particulier, les appels système sont des expressions, on pose donc comme hypothèse qu'ils n'ont pas d'effets de bord sur les variables du programme.

$$
\begin{aligned}
&\texttt{expr} \vdash \sigma \to \sigma && \text{(en particulier, } \texttt{Call(expr)} \vdash \sigma \to \sigma) \\
&\texttt{Var := expr} \vdash \sigma \to \sigma[\texttt{Var} \mapsto d] && \text{si } \mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr}]\!] = d \\
&\texttt{C1 ; C2} \vdash \sigma \to \sigma'' && \text{si } \texttt{C1} \vdash \sigma \to \sigma' \text{ et } \texttt{C2} \vdash \sigma' \to \sigma'' \\
&\texttt{while expr do C} \vdash \sigma \to \sigma && \text{si } \mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr}]\!] = \texttt{nil} \\
&\texttt{while expr do C} \vdash \sigma \to \sigma'' && \text{si } \mathsf{E}_{\Sigma,\sigma}[\![\texttt{expr}]\!] \neq \texttt{nil} \\
& && \text{et } \texttt{C} \vdash \sigma \to \sigma' \text{ et } \texttt{while expr do C} \vdash \sigma' \to \sigma''
\end{aligned}
$$

Remarque 7. La sémantique de `WHILE` est définie pour la plupart des commandes avec une sémantique à petit pas, à la manière de Plotkin [Plo81]. Il y a toutefois une exception avec le corps des boucles `while`, qui sont définies avec une sémantique à

grands pas à la Kahn [Kah87]. En effet, le corps des boucles est une commande, c'est-à-dire un sous-programme pour lequel on n'observe que l'effet global à chaque itération.

Exécution des programmes

Si l'on considère que l'appel `read` lit l'entrée du programme et `write` renvoie un résultat, la sémantique $[\![.]\!] : \texttt{WHILE-programs} \to (\mathbb{D} \to \mathbb{D} \cup \{\bot\})$ permet d'associer une fonction à chaque programme de la forme `X := read(); C; write(Y)` :

$$[\![\texttt{p}]\!](x) = \begin{cases} y & \text{si } \exists \sigma, y \text{ tel que } \texttt{C} \vdash \sigma_1[\texttt{X} \mapsto x] \to \sigma \text{ et } \sigma(\texttt{Y}) = y \\ \bot & \text{sinon} \end{cases}$$

2.1.6 Exemple

A titre d'exemple, voici un programme en `WHILE` qui inverse son entrée :

```
X := read();
Y := nil;
while X do (Y := cons(head(X), Y); X := tail(X));
write(Y)
```

2.2 Un langage de bas niveau : Machine-GOTO

Nous allons introduire un langage machine simple, `Machine-GOTO`, dérivé du langage `GOTO` [Jon97]. On introduit les changements suivants :

- les programmes en `Machine-GOTO` n'ont qu'une seule variable, la mémoire ;
- les expressions ne sont plus récursives.

2.2.1 Domaine de calcul

Les programmes en `Machine-GOTO` calculent sur le même ensemble $\mathbb{D}$ que les programmes en `WHILE`. Pour être plus précis, ils travaillent sur un encodage des arbres binaires en mémoire. Un tel encodage est donné en section 2.2.3.

La mémoire est une suite arbitrairement grande de cases pouvant contenir un symbole $m \in B$, B étant un ensemble fini (comme par exemple, l'ensemble des octets). On accède au contenu de la mémoire via une valuation $\mu : \mathbb{N} \to B$, qui associe une valeur à une adresse.

Remarque 8. Chaque case mémoire ne peut contenir qu'une quantité finie d'information, alors qu'en `WHILE` la quantité d'information contenue dans une variable n'est pas bornée. Cela implique un codage non trivial des variables vers les cases mémoires (section 2.2.3).

2.2.2 Syntaxe abstraite

On utilise les conventions suivantes :
- les adresses mémoire sont notées *Mem* ;
- les labels sont notés *Lab* ;
- les appels systèmes sont notés *Call*.

Les programmes `Machine-GOTO` ont la syntaxe EBNF suivante :

$$
\begin{aligned}
\textit{expression} &::= \texttt{nil} \\
&\mid \textit{Mem} \\
&\mid \texttt{cons}(\textit{Mem}, \textit{Mem}) \\
&\mid \texttt{head}(\textit{Mem}) \\
&\mid \texttt{tail}(\textit{Mem}) \\
&\mid \textit{Call}(\textit{Mem}) \\
\textit{commande} &::= \textit{expression} \\
&\mid \textit{Mem}\ \texttt{:=}\ \textit{expression} \\
&\mid \texttt{if}\ \textit{expression}\ \texttt{goto}\ \textit{Lab} \\
\textit{commande-lab} &::= \textit{Lab}\texttt{:}\ \textit{commande} \\
&\mid \textit{commande-lab}\texttt{;}\ \textit{commande-lab} \\
\textit{programme} &::= \textit{commande-lab}
\end{aligned}
$$

Par convention, on écrira `goto Lab` pour exprimer le saut inconditionnel `if true goto Lab`.

2.2.3 Syntaxe concrète

Nous avons vu en section 2.1.4 que l'on disposait d'un encodage des programmes en arbres. Nous allons maintenant voir comment encoder les arbres en symboles puis les stocker dans la mémoire.

Encodage des arbres

On se donne un alphabet $A \subseteq B$ comprenant les symboles $\boldsymbol{o}$, ., **(** et **)**. On définit ensuite un encodage calculable des arbres binaires en suites de symboles, $enc : \mathbb{D} \to A^*$, qui vérifie les propriétés suivantes :

1. $enc(\texttt{nil}) = \boldsymbol{o}$
2. $enc(\texttt{cons}(a, b)) = \boldsymbol{(}enc(a).enc(b)\boldsymbol{)}$

On note $|.| : A^* \to \mathbb{N}$ la fonction calculable qui donne la longueur des mots de A^*.

On se donne également la fonction partielle de décodage $dec : B^* \to \mathbb{D}$, telle que :

1. $\forall d \in \mathbb{D},\ dec(enc(d)) = d$
2. $\forall b \in B^* - enc(\mathbb{D}),\ dec(b) = \bot$

Un programme $p \in$ `Machine-GOTO` peut donc être encodé en une suite de symboles $enc(\underline{p})$.

Stockage en mémoire

Les mots composés de plusieurs symboles sont stockés en mémoire à des adresses contiguës. Pour $addr \in \mathbb{N}$, $d \in \mathbb{D}$ et une valuation de la mémoire μ, on note $\mu' = \mu[addr \mapsto d]$ la nouvelle valuation de la mémoire avec l'encodage de d à l'addresse $addr$. En particulier, si $enc(d) = a_0, a_1, \ldots, a_n$ est la décomposition de $enc(d)$ en symboles de A, alors cette nouvelle valuation est définie de la manière suivante :

$$\mu'(x) = \begin{cases} a_i & \text{si } x = addr + i \text{ et } 0 \leq i \leq n \\ \mu(x) & \text{sinon} \end{cases}$$

On se donne également la fonction calculable $load_\mu$ qui permet de charger l'encodage d'un arbre stocké en mémoire :

$$\begin{aligned} load_{\mu[x \mapsto d]} :\ & \mathbb{N} \to B^* \\ & x \to b \text{ tel que } dec(b) = d \end{aligned}$$

2.2.4 Sémantique

Évaluation des expressions

Comme précédemment, on note $\mathsf{E}_{\Sigma,\mu} : expression \to \mathbb{D}$ la fonction d'évaluation des expressions (pour un environnement et une valuation de la mémoire donnés). Cette fonction est définie ainsi :

$$\begin{aligned} \mathsf{E}_{\Sigma,\mu}[\![\texttt{nil}]\!] &= \texttt{nil} \\ \mathsf{E}_{\Sigma,\mu}[\![\texttt{Mem}]\!] &= dec(load_\mu(\texttt{Mem})) \\ \mathsf{E}_{\Sigma,\mu}[\![\texttt{cons(Mem1, Mem2)}]\!] &= \texttt{cons}(\mathsf{E}_{\Sigma,\mu}[\![\texttt{Mem1}]\!],\ \mathsf{E}_{\Sigma,\mu}[\![\texttt{Mem2}]\!]) \\ \mathsf{E}_{\Sigma,\mu}[\![\texttt{head(Mem)}]\!] &= \begin{cases} e & \text{si } \mathsf{E}_{\Sigma,\mu}[\![\texttt{Mem}]\!] = \texttt{cons}(e,\ f) \\ \texttt{nil} & \text{sinon} \end{cases} \\ \mathsf{E}_{\Sigma,\mu}[\![\texttt{tail(Mem)}]\!] &= \begin{cases} f & \text{si } \mathsf{E}_{\Sigma,\mu}[\![\texttt{Mem}]\!] = \texttt{cons}(e,\ f) \\ \texttt{nil} & \text{sinon} \end{cases} \\ \mathsf{E}_{\Sigma,\sigma}[\![\texttt{Call(Mem)}]\!] &= \Sigma(\texttt{Call})(\mathsf{E}_{\Sigma,\sigma}[\![\texttt{Mem}]\!]) \end{aligned}$$

Exécution des commandes

Étant donnée une valuation de la mémoire μ, la relation $\texttt{C} \vdash \mu \to \mu'$ indique que la mémoire a pour valuation μ' après exécution de la commande `C` sur la mémoire μ. On définit cette relation de la manière suivante :

$$\begin{aligned}
&\texttt{expr} \vdash \mu \to \mu \\
&\texttt{Mem := expr} \vdash \mu \to \mu[\texttt{Mem} \mapsto d] && \text{si } \mathsf{E}_{\Sigma,\mu}[\![\texttt{expr}]\!] = d \\
&\texttt{if expr goto Lab} \vdash \mu \to \mu
\end{aligned}$$

Image et évolution du programme

L'image d'un programme est une valuation spécifique, séparée de la mémoire. On la définit ainsi :

– pour un programme $\texttt{p} = \texttt{l}_1\texttt{: C}_1\texttt{; } \ldots\texttt{; l}_n\texttt{: C}_n$, l'image initiale est

$$\sigma_0^{\texttt{p}} = [\texttt{l}_1 \mapsto \underline{\texttt{C}_1},\ \ldots,\ \texttt{l}_n \mapsto \underline{\texttt{C}_n}]$$

– lors de l'exécution d'une commande $\texttt{l: C}$, l'image du programme devient

$$\sigma_{i+1}^{\texttt{p}} = \sigma_i^{\texttt{p}}[\texttt{l} \mapsto \underline{\texttt{C}}]$$

La valuation $\sigma^{\texttt{p}}$ est un accumulateur qui permet de garder une trace des commandes exécutées par un programme.

On appelle *évolution du programme* la suite des $(\sigma_i^{\texttt{p}})_{i\in\mathbb{N}}$. Cette suite jouera un rôle particulier dans le chapitre 3 pour la définition des programmes auto-modifiants en langage `Machine-GOTO`.

Exécution des programmes

En `WHILE`, l'exécution des programmes découle naturellement de l'exécution séquentielle des commandes. En particulier, pour un programme $\texttt{p} = \texttt{C}_1\texttt{; } \ldots\texttt{; C}_n$, la commande à exécuter après $\texttt{C}_i$ ne peut être que $\texttt{C}_{i+1}$ (et ne sera exécutée que si $\texttt{C}_i$ termine).

La situation est différente pour le langage `Machine-GOTO`, pour lequel le flot de contrôle n'est pas structuré. Une commande $\texttt{C}_i$ peut transférer le contrôle à n'importe quelle autre commande du programme. La commande suivante à exécuter est donnée par la fonction calculable $\mathit{next} : \mathit{Lab} \to \mathit{Lab}$.

Remarque 9. On considère que le programme est initialement chargé en mémoire et que les labels sont des adresses, c'est-à-dire :

$$\mathit{Lab} \subseteq \mathit{Mem} \text{ et } \mu_0 = \sigma_0^{\texttt{p}}$$

Définition 10. La fonction *next* est définie pour un environnement (Σ, μ) donné :

$$next(l) = \begin{cases} l' & \text{si } \mu(l) = \texttt{if expr goto l'} \text{ et } \mathsf{E}_{\Sigma,\mu}[\![\texttt{expr}]\!] \neq \texttt{nil} \\ l + |enc(\mu(l))| & \text{sinon} \end{cases}$$

On note $\mathrm{ip} \in Lab$ le pointeur d'instruction, et on définit l'exécution d'un programme $\mathtt{p} = \mathtt{C}_1\,;\ \ldots\ ;\ \mathtt{C}_n$ comme une succession de configurations $(\sigma_i^{\mathtt{p}},\ \mathrm{ip}_i,\ \mu_i)_{i\in\mathbb{N}}$:

- $(\sigma_0^{\mathtt{p}},\ \mathrm{ip}_0,\ \mu_0) = ([\mathtt{l}_1 \mapsto \underline{\mathtt{C}_1},\ \ldots,\ \mathtt{l}_n \mapsto \underline{\mathtt{C}_n}],\ \mathtt{l}_1,\ \sigma_0^p)$
- Soit $i \in \mathbb{N}$, si on a $\mu_i(\mathrm{ip}) = \mathtt{C}$ et $\mathtt{C} \vdash \mu_i \to \mu'$, alors :

$$(\sigma_{i+1}^{\mathtt{p}},\ \mathrm{ip}_{i+1},\ \mu_{i+1}) = (\sigma_i^{\mathtt{p}}[\mathrm{ip} \mapsto \underline{\mathtt{C}}],\ next(\mathrm{ip}),\ \mu')$$

Comme en section 2.1.5, la sémantique que l'on vient de définir permet d'associer une fonction de $\mathbb{D}$ dans $\mathbb{D} \cup \{\bot\}$ à chaque programme.

2.2.5 Exemple

A titre d'exemple, voilà le programme en `Machine-GOTO` qui inverse son entrée :

```
l1: @100 := read();
l2: @200 := nil;
l3: if @100 goto l5;
l4: goto l9;
l5: @300 := head(@100);
l6: @200 := cons(@300, @200);
l7: @100 := tail(@100);
l8: goto l3;
l9: write(@200)
```

Contrairement à son équivalent en `WHILE`, ce programme n'est correct que pour certaines entrées, car les adresses où sont stockées les variables sont figées par le programme. On aura un phénomène de débordement de tampon si la taille de l'entrée dépasse 100 symboles (c'est-à-dire si $|enc(\underline{\Sigma(\texttt{read})(\texttt{nil})})| > 100$), et le comportement du programme sera imprévisible. Ce type de problème est courant lorsqu'on passe d'une abstraction fonctionnelle à une concrétisation sur une machine à mémoire finie.

2.3 Conclusion

Nous avons défini :

- un modèle de calcul de haut niveau, `WHILE` ;
- un modèle proche de l'assembleur, `Machine-GOTO`.

Ces deux langages de programmation, dont la syntaxe concrète est encodable dans le domaine de calcul, permettront de définir et illustrer différentes formes d'auto-modifications de manière naturelle dans le chapitre 3.

La modélisation des appels système, absente des modèles classiques en calculabilité, revêt ici un caractère central puisqu'elle permettra de définir dans le chapitre 4 la notion de comportement et de politique de sécurité.

3 Définition et propriétés des programmes auto-modifiants

Nous allons définir dans ce chapitre la notion de programme auto-modifiant en nous appuyant sur les modèles de calcul introduits précédemment. En particulier, nous nous appuierons sur :

- le langage `WHILE` pour définir les auto-modifications externes, qui nécessitent un appel à l'environnement ;
- le langage `Machine-GOTO` pour définir les auto-modifications internes, qui n'emploient que des opérations naturelles du langage.

Nous discuterons ensuite l'idée selon laquelle ces deux types d'auto-modifications, qui s'appliquent naturellement à des modèles de calcul très différents, traduisent en fait un même principe et ont les mêmes propriétés. Nous montrerons donc quelques propriétés fondamentales des programmes auto-modifiants, indifféremment de leur nature interne ou externe.

Une partie de ce chapitre a été publiée dans [BMR09].

3.1 Introduction aux programmes auto-modifiants

3.1.1 Exemple informel de polymorphisme

À la manière de Smullyan [Smu94] et Kaczmarek [Kac08] pour l'auto-référence, nous allons introduire la notion de code auto-modifiant avec des exemples en langage naturel. Pour cela, nous utilisons un modèle simple d'algorithme : les recettes de cuisine.

Prenons par exemple la recette de la pâte à crêpes :

```
1. Dans un saladier, verser 250g de farine et 4 oeufs.
2. Mélanger avec 0,5L de lait.
```

Dans ce contexte, une *signature* est un motif permettant d'identifier une recette à partir de son texte. Par exemple, la recette ci-dessus peut-être identifiée à l'aide du motif suivant :

```
Chercher le mot "farine", puis le mot "oeufs", puis le mot "lait".
```

Évidemment, cette signature n'identifie pas exclusivement la recette des crêpes ci-dessus, elle identifie toute recette qui utilise, dans l'ordre, les mots « farine », « oeufs » et « lait ». Elle a toutefois l'avantage d'être plus courte que la recette que l'on veut identifier, et d'être simple à détecter (cela ne prend pas beaucoup plus de temps que de simplement lire la recette).

Supposons que l'auteur de la recette des crêpes ne soit pas satisfait d'être identifié par une telle signature, il pourrait modifier la recette de manière à ne plus correspondre au motif, tout en s'assurant que le résultat soit le même (c'est-à-dire des crêpes) :

```
1. Dans un bol, verser 0,5L de lait.
2. Dans un saladier, verser 250g de farine et 4 oeufs.
3. Mélanger le contenu du bol et le contenu du saladier.
```

La signature ne correspond pas à la deuxième recette, car on ne trouve pas le mot « lait » après le mot « oeufs ». Cela met en évidence un problème fondamental de la recherche par signature : deux programmes avec des comportements équivalents (dans notre exemple, deux recettes pour le même plat) ne seront pas nécessairement identifiés par la même signature. Dans le domaine de la virologie informatique, ce comportement correspond au métamorphisme [Szo05].

Pour éviter les permutations d'ingrédients, une meilleure signature serait la suivante :

```
Chercher dans n'importe quel ordre : "farine", "oeufs" et "lait".
```

L'auteur de la recette des crêpes a décidé d'échapper à cette nouvelle signature en se munissant d'une nouvelle opération, le code de César. Ce code consiste à coder un message en décalant chaque caractère de trois lettres vers la droite. En utilisant le code de César, la ligne :

```
Dans un saladier, verser 250g de farine et 4 oeufs.
```

devient :

```
Gdqv xq vdodglhu, yhuvhu 250j gh idulqh hw 4 rhxiv.
```

Et la recette devient donc :

```
1. Passer à l'étape (3.).
2. Gdqv xq vdodglhu, yhuvhu 250j gh idulqh hw 4 rhxiv.
3. Décaler (2.) de 3 lettres vers la gauche, puis réaliser (2.).
4. Mélanger avec 0,5L de lait.
```

Le résultat de la recette est toujours le même, mais cette fois on ne se rend pas compte en lisant le texte que l'on a besoin de farine et d'oeufs. Pour le savoir, il faut *exécuter* les instructions de la recette. La signature ne s'applique donc pas, et

il serait vain de vouloir utiliser des mots codés comme "idulqh" et "rhxiv" dans une signature, car il est très facile d'avoir des codages différents. En décalant de n lettres au lieu de 3 dans le code de César, on peut ainsi avoir 25 codages différents de la recette, mais on pourrait utiliser des codages avec beaucoup plus de combinaisons possibles.

La technique illustrée ici est connue sous le nom de polymorphisme [Szo05], et utilise du code auto-modifiant pour réaliser ce polymorphisme. L'auto-modification se produit au cours de l'exécution de la recette, lorsque le texte de la deuxième ligne change. Dans les sections suivantes, nous allons définir formellement les programmes auto-modifiants et montrer quelques propriétés mathématiques de ces programmes.

3.1.2 Autres applications des programmes auto-modifiants

La notion de programme auto-modifiant est liée à celle de réflexivité, c'est-à-dire à la capacité pour un langage de programmation de représenter ses programmes comme des données du langage. Dans un langage réflexif, les programmes ont accès à leur structure interne comme une donnée, qu'ils peuvent inspecter et éventuellement modifier. Les programmes auto-modifiants, réputés complexes à écrire, vérifier et maintenir, ont des applications dans de nombreux domaines :

- le polymorphisme viral, illustré dans la section précédente ;
- la protection contre les modifications non-autorisées [Auc96, GCK05] ;
- la compression des programmes [OMR08] ;
- la performance des programmes avec la compilation à la volée [Ayc03] ;
- la réécriture dynamique de code [Bel05, BDB00].

D'autre part, les programmes auto-modifiants posent problème pour la vérification formelle, car une hypothèse de départ courante des systèmes de vérification est que le texte des programmes est fixe. Des publications récentes s'intéressent au problème de la vérification de programmes dont le contenu peut varier au cours de l'exécution [Myr10, CSV07].

3.2 Auto-modifications internes

Les auto-modifications internes reposent uniquement sur des opérations du langage et influent sur l'image du programme pour l'interpréteur (définie en section 2.2.4). Cela implique que ce type d'auto-modification est possible si l'interpréteur du langage donne accès aux programmes à leur propre image sous forme de variable, ou s'il leur permet de générer de nouvelles instructions dynamiquement dans une forme semblable à l'image initiale du programme. Ce type d'auto-modification est emblématique des langages machines, nous allons donc présenter un exemple en `Machine-GOTO`.

3.2.1 Définition

Définition 11. On dit qu'un programme est *statique sur l'entrée x*, si lors de son exécution (potentiellement infinie) l'image du programme est stable :

$$[\![\mathtt{p}]\!](x) \equiv (\sigma_i^p,\ \mathrm{ip}_i,\ \mu_i)_{i\in\mathbb{N}} \implies \forall i \in \mathbb{N}\ \sigma_i^p = \sigma_0^p$$

On dit qu'un programme est *auto-modifiant sur l'entrée x* s'il n'est pas statique sur l'entrée x.

Définition 12. On généralise la propriété précédente sur l'ensemble des exécutions possibles du programme.

- Un programme est *statique* s'il est statique sur toutes ses entrées.
- Un programme est (inconditionnellement) *auto-modifiant* s'il est auto-modifiant sur toutes ses entrées.
- Un programme qui n'est ni statique ni auto-modifiant est dit *conditionnellement auto-modifiant*.

3.2.2 Exemple en `Machine-GOTO`

L'auto-modification interne s'applique naturellement aux langages machine, car ils ne font généralement pas la distinction entre code et données. En `Machine-GOTO`, cela correspond à $Lab \subseteq Mem$, c'est-à-dire que les labels (désignant le code) sont aussi des adresses mémoire (désignant les données). Voici un exemple de programme auto-modifiant en `Machine-GOTO` pour l'environnement $\Sigma = \{\mathtt{enc},\ \mathtt{stop}\}$ avec $\Sigma(\mathtt{enc}) = enc$ (la fonction d'encodage des arbres en octets) :

```
l1: l1 := enc(stop)
l2: goto l1
```

L'exécution de la première commande provoque la transformation suivante de l'image du programme :

$$\sigma_0^p = [\mathtt{l1} \mapsto \mathtt{l1\ :=\ enc(stop)},\ \mathtt{l2} \mapsto \mathtt{goto\ l1}]$$
$$\downarrow$$
$$[\mathtt{l1} \mapsto \mathtt{stop},\ \mathtt{l2} \mapsto \mathtt{goto\ l1}]$$

La deuxième commande provoque l'exécution de la nouvelle commande désignée par `l1`, c'est-à-dire `stop`. On peut obtenir le même comportement sans toutefois utiliser la réécriture du programme sur lui même :

```
l1: @0 := enc(stop)
l2: goto @0
```

L'image de ce programme n'est pas stable lors de l'exécution, car l'image initiale est étendue avec une nouvelle commande :

$$\sigma_0^p = [\texttt{l1} \mapsto \texttt{@0 := enc(stop)},\ \texttt{l2} \mapsto \texttt{goto @0}]$$
$$\downarrow$$
$$[\texttt{l1} \mapsto \texttt{@0 := enc(stop)},\ \texttt{l2} \mapsto \texttt{goto @0},\ \texttt{@0} \mapsto \texttt{stop}]$$

Ces deux types d'auto-modifications internes étant équivalentes dans ce contexte, nous ne ferons plus la distinction par la suite.

3.3 Auto-modifications externes

Les auto-modifications externes reposent sur la présence dans l'environnement d'un appel à l'interpréteur, que l'on appelle généralement `eval` dans les langages dynamiques de haut niveau comme Lisp, JavaScript, Python, Ruby... Nous allons donc illustrer ce type d'auto-modification avec le langage `WHILE`.

3.3.1 Définition

Définition 13. On appelle fonction `eval` toute fonction partielle $f \in \Sigma$ de la forme

$$f : \mathbb{D} \to \mathbb{D}$$
$$\texttt{p} \to [\![\texttt{p}]\!](\texttt{nil})$$

Définition 14. On dit qu'un programme est *statique sur l'entrée* x si son exécution sur x ne conduit pas à l'évaluation d'une expression de la forme `f expr`, où $\texttt{f} \in \Sigma$ est une fonction `eval`.

Comme précédemment, on en déduit la définition des propriétés *auto-modifiant sur une entrée*, *statique*, *auto-modifiant* et *conditionnellement auto-modifiant*.

3.3.2 Exemple en `WHILE`

Le programme suivant pour l'environnement $\Sigma = \{\texttt{read}, \texttt{eval}\}$ ne fait rien par lui même, il se contente d'exécuter son entrée. On appelle ce type de programmes un mandataire (ou proxy) d'exécution :

```
proxy =
    X := read();
    eval(X)
```

Ce type d'auto-modification est assez semblable aux auto-modifications internes sans réécriture sur soi, puisqu'elle provoque l'exécution de nouvelles commandes

en dehors de l'image originale du programme. Toutefois, l'appel à `eval` se fait en boîte noire et est externe à la sémantique de `WHILE`. Le programme est donc lié à son environnement et la sémantique de `WHILE` ne suffit plus à prédire les effets des programmes auto-modifiants externes.

3.4 Propriétés fondamentales des programmes auto-modifiants

3.4.1 Équivalence entre auto-modifications internes et externes

Dans cette section et les chapitres qui suivent, sauf mention explicite, nous ne ferons plus la distinction entre auto-modifications internes et externes. En effet, la distinction selon le mécanisme employé traduit le même phénomène : l'exécution d'un programme auto-modifiant peut conduire à exécuter sa propre sortie (puisque les nouvelles commandes sont calculées dynamiquement, on les assimile à des sorties partielles du programme).

La distinction entre auto-modification interne et externe permet de différencier la manière dont l'exécution est réalisée, mais ne change pas l'aspect fondamental qui est l'exécution d'une sortie. Par la suite, on utilisera donc indifféremment des exemples en `WHILE` ou `Machine-GOTO`.

3.4.2 Calculabilité de la détection des programmes auto-modifiants

Définition 15. Soit *stat* la fonction totale définie de la manière suivante :

$$stat(\mathtt{p}, x) = \begin{cases} \mathtt{true} & \text{si } \mathtt{p} \text{ est statique sur l'entrée } x \\ \mathtt{false} & \text{sinon} \end{cases}$$

Théorème 16. *La fonction stat n'est pas calculable.*

Démonstration. Le résultat se démontre avec une diagonalisation, semblable au problème de l'arrêt.

Supposons que *stat* est calculable, alors il existe un programme statique `q` tel que $[\![\mathtt{q}]\!] = stat$. On peut supposer que `q` est statique car l'ensemble des programmes statiques est Turing-complet.

Soit le programme `diag` suivant :

```
diag =
    X := read();
    Y := q(cons(diag, enc(diag)));
    while Y do eval(X)
```

Comme $[\![\mathtt{q}]\!]$ est totale, on a soit $[\![\mathtt{q}]\!](\mathtt{diag}, \underline{\mathtt{diag}}) = \mathtt{false}$, soit $[\![\mathtt{q}]\!](\mathtt{diag}, \underline{\mathtt{diag}}) = \mathtt{true}$.

- si $[\![\mathtt{q}]\!](\mathtt{diag}, \underline{\mathtt{diag}}) = \mathtt{false}$, alors par définition de `diag`, `eval(X)` n'est pas exécuté. Cela implique que `diag` est statique sur l'entrée $\underline{\mathtt{diag}}$, et donc que $[\![\mathtt{q}]\!](\mathtt{diag}, \underline{\mathtt{diag}}) = \mathtt{true}$, on aboutit à une contradiction.
- si $[\![\mathtt{q}]\!](\mathtt{diag}, \underline{\mathtt{diag}}) = \mathtt{true}$, alors par définition de `diag`, `eval(X)` est exécuté. Cela implique que `diag` est auto-modifiant sur l'entrée $\underline{\mathtt{diag}}$, et donc que $[\![\mathtt{q}]\!](\mathtt{diag}, \underline{\mathtt{diag}}) = \mathtt{false}$, on aboutit à une contradiction.

Dans les deux cas, on aboutit à une contradiction, l'hypothèse *stat* calculable est donc fausse. □

3.4.3 Différences entre programmes statiques et auto-modifiants

Propriété 17. *On note l'ensemble des programmes statiques* Stat*, l'ensemble des programmes conditionnellement auto-modifiants* CondSmc *et l'ensemble des programmes auto-modifiants* Smc*.*

Ces ensembles vérifient les propriétés suivantes :

1. *ils sont disjoints deux à deux ;*
2. *leur union est égale à l'ensemble des programmes, noté* Progs *;*
3. *ils sont indécidables.*

Démonstration. La propriété 1 découle de la définition des ensembles Stat, Smc et CondSmc : les propriétés caractéristiques de ces ensembles sont mutuellement exclusives, donc tout programme ne peut en vérifier qu'une à la fois.

La propriété 2 découle également de la définition des ensembles, tout programme devant vérifier au moins une des propriétés caractéristiques.

La propriété 3 découle directement du théorème 16. □

Pour montrer une différence importante entre programmes statiques et auto-modifiants, nous allons nous intéresser à la restriction aux programmes sans boucles. On appelle ces programmes les *programmes fuyants*, car ils ne peuvent jamais revenir sur leurs pas. En particulier, pour le langage `WHILE`, il s'agit des programmes avec les commandes `while E do C` restreints à zéro ou une exécution du corps de la boucle, `C`. Une telle commande agit comme un test `if E do C`, avec la sémantique suivante :

$$\mathtt{if\ expr\ do\ C} \vdash \sigma \to \sigma \qquad \text{si } \mathsf{E}_{\Sigma,\sigma}[\![\mathtt{expr}]\!] = \mathtt{nil}$$

$$\mathtt{if\ expr\ do\ C} \vdash \sigma \to \sigma' \qquad \text{si } \mathsf{E}_{\Sigma,\sigma}[\![\mathtt{expr}]\!] \neq \mathtt{nil} \text{ et } \mathtt{C} \vdash \sigma \to \sigma'$$

Propriété 18. *Les programmes fuyants statiques sont calculables en temps constant, et ne sont donc pas Turing-complets. Les programmes fuyants auto-modifiants sont Turing-complets.*

Démonstration. Pour montrer que les programmes fuyants auto-modifiants sont Turing-complets, on traduit les programmes statiques normaux en programmes fuyants auto-modifiants. Pour cela, on définit le programme `Z` tel que :

$$[\![\texttt{Z}]\!] : \underline{\texttt{if E do C}} \rightarrow \underline{\texttt{if E do (C; q := Z(if E do C); eval(q))}}$$

Toute boucle `while E do C` peut alors se réécrire uniquement avec `if` et `Z` :

$$[\![\texttt{while E do C}]\!] = [\![\texttt{q := Z(}\underline{\texttt{if E do C}}\texttt{); eval(q)}]\!]$$

□

3.5 Transformation des programmes

3.5.1 Des statiques en auto-modifiants

Une opération courante pour la compression ou la protection des programmes consiste à transformer un programme statique quelconque en programme auto-modifiant. On s'intéresse donc d'abord à cette transformation, qui peut se faire de manière directe.

Propriété 19 (Obfuscation)**.** *Tout programme statique peut être transformé en un programme auto-modifiant équivalent par une fonction calculable en temps constant.*

Démonstration. En réutilisant le programme `proxy` (vu en section 3.3.2), et si on suppose que la fonction de spécialisation (ou fonction S-m-n) est calculable en temps constant par le programme `spec`, alors la fonction suivante satisfait la propriété :

$$\begin{aligned} f : \ &\textsc{Stat} \rightarrow \textsc{Smc} \\ &\texttt{p} \rightarrow [\![\texttt{spec}]\!](\texttt{proxy}, \ \texttt{p}) \end{aligned}$$

□

Les programmes auto-modifiants étant plus complexes à analyser que les programmes statiques, la transformation ci-dessus peut être vue comme une forme d'obfuscation. De manière intuitive, l'obfuscation est une transformation préservant la sémantique qui prend en entrée un programme dans une forme simple (ou claire) et qui renvoie un programme équivalent dans une forme complexe. On s'intéresse donc généralement à la transformation inverse, qui permet d'éclaircir (ou déobfusquer) les programmes.

3.5.2 Des auto-modifiants externes en statiques

On a défini l'auto-modification pour les programmes `WHILE` comme un appel externe à l'interpréteur. Tout programme `WHILE` peut donc être réécrit simplement en programme statique.

Propriété 20 (Déobfuscation pour `WHILE`)**.** *Pour le langage* ***WHILE****, il existe une fonction de compilation f calculable et totale qui vérifie :*

$$\begin{aligned} f : \ &\textsc{Progs} \to \textsc{Stat} \\ &\boldsymbol{p} \to \boldsymbol{q} \text{ tel que } [\![\boldsymbol{p}]\!]_{\textsc{Progs}} = [\![\boldsymbol{q}]\!]_{\textsc{Stat}} \end{aligned}$$

Démonstration. Comme le langage `WHILE` est Turing-complet, on a :

$$\forall g \in \Sigma \ \exists \mathtt{p}_g \in \texttt{WHILE-progs} \mid [\![\mathtt{p}_g]\!] = g$$

C'est en particulier vrai pour les fonctions `eval` de l'environnement.

Soit $\mathtt{p}_f$ un programme `WHILE` tel que :

$$\begin{aligned} [\![\mathtt{p}_f]\!] : \ &\texttt{WHILE-progs} \to \texttt{WHILE-progs} \\ &\mathtt{p} \to \mathtt{q} \end{aligned}$$

où `q` est le programme `p` où tous les appels à une fonction `g(X)` ont été remplacés par $\mathtt{p}_g$`(`$\mathtt{p}_f$`(X))` si g est une fonction `eval`. Par construction, on a donc $[\![\mathtt{q}]\!] = [\![\mathtt{p}]\!]$ et $\mathtt{q} \in \textsc{Stat}$.

La fonction f recherchée est donc calculable par le programme $\mathtt{p}_f$. □

3.5.3 Des auto-modifiants internes en statiques

Dans le cadre des programmes `Machine-GOTO`, nous rappelons dans un premier temps la définition d'une énumération acceptable des fonctions partielles récursives et le théorème d'isomorphisme de Rogers.

Définition 21 (Énumération acceptable)**.** Une suite $\varphi_0, \varphi_1, \ldots$ de fonctions partielles récursives définit une *énumération acceptable* [Rog67] si :

1. elle est Turing-complète, c'est-à-dire que pour toute fonction partielle calculable $f : \mathbb{N} \to \mathbb{N}_\perp$, il existe $p \in \mathbb{N}$ tel que $f = \varphi_p$
2. la fonction universelle *univ* est calculable, avec

$$\begin{aligned} \mathit{univ} : \mathbb{N} \times \mathbb{N} &\to \mathbb{N}_\perp \\ \mathtt{p}, x &\to \varphi_{\mathtt{p}}(x) \end{aligned}$$

3. pour tout m et pour tout n, il existe une fonction calculable $s_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ telle que pour tout indice `p` et pour toute entrée $(x_1, \ldots, x_m, y_1, \ldots, y_n) \in \mathbb{N}^{m+n}$

$$\varphi_{\mathtt{p}}^{m+n}(x_1, \ldots, x_m, y_1, \ldots, y_n) = \varphi^n_{s_n^m(\mathtt{p}, x_1, \ldots, x_m)}(y_1, \ldots, y_n)$$

Théorème 22 (Isomorphisme de Rogers [Rog67])**.** *Pour tous langages* `L` *et* `M` *définissant une énumération acceptable des fonctions partielles récursives, il existe une fonction totale, calculable et bijective* f *telle que :*

$$f : \texttt{L-progs} \to \texttt{M-progs}$$
$$\texttt{p} \to \texttt{q} \text{ tel que } [\![\texttt{p}]\!]_L = [\![f(\texttt{q})]\!]_M$$

Le langage `Machine-GOTO` (comprenant les programmes auto-modifiants) est une extension du langage `GOTO` dont Jones a montré qu'il définissait une énumération acceptable des fonctions partielles récursives. Le langage `GOTO` correspond à l'ensemble Stat, qui définit donc également une énumération acceptable. En appliquant le théorème d'isomorphisme de Rogers, nous avons donc la propriété suivante.

Propriété 23 (Déobfuscation pour `Machine-GOTO`)**.** *Pour le langage* `Machine-GOTO`*, il existe une fonction de compilation* f *calculable et totale qui vérifie :*

$$f : \textsc{Progs} \to \textsc{Stat}$$
$$\texttt{p} \to \texttt{q} \text{ tel que } [\![\texttt{p}]\!]_{\textsc{Progs}} = [\![\texttt{q}]\!]_{\textsc{Stat}}$$

Remarque 24. La propriété ci-dessus a pour conséquence que tout programme auto-modifiant ou conditionnellement auto-modifiant peut être transformé en un programme statique équivalent. On peut donc imaginer la restriction de la fonction de compilation f à Smc comme une fonction de déobfuscation.

Cette approche a toutefois plusieurs limitations, car :
- la complexité de f est inconnue ;
- la complexité de $f(\texttt{p})$ par rapport à celle de $\texttt{p}$ est inconnue ;
- pour $\texttt{p} \in \textsc{Stat}$, nous n'avons pas nécessairement $f(\texttt{p}) = \texttt{p}$.

Ces deux derniers points sont particulièrement importants car ils vont à l'encontre de la notion classique de déobfuscation. En particulier, $\texttt{p}$ pourrait être calculable en temps polynomial et $f(\texttt{p})$ en temps exponentiel, ce qui n'est pas souhaitable pour des raisons évidentes. On pourrait également avoir une forme $f(\texttt{p})$ beaucoup plus grande et complexe que $\texttt{p}$ pour les programmes "normaux" de Stat, or la définition intuitive d'une fonction de déobfuscation est qu'elle doit rendre les programmes plus simples.

En abandonnant la contrainte selon laquelle f doit être bijective dans le théorème d'isomorphisme de Rogers, on peut obtenir de manière constructive une fonction de déobfuscation plus efficace.

Propriété 25. *Il existe une fonction de compilation* $\mathsf{D} : \textsc{Smc} \cup \textsc{CondSmc} \to \textsc{Stat}$ *telle que :*
- D *est calculable en temps polynomial ;*
- $temps(\mathsf{D}(\texttt{p}),\ x) = O(temps(\texttt{p},\ x))$.

Démonstration. On note `Int` un auto-interpréteur de `WHILE` (resp. `Machine-GOTO`). On suppose que cet interpréteur provoque un ralentissement linéaire des programmes, c'est-à-dire :

$$\exists M \forall x \ temps(\texttt{Int}, \ (\texttt{p}, x)) \leq M \times temps(\texttt{p}, \ x)$$

En appliquant la première projection de Futamura [Fut99], on obtient la fonction de compilation souhaitée :

$$\mathsf{D} : \texttt{p} \rightarrow [\![\texttt{spec}]\!](\texttt{Int}, \ \texttt{p})$$

□

Remarque 26. Le résultat précédent signifie que l'on peut obtenir avec des briques simples (un auto-interpréteur du langage considéré et un spécialiseur) une fonction de déobfuscation « efficace » au sens de sa complexité et de la complexité du programme obtenu. Il faut toutefois relativiser ce résultat car, bien que le programme obtenu soit statique, sa sémantique dépend toujours de l'interprétation du programme auto-modifiant original.

3.6 Conclusion

Nous avons défini deux formes d'auto-modifications, internes et externes, selon le modèle de calcul considéré. Nous avons ensuite discuté l'idée selon laquelle il s'agissait en réalité de deux expressions équivalentes d'un même phénomène : l'exécution par un programme d'une partie de sa sortie.

Nous nous sommes enfin intéressés aux propriétés fondamentales du phénomène au coeur de l'auto-modification. En particulier, nous avons montré :

- qu'il était indécidable de détecter si un programme était auto-modifiant ou statique sur une entrée (théorème 16) ;
- qu'une classe particulière de programmes, les programmes fuyants, ne calculaient pas le même ensemble de fonctions selon qu'on leur autorise d'être auto-modifiants ou non (propriété 18) ;
- qu'il existait des fonctions calculables conservant la sémantique permettant de passer des programmes statiques aux auto-modifiants et vice-versa (propriétés 19, 20 et 23).

4 Analyse fonctionnelle des programmes auto-modifiants

Dans les chapitres précédents, nous nous sommes intéressés aux fonctions calculées par les programmes. Nous avons donc uniquement regardé leurs entrées et sorties, en faisant abstraction de toutes les étapes de calcul intermédiaires et de l'environnement. Ce chapitre est consacré aux interactions entre les programmes et l'environnement, qui permettent de définir la notion de *comportement* d'un programme et de *politique* sur ces comportements.

Nous chercherons ensuite à définir un moyen d'appliquer des politiques de sécurité à des programmes inconnus. Nous nous intéresserons notamment à l'instrumentation, qui fonctionne par réécriture. Cette approche a l'avantage de ne pas nécessiter la coopération du programme original, elle ne requiert pas de modifier l'environnement, et s'applique de manière élégante aux programmes auto-modifiants.

Une partie de ce chapitre a été publiée dans [Rey10].

4.1 Comportement des programmes et politiques de sécurité

4.1.1 Définition du comportement des programmes

L'exécution d'un programme $\mathtt{p}$ sur une entrée x (c'est-à-dire sur une valuation $\sigma_0 = [\mathtt{X} \to x]$) est la suite de commandes

$$[\![\mathtt{p}]\!](x) \equiv ((\mathtt{C}_i,\ \sigma_i))_{i\in\mathbb{N}} \text{ si } \mathtt{C}_i \vdash \sigma_i \to \sigma_{i+1}$$

Définition 27. Le *comportement de $\mathtt{p}$ sur x* est la suite des appels système observés pendant l'exécution de $\mathtt{p}$ sur x. On l'obtient en faisant abstraction des commandes qui ne sont pas des appels système. On utilise pour cela une fonction d'abstraction $\alpha : \mathbb{N} \to \mathbb{N}$ strictement croissante telle que :

$$\mathsf{Comp}(\mathtt{p}, x) = ((\mathtt{A}_i,\ y_i))_{i\in\mathbb{N}} \text{ si } \mathtt{C}_{\alpha(i)} = \mathtt{A}_i\mathtt{(Y)} \text{ ou } \mathtt{C}_{\alpha(i)} = (\mathtt{Z\ :=\ A}_i\mathtt{(Y)})$$
$$\text{avec } \mathtt{A}_i \in \Sigma \text{ et } \sigma_{\alpha(i)}(\mathtt{Y}) = y_i$$

Le *comportement de $\mathtt{p}$* est la fonction $\mathsf{Comp}(\mathtt{p}) = \lambda x.\mathsf{Comp}(\mathtt{p}, x)$

Propriété 28. *Si on fait les hypothèses suivantes :*

1. *le système est dans un état pré-existant κ,*
2. *l'exécution de $\mathtt{p}$ sur x dans l'état κ du système conduit à l'état κ',*
3. *seuls les appels à Σ permettent de modifier l'état du système,*

alors la transition $\|\mathtt{p}, x\|(\kappa) = \kappa'$ est entièrement déterminée par $\mathsf{Comp}(\mathtt{p}, x)$.

La définition et la propriété ci-dessus permettent de modéliser l'architecture des systèmes informatiques courants, qui séparent généralement les programmes de niveau système (considérés comme de confiance) et de niveau utilisateur (considérés comme potentiellement malveillants). En particulier, cette modélisation met en évidence le fait que si l'on considère que la sécurité du système ne dépend que de sa configuration κ, alors les seules actions sensibles (voir section 1.3) réalisables par les programmes sont celles qui peuvent modifier l'état de ce système, c'est-à-dire les appels à Σ.

4.1.2 Définition des politiques de sécurité

Définition 29. Une *politique de sécurité* est un prédicat φ sur la configuration du système et les comportements des programmes : étant donnés un état initial κ considéré comme sûr et un programme $\mathtt{p}$ tel que $\|\mathtt{p}, x\|(\kappa) = \kappa'$, le prédicat $\varphi(\mathtt{p}, x)$ sera vrai si et seulement si l'état final κ' est également considéré comme sûr.

On notera $\varphi(\mathtt{p})$ ssi $\forall x\ \varphi(\mathtt{p},\ x)$. D'autre part, φ définit les ensembles suivants :

$$\textsc{Safe}_\varphi = \{\mathtt{p} \mid \varphi(\mathtt{p})\}$$
$$\textsc{Unsafe}_\varphi = \{\mathtt{p} \mid \neg\varphi(\mathtt{p})\}$$

Remarque 30. Dans la définition 29, la vérité de $\varphi(\mathtt{p},\ x)$ dépend de l'état final du système κ'. On peut donc imaginer que les seuls comportements malveillants modélisés sont ceux qui modifient activement l'état du système, comme par exemple un programme nuisible empêchant le système de démarrer. En réalité, l'intégralité de la suite $\mathsf{Comp}(\mathtt{p}, x)$ pourrait intervenir dans le calcul de $\varphi(\mathtt{p},\ x)$, ce qui permettrait également de prendre en compte des comportements plus passifs, comme par exemple de la fuite d'information.

Remarque 31. La suite $\mathsf{Comp}(\mathtt{p}, x)$ conserve l'ordre dans lequel les appels sont effectués. On peut donc imaginer des politiques qui n'interdisent pas des appels individuels (des actions atomiques) mais des suites d'appels. Par exemple, une politique pourrait autoriser l'écriture sur un réseau public puis la lecture de données privées, mais pas l'inverse. Cette politique garantirait que les données privées ne peuvent pas être envoyées sur le réseau public. En généralisant cette notion de suite d'appels, Kinder et al. [KKSV05] ont montré que si φ était une formule en CTL [CES86], alors on pouvait également définir une politique comme un chemin dans un graphe d'appels.

Définition 32. Une politique de sécurité *triviale* est une politique φ telle que :

$$\forall \mathtt{p}\ \varphi(\mathtt{p}) \text{ ou } \forall \mathtt{p}\ \neg\varphi(\mathtt{p})$$

Pour une telle politique, Safe (resp. Unsafe) est soit vide soit égal à l'ensemble des programmes.

Définition 33. Une politique de sécurité *extensionnelle* est une politique φ telle que :

$$\forall \mathtt{p}, \mathtt{q}\ \mathsf{Comp}(\mathtt{p}) = \mathsf{Comp}(\mathtt{q}) \implies \varphi(\mathtt{p}) = \varphi(\mathtt{q})$$

4.1.3 Application aux programmes malveillants

Nous pouvons maintenant reformuler la définition informelle des programmes malveillants donnée en section 1.3 avec la notion plus précise de politique de sécurité, permettant de modéliser les attentes de l'utilisateur.

Définition 34. Un *programme malveillant* (ou malware) pour la politique de sécurité φ et l'observateur $\mathcal{O}$ donnés est un programme $\mathtt{p}$ tel que :

- $\exists x\ \neg\varphi(\mathtt{p}, x)$
- $\mathcal{O}$ juge que $\mathsf{Comp}(\mathtt{p}, x)$ est intentionnel et en violation des principes d'information et de contrôle (voir définition 2).

On note $\textsc{Malware}_{\mathcal{O},\varphi}$ l'ensemble des malwares pour un observateur et une politique donnés.

Remarque 35. On déduit de la définition 34 que :

$$\textsc{Malware}_{\mathcal{O},\varphi} \subseteq \textsc{Unsafe}_{\varphi}$$

Comme $\textsc{Malware}_{\mathcal{O},\varphi}$ dépend de φ mais également de $\mathcal{O}$, il est impossible de décider de manière automatique pour $\mathtt{p}$ quelconque si $\mathtt{p} \in \textsc{Malware}_{\mathcal{O},\varphi}$. Une approximation courante est d'interdire tous les programmes de $\textsc{Unsafe}_{\varphi}$, comprenant notamment l'ensemble des malwares.

4.2 Conformité à une politique

4.2.1 Indécidabilité de la détection

Nous avons vu qu'une politique de sécurité permet de définir des sous-ensembles de programmes vérifiant certaines propriétés, comme $\textsc{Safe}_{\varphi}$ et $\textsc{Unsafe}_{\varphi}$. Nous allons voir que seules les politiques de sécurité triviales sont décidables.

Propriété 36. *Toute politique de sécurité non-triviale extensionnelle est indécidable.*

Démonstration. La démonstration de cette propriété est basée sur la démonstration du théorème de Rice [Ric53]. Soit φ une politique non-triviale extensionnelle, nous allons montrer que si φ est décidable, alors le problème de l'arrêt est décidable.

Soit un programme $\mathtt{b}$ et $e \in \mathbb{D}$ tels que $\mathsf{Comp}(\mathtt{b}, e) = \emptyset$. On suppose pour la suite que $\varphi(\mathtt{b})$.

Comme φ est non-triviale, il existe un programme $\mathtt{c}$ tel que $\neg\varphi(\mathtt{c})$.

Soit un programme $\mathtt{p}$ tel que $\mathsf{Comp}(\mathtt{p}, e) = \emptyset$. On construit un programme $\mathtt{q}$ (en langage `WHILE`, la construction serait similaire en `Machine-GOTO`) :

```
q =
    X := read();
    resultp := p(e);
    resultc := c(X);
    write(resultc);
```

On distingue alors deux cas, selon que le programme $\mathtt{p}$ s'arrête sur l'entrée e ou non.

- si $\mathtt{p}$ ne s'arrête pas sur e, alors on a $\mathsf{Comp}(\mathtt{q}) = \lambda x.\emptyset$. En particulier, $\mathsf{Comp}(\mathtt{q}) = \mathsf{Comp}(\mathtt{b})$, donc par extensionnalité de φ on a $\varphi(\mathtt{q})$
- si $\mathtt{p}$ s'arrête sur e, alors on a $\mathsf{Comp}(\mathtt{q}) = \mathsf{Comp}(\mathtt{c})$. Or $\neg\varphi(\mathtt{c})$, donc par extensionnalité de φ on a $\neg\varphi(\mathtt{q})$

Nous venons de montrer que $\varphi(\mathtt{q}) \iff \mathtt{p}$ s'arrête sur e, donc si φ était décidable alors le problème de l'arrêt serait décidable.

Nous avons supposé que $\varphi(\mathtt{b})$ était vrai. On applique le même raisonnement à la propriété à la propriété $\neg\varphi$ dans le cas $\neg\varphi(\mathtt{b})$. On aboutit dans les deux cas à la contradiction "problème de l'arrêt décidable", donc l'hypothèse "φ décidable" est fausse. □

Remarque 37. En particulier, le problème de la détection virale peut se formuler comme une politique de sécurité. Soient $\mathtt{p}$ et x tels que $[\![\mathtt{p}]\!](x) \equiv (\mathtt{C}_i,\ \sigma_i)_{i\in\mathbb{N}}$. La politique de détection virale φ_v est définie ainsi :

$$\varphi_v(\mathtt{p},\ x) \iff \forall i\ \mathtt{C}_i \neq \mathtt{write(Y)} \text{ avec } \sigma_i(\mathtt{Y}) = \underline{\mathtt{p}}$$

$\textsc{Unsafe}_{\varphi_v}$ est exactement l'ensemble des programmes qui sont auto-reproducteurs sur au moins une entrée. Or φ_v est une politique non-triviale et extensionnelle, donc elle est indécidable. Ce résultat est semblable au résultat montré par Cohen [Coh86] sans la notion de mutation, on peut donc voir les virus comme un cas particulier des programmes malveillants. Le même résultat d'indécidabilité s'applique au problème plus général des malwares : pour toute définition non-triviale d'un programme malveillant, la détection est indécidable.

4.2.2 Approximations décidables

Nous avons vu que seules les politiques triviales étaient décidables pour des programmes quelconques. Il est donc nécessaire d'effectuer des approximations afin de pouvoir rendre les politiques de sécurité décidables. Il existe de nombreuses approches permettant de décider les politiques dans certains cas, par exemple :

- les approches prédictives (ou statiques) comme les codes porteurs de preuves (ou *proof-carrying code* [Nec97]) ou la vérification de contraintes sur le programme avec [Mye99] ou sans annotations [YSD+09] ;
- les approches dynamiques, permettant de faire tourner des programmes dans un bac à sable par réécriture [FC08, KBA02], interposition [GPR04] ou interprétation [LY99].

Les approches prédictives cherchent à déterminer à l'avance si les programmes vérifient la politique de sécurité, ce qui implique de se limiter au sous-ensemble des programmes pour lesquels c'est possible. À l'inverse, les approches dynamiques vérifient en temps réel si la politique est respectée et lèvent une erreur si ce n'est pas le cas.

Les programmes auto-modifiants posent plus particulièrement problème aux approches prédictives. Nous allons mettre en évidence ce phénomène avec une approche courante qui consiste à déterminer statiquement les symboles utilisés par le programme.

Définition 38. Pour $\mathtt{p} \in$ Progs, on définit la fonction AppAtt de la manière suivante :

$$\mathsf{AppAtt}(\mathtt{p}) = \{\mathtt{Call} \mid \exists x, j \text{ tel que } \mathsf{Comp}(\mathtt{p}, x) = ((\mathtt{C}_i,\ y_i))_{i\in\mathbb{N}} \text{ et } \mathtt{C}_j = \mathtt{Call}\}$$

De manière intuitive, $\mathsf{AppAtt}(\mathtt{p})$ est l'ensemble des appels atteignables de $\mathtt{p}$, c'est-à-dire pour lesquels il existe une entrée conduisant à leur exécution. On en déduit que $\mathsf{AppAtt}(\mathtt{p}) \subseteq \Sigma$ est semi-décidable.

Définition 39. On définit la fonction Symb de la manière suivante, pour tout $\mathtt{p} = \mathtt{C}_1\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{C}_n$:

$$\mathsf{Symb} : \mathtt{p} \to \{\mathtt{Call} \mid \exists i\ \mathtt{C}_i = \mathtt{Call(X)} \text{ ou } \mathtt{C}_i = (\mathtt{Y\ :=\ Call(X)})\}$$

De manière intuitive, $\mathsf{Symb}(\mathtt{p})$ est l'ensemble des symboles de Σ qui apparaissent dans la syntaxe de $\mathtt{p}$. La fonction Symb est clairement calculable et totale, et on a $\mathsf{Symb}(\mathtt{p}) \subseteq \Sigma$.

Remarque 40. Il existe des programmes pour lesquels $\mathsf{AppAtt}(\mathtt{p})$ est inclus strictement dans $\mathsf{Symb}(\mathtt{p})$. Par exemple soit $\mathtt{p}$ le programme défini de la manière suivante :

```
p =
    while false do write(nil);
    stop
```

On a $\mathsf{Symb}(\mathtt{p}) = \{\mathtt{write}, \mathtt{stop}\}$ et $\mathsf{AppAtt}(\mathtt{p}) = \{\mathtt{stop}\}$ car il n'existe pas d'entrée permettant d'exhiber le comportement (`write`, `nil`).

Propriété 41. *p statique* $\implies \mathsf{AppAtt}(p) \subseteq \mathsf{Symb}(p)$

La propriété ci-dessus a pour conséquence que Symb est une approximation calculable de AppAtt pour les programmes statiques. Cela a des conséquences pratiques, par exemple le système Google Native Client [YSD+09] définit un ensemble de symboles interdits $I \subseteq \Sigma$ et utilise la propriété suivante :

$$\mathtt{p} \text{ statique et } I \cap \mathsf{Symb}(\mathtt{p}) = \emptyset \implies I \cap \mathsf{AppAtt}(\mathtt{p}) = \emptyset$$

Cette propriété se traduit de la manière suivante : si `p` est statique et qu'il ne comporte pas de symboles interdits, alors il n'existe aucune entrée lui permettant d'effectuer des appels interdits.

4.3 Analyse par instrumentation

4.3.1 Définition

Dans le cas général, les programmes sont potentiellement auto-modifiants. Déterminer $\mathsf{Symb}(\mathtt{p})$ n'est donc pas suffisant pour l'analyse. Se placer au niveau de l'interpréteur permet de voir les commandes passées sans ambiguïté, mais ce n'est pas toujours possible :

- les programmes dits natifs sont interprétés directement par un processeur matériel, que l'on ne peut donc pas modifier ;
- les architectures matérielles de type CISC (pour *Complex Instruction Set Computer*[1]) sont notoirement difficiles à implémenter à l'identique sous forme logicielle de par leur complexité ;
- la documentation de certaines architectures matérielles est incomplète (c'est le cas de l'architecture x86).

Pour pallier à ces problèmes, il est possible d'analyser les programmes par instrumentation, ou réécriture dynamique de code. Une fonction d'instrumentation peut être vue comme une fonction de compilation

$$[\![\mathtt{Instr}]\!] : \textsc{Progs} \to \textsc{Progs}$$
$$\mathtt{p} \to \mathtt{p'}$$

Cette fonction garantit pour tout `p` et pour une politique donnée φ :

- un *contrôle* sur le comportement de `p'`, qui se traduit par $\varphi(\mathtt{p'})$;
- un certain niveau de *transparence*, c'est-à-dire $\varphi(\mathtt{p}, x) \implies [\![\mathtt{p}]\!](x) = [\![\mathtt{p'}]\!](x)$

En particulier, $\varphi(\mathtt{p}) \implies [\![\mathtt{p}]\!] = [\![\mathtt{p'}]\!]$. Cela signifie que si le programme original respecte la politique φ, alors la sémantique est préservée par instrumentation.

1. ordinateur à jeu d'instruction complexe

4.3.2 Problèmes liés à la transparence

La contrainte de transparence est très difficile à atteindre sur des machines concrètes à mémoire finie. Typiquement, le programme d'instrumentation va faire des allocations dans l'espace mémoire du programme à instrumenter, donc certaines allocations du programme original risquent d'échouer. D'autre part, certains programmes peuvent employer des techniques d'introspection pour vérifier leur intégrité, ce qui signifie que le programme d'instrumentation doit résister à des programmes qui cherchent à le détecter activement.

Par exemple, le programme suivant en `Machine-GOTO` utilise de la vérification d'intégrité :

```
self_check =
    l1:    @100 := l1
    l2:    if equals (cons l1 d) goto exit2
    exit1: write 'error'
    exit2: write 'success'
```

Si on suppose que `equals` est un programme en `Machine-GOTO` tel que :

$$\forall x \ [\![\texttt{equals}]\!](x) = \begin{cases} \texttt{false} & \text{si } x = \texttt{nil} \\ \texttt{false} & \text{si } x = \texttt{cons}(d1, d2) \text{ et } d1 \neq d2 \\ \texttt{true} & \text{si } x = \texttt{cons}(d1, d2) \text{ et } d1 = d2 \end{cases}$$

Alors le programme `self_check` renvoie `'success'` si l'instruction pointée par `l1` est différente d'une valeur donnée `d`, et renvoie `'error'` sinon. Pour un programme d'instrumentation naïf qui changerait l'instruction pointée par `l1` et ne changerait pas la valeur lue par le programme, on aurait

$$[\![\texttt{self_check}]\!](\texttt{nil}) = \texttt{'success'}$$

$$\text{et } [\![[\![\texttt{Instr}]\!](\texttt{self_check})]\!](\texttt{nil}) = \texttt{'error'}$$

Le problème de la transparence est crucial pour l'analyse de programmes malveillants, car il faut s'assurer que le comportement du programme dans l'environnement d'analyse est identique au comportement qu'il aurait dans un environnement normal. Nous verrons dans le chapitre 7 que de nombreux programmes malveillants réels emploient des techniques d'introspection et modifient leur comportement s'ils détectent l'environnement d'analyse [RKK07, PMR+09]. Heureusement, il existe des programmes d'instrumentation concrets qui fournissent un bon niveau de transparence comme Pin [LCM+05], Valgrind [NS03] et DynamoRIO [Bru04].

Par la suite, nous utiliserons Pin comme outil sous-jacent pour réaliser des expériences sur des programmes malveillants. Même s'il existe de nombreux moyens de détecter spécifiquement Pin [Mus09], une étude [MPRB09] a montré que sa transparence par rapport à la sémantique des instructions x86 était meilleure que celle de certains interpréteurs utilisés en production comme QEMU [Bel05] et Bochs [Law96].

4.3.3 Construction d'un programme d'instrumentation

Pour illustrer l'instrumentation des programmes auto-modifiants, nous allons montrer comment appliquer des politiques de sécurité par réécriture aux programmes auto-modifiants. Pour cet exemple, nous nous restreindrons aux programmes auto-modifiants externes, plus simples à traiter.

Instrumentation des programmes statiques

On se donne une politique de sécurité φ qui interdit les appels dans un sous-ensemble $I \subseteq \Sigma$. Pour tout programme $\mathtt{p} \in \textsc{Safe}_\varphi$, on a donc $\mathsf{AppAtt}(\mathtt{p}) \cap I = \emptyset$. Si on ne s'intéresse qu'au contrôle et pas à la transparence, il suffit de réécrire tout programme de la manière suivante (en pseudo-code) :

```
fonction static_rewrite(p) {
    pour Ci dans p = C1; ...; Cn
        dans Ci, remplacer Call(X) par Error si Call dans I
    renvoyer p
}
```

Si $\mathtt{eval} \notin I$, ce programme ne fonctionne que sur les programmes statiques. En effet, $\forall \mathtt{p}\ \mathsf{Symb}([\![\mathtt{static_rewrite}]\!](\mathtt{p})) \cap I = \emptyset$, donc d'après la propriété 41 on a le résultat souhaité :

$$[\![\mathtt{static_rewrite}]\!](\textsc{Stat}) \subseteq \textsc{Safe}_\varphi$$

Instrumentation des programmes auto-modifiants

Une fonctionnalité essentielle d'un programme d'instrumentation est la gestion efficace des programmes auto-modifiants. On part du constat que les programmes auto-modifiants externes sont en réalité des suites de programmes statiques, chaque élément de la suite appelant le suivant par `eval`. Afin de les gérer, on définit donc une analyse statique qui consiste à neutraliser les appels sensibles et interposer un appel à l'analyse statique sur l'argument du prochain `eval`. Si on appelle le programme d'instrumentation `dynamic_rewrite`, sa gestion des programmes auto-modifiants peut être résumée avec la règle de réécriture suivante :

$$\mathtt{eval(p)} \rightarrow \mathtt{eval(dynamic_rewrite(p))}$$

Le fonctionnement complet de l'instrumentation permettant de gérer à la fois les programmes statiques et auto-modifiants est illustré par le programme en pseudo-code suivant :

```
fonction dynamic_rewrite(p) {
    q = static_rewrite(p)
    pour Ci dans q = C1; ...; Cn
```

```
        dans Ci, remplacer eval(X) par eval(dynamic_rewrite(X))
    renvoyer q
}
```

Cette fois, on a la propriété

$$[\![\texttt{dynamic_rewrite}]\!](\textsc{Progs}) \subseteq \textsc{Safe}_\varphi$$

Auto-instrumentation

La version de `dynamic_rewrite` ci-dessus est incapable de s'auto-instrumenter correctement, car $[\![\texttt{dynamic_rewrite}]\!](\texttt{dynamic_rewrite})$ est un programme récursif qui ne s'arrête jamais. Si on désigne par $\texttt{p}^n$ le programme défini par récurrence :

- $\texttt{p}^1 = \texttt{p}$
- pour $n \geq 1$, $\texttt{p}^{n+1} = [\![\texttt{p}^n]\!](\texttt{p})$

Alors on a :

$$[\![\texttt{dynamic_rewrite}^2]\!] : \mathbb{D} \to \mathbb{D} \cup \{\bot\}$$
$$d \to \bot$$

La nouvelle version, capable de s'auto-instrumenter, est la suivante :

```
fonction self_rewrite(p) {
    q = static_rewrite(p)
    pour Ci dans q = C1; ...; Cn
        dans Ci, remplacer:
             * self_rewrite par self_rewrite_again
             * eval(X) par eval(self_rewrite(X))
    renvoyer q
}
```

Cette fois, `self_rewrite` vérifie la propriété suivante :

$$\forall n \geq 1\ [\![\texttt{self_rewrite}^n]\!](\textsc{Progs}) \subseteq \textsc{Safe}_\varphi$$

Application à JavaScript

Nous avons réalisé un prototype de programme d'instrumentation permettant d'analyser des programmes malveillants auto-modifiants en JavaScript. Ce prototype, dont la structure est très semblable à celle de `self_rewrite`, a été décrit plus précisément dans [Rey10] et est disponible sur Internet sous licence libre, à l'adresse suivante :

http://code.google.com/p/cremebrulee/

Il permet de gérer les actions sensibles couramment effectuées par les scripts malveillants que l'on retrouve sur des sites de distribution de malwares ou sur des sites piratés. Par exemple :

- l'injection d'un document invisible dans la page courante (une `iframe`) ;
- l'insertion de nouveaux scripts, d'images ou d'objets ActiveX ;
- la redirection du navigateur vers une nouvelle adresse.

4.4 Conclusion

Dans le chapitre 2, nous avons introduit des langages avec la notion d'environnement et d'appel système. Cet environnement prend ici toute sa place car on définit le comportement d'un programme comme ses interactions avec l'environnement. On en déduit une définition naturelle des politiques de sécurité comme des procédures de décision sur les comportements, puis une définition des programmes malveillants comme des programmes capables de violer intentionnellement une politique de sécurité.

Nous avons également montré que le théorème classique d'indécidabilité de la détection virale pouvait se généraliser sous forme d'un problème d'application de politique. Cela permet de montrer que l'auto-reproduction n'est pas au coeur de l'indécidabilité : quelle que soit la manière non-triviale dont on définit un comportement indésirable, déterminer si un programme quelconque implémente ce comportement est indécidable.

Enfin, nous nous sommes intéressés à l'analyse par instrumentation, particulièrement adaptée aux programmes auto-modifiants. Nous avons construit un modèle théorique de programme d'instrumentation capable de s'auto-instrumenter, et nous l'avons appliqué à l'analyse de programmes malveillants auto-modifiants en JavaScript.

Analyse opérationnelle

5 Typage dynamique et comportements auto-modifiants

Ce chapitre est consacré à l'analyse dynamique des programmes auto-modifiants en code machine. La plupart des programmes malveillants actuels étant écrits pour l'architecture x86, dont `Machine-GOTO` fournit un modèle sans pointeurs, il est nécessaire de s'intéresser au code machine afin d'obtenir des résultats expérimentaux. Nous proposons une analyse basée sur un typage dynamique de la mémoire permettant de détecter différents comportements auto-modifiants, ainsi que des exemples d'applications (chapitre 6) et une validation par l'expérience de son efficacité sur différents types de programmes auto-modifiants, en particulier des programmes malveillants (chapitre 7).

Une partie de ce chapitre a été publiée dans [GMR09].

5.1 Exemple en x86

Nous allons illustrer le principe du typage dynamique de la mémoire avec un exemple en x86 extrait du virus Parite.B [1]. Ce virus utilise une boucle simple afin de se déchiffrer en mémoire puis transfère le contrôle à la zone écrite. La boucle de déchiffrement fonctionne de la manière suivante (nous utilisons des symboles comme `$index` et `$key` au lieu des valeurs concrètes afin de faciliter la compréhension) :

```
@a: mov esi, $index
@b: xor [@offset + esi], $key
@c: sub esi, 4
@d: jnz @b
@offset: [encrypted data]
```

Le principe de l'analyse que nous proposons est d'attribuer à chaque adresse mémoire un niveau de lecture, d'écriture et d'exécution. Nous procédons de la manière suivante :

1. L'instruction à l'adresse `@a` est exécutée, elle reçoit un niveau d'exécution de 1.
2. De même, l'instruction à l'adresse `@b` reçoit un niveau d'exécution de 1. Comme elle lit et écrit le contenu de l'adresse `@offset + esi`, cette adresse reçoit un niveau de lecture et d'écriture de 1.

1. http://www.eset.com/threat-center/encyclopedia/threats/pariteb

3. On procède de la même manière pour les instructions `@c` et `@d`, qui reçoivent un niveau d'exécution de 1.

4. L'instruction à l'adresse `@d` est un saut conditionnel. Si la condition est vérifiée, alors on répète les opérations de la boucle.

5. Lorsque la boucle est terminée (c'est-à-dire que la condition n'est plus vérifiée), le contrôle est transféré à l'adresse `@offset`. Comme cette adresse a un niveau d'écriture de 1 (donné par l'instruction `@b`), elle obtient alors un niveau d'exécution de 2, nous sommes en présence de code auto-modifiant.

À la fin de l'exécution, nous constatons les propriétés suivantes :
- les adresses mémoire avec un niveau d'exécution de 0 n'ont jamais été exécutées ;
- les adresses de niveau 1 ont été exécutées un nombre arbitraire de fois, et étaient dans l'image originale du programme σ_0^p (voir section 2.2.4) ;
- les adresses de niveau 2 ou plus ont été générées par le programme lui même, et n'étaient donc pas dans son image originale ;
- plus particulièrement, on peut suivre l'évolution du programme : toute instruction de niveau $n+1$ a été générée par une instruction de niveau n.

Nous allons voir dans les sections suivantes les règles précises permettant de typer la mémoire, et les informations que nous pouvons en tirer.

5.2 Typage dynamique de la mémoire

5.2.1 Définition préliminaire

Définition 42. Pour une mémoire μ et une commande ip: `C` données, on définit
- $Code_\mu(\text{ip: } \texttt{C})$ l'ensemble des adresses faisant partie de l'encodage de la commande `C`, c'est-à-dire :

$$Code_\mu(\text{ip: } \texttt{C}) = \{n \mid \text{ip} \leq n < \text{ip} + |enc(\underline{\texttt{C}})|\}$$

- $In_\mu(\texttt{C})$ l'ensemble des adresses lues par la commande `C` dans la mémoire μ ;
- $Out_\mu(\texttt{C})$ l'ensemble des adresses écrites par la commande `C` dans la mémoire μ.

Pour le langage `Machine-GOTO`, il est nécessaire de savoir dans quel environnement μ la commande est exécutée pour déterminer les ensembles *In* et *Out*. En effet, la taille d'une lecture/écriture en mémoire dépend de la taille des opérandes, alors que dans la plupart des langages machines comme x86 la taille d'une lecture/écriture ne dépend que de l'instruction et pas de l'environnement.

5.2.2 Système de typage classique

On se munit d'un environnement de typage Γ qui associe un niveau de lecture (R), d'écriture (W) et d'exécution (X) à chaque adresse mémoire m :

$$\begin{aligned}\Gamma : \mathbb{N} &\to \mathbb{N}^3 \\ m &\to (k_r, k_w, k_x)\end{aligned}$$

Si $\Gamma(m) = (k_r, k_w, k_x)$, on dit que m a un niveau de lecture (resp. écriture, exécution) de k_r (resp. k_w, k_x) par rapport à Γ et on écrit $\Gamma \vdash m : R(k_r)$ (resp. $\Gamma \vdash m : W(k_w)$, $\Gamma \vdash m : X(k_x)$).

Définition 43 (Niveau d'une instruction)**.** On note k_w le niveau d'écriture de l'instruction courante. On le définit comme le niveau d'écriture maximal des adresses qui la composent :

$$k_w = \max_{m \in Code_\mu(\text{ip: C})} \{k \mid \Gamma \vdash m : W(k)\}$$

Remarque 44. Cette définition de k_w permet de considérer les instructions comme des atomes d'exécution. Ainsi, même si les adresses qui composent une instruction ont des niveaux d'écriture différents, on prendra le maximum de ces niveaux que l'on considérera comme l'unique niveau de l'instruction.

Nous étendons ensuite la relation d'exécution des commandes de `Machine-GOTO` définie en section 2.2.4 avec l'environnement de typage. La nouvelle relation s'écrit $\texttt{C} \vdash (\text{ip}, \mu, \Gamma) \to (\text{ip}', \mu', \Gamma')$, et on obtient Γ' à partir de Γ en appliquant les règles d'exécution, de lecture et d'écriture.

Définition 45 (Règle d'exécution)**.**

$$\frac{m \in Code_\mu(\text{ip: C})}{\Gamma' \vdash m : X(k_w + 1)}$$

Une adresse exécutée a un niveau égal au niveau d'écriture de l'instruction à laquelle elle appartient plus 1.

Définition 46 (Règle de lecture)**.**

$$\frac{m \in In_\mu(\texttt{C})}{\Gamma' \vdash m : R(k_w + 1)}$$

Une adresse lue par une instruction de niveau $k_w + 1$ a un niveau de lecture de $k_w + 1$.

Définition 47 (Règle d'écriture)**.**

$$\frac{m \in Out_\mu(\texttt{C})}{\Gamma' \vdash m : W(k_w + 1)}$$

Une adresse écrite par une instruction de niveau $k_w + 1$ a un niveau d'écriture de $k_w + 1$.

Remarque 48. La règle d'exécution est toujours appliquée, ce qui permet de ne pas avoir de cas particulier pour les transferts de contrôle. Dans le cas x86, cela permet notamment de prendre en compte les sauts directs et indirects mais également les transferts asynchrones (comme les exceptions) et les sauts implicites par transfert à l'instruction suivante : quelle que soit la manière d'arriver à une commande, la même règle s'applique.

Remarque 49. Pour certaines commandes, aucune des règles de lecture et d'écriture ne s'appliquent. Par exemple, la commande `nil` est une commande valide qui ne fait rien, et en particulier qui ne lit pas et n'écrit pas dans la mémoire. À l'inverse, des commandes comme `m := cons(m, m)` provoquent à la fois une lecture et une écriture de l'adresse `m` (on applique donc les deux règles).

Définition 50. Pour un environnement de typage initial $\Gamma_0(m) = (0, 0, 0)$ pour tout m, une exécution bien typée du programme `p` sur une entrée x est la suite :

$$[\![\mathtt{p}]\!](x) \equiv (\mathrm{ip}_i, \mu_i, \Gamma_i)_{i\in\mathbb{N}} \text{ si } \mathtt{C}_i \vdash (\mathrm{ip}_i, \mu_i, \Gamma_i) \to (\mathrm{ip}_{i+1}, \mu_{i+1}, \Gamma_{i+1})$$

On note k_i le niveau d'exécution maximal atteint à l'étape de calcul i :

$$k_i = \max_{j\leq i}\{k_j \mid \Gamma_j \vdash \mathrm{ip}_j : X(k_j)\}$$

Le niveau k d'une exécution bien typée est défini comme le niveau d'exécution maximal atteint pour l'ensemble du calcul :

$$k = \begin{cases} \max_i\{k_i\} & \text{si défini} \\ \infty & \text{sinon} \end{cases}$$

Remarque 51. Le fait qu'un programme ne termine pas est une condition nécessaire mais pas suffisante pour que $\max_i\{k_i\}$ ne soit pas défini. Par exemple, le programme constitué de l'unique ligne `while true do nil` ne s'arrête pour aucune entrée, mais a un niveau d'exécution $k = 1$.

En revanche, si l'on transforme ce programme en programme fuyant (cf. propriété 18 du chapitre 3), c'est-à-dire en :

```
q := Z(if true do nil); eval(q)
```

alors on obtient un programme équivalent pour lequel $k = \infty$.

Propriété 52. *Le niveau d'exécution d'un programme* `p` *sur une entrée* x *est strictement supérieur à 1 si et seulement si* `p` *est auto-modifiant sur l'entrée* x*.*

5.2.3 Système de typage monotone

Le système de typage défini ci-dessus peut croître et décroître au cours de l'exécution, c'est par exemple le cas pour le programme ci-dessous :

```
l1: @l2 := enc (goto @l3)
l2: stop
l3: stop
```

Quelle que soit son entrée, le niveau de `l2` sera 2 et le niveau de `l3` sera 1. Il peut être intéressant de considérer que le niveau de `l3` est également 2 même si cette commande n'a jamais été écrite. À cet effet, nous allons maintenant introduire un nouveau système de typage $\overline{\Gamma}$ pour lequel le niveau d'exécution est croissant.

Les règles de $\overline{\Gamma}$ sont identiques à celle de Γ à l'exception de la règle d'exécution. À la place, nous appliquons la règle d'exécution monotone.

Définition 53 (Règle d'exécution monotone)**.** On rappelle que k_w est le niveau de l'instruction courante (définition 43) et k_i est le niveau d'exécution maximal atteint à l'étape i (définition 50).

$$\frac{m \in Code_{\mu}(\text{ip: C})}{\overline{\Gamma}' \vdash m : X(\max(k_i, k_w + 1))}$$

Soit $(\text{ip}_i, \mu_i, \overline{\Gamma}_i)_{i\in\mathbb{N}^*}$ une exécution bien typée et $\leq$ l'ordre naturel sur les triplets défini comme $(a, b, c) \leq (a', b', c')$ si $a \leq a'$, $b \leq b'$ et $c \leq c'$.

Propriété 54. *Pour tout i et pour tout m, on a $\overline{\Gamma}_i(m) \leq \overline{\Gamma}_{i+1}(m)$.*

Propriété 55. *Étant données les exécutions bien typées d'un programme `p` sur une entrée x :*

$$(ip_i, \mu_i, \Gamma_i)_{i\in\mathbb{N}^*}$$
$$(ip_i, \mu_i, \overline{\Gamma}_i)_{i\in\mathbb{N}^*}$$

à tout moment, le niveau monotone d'une adresse est supérieur à son niveau non monotone :

$$\forall m\ \forall i,\ \Gamma_i \vdash m : X(k)\ \textit{et}\ \overline{\Gamma}_i \vdash m : X(\overline{k}) \implies k \leq \overline{k}$$

La propriété suivante explicite le fait que les deux systèmes de typage utilisés caractérisent exactement la même classe de programme auto-modifiants. Pour cela, nous allons montrer que tout programme statique en typage classique est également statique en typage monotone et vice versa.

Propriété 56. *On note k le niveau de `p` sur x et $\overline{k}$ son niveau monotone (définition 50). On a alors la propriété suivante :*

$$k = 1 \iff \overline{k} = 1$$

Démonstration. Par définition, k et $\overline{k}$ sont deux entiers strictement positifs.

- on a $\overline{k} = 1 \implies k = 1$ car $0 < k \leq \overline{k}$ (propriété 55)
- montrons que $k = 1 \implies \overline{k} = 1$.
  - comme précédemment, k_i et $\overline{k}_i$ correspondent au niveau d'exécution maximal atteint en typage classique et monotone à l'étape i (définition 50)
  - supposons $\overline{k} > 1$, cela implique $\exists m, i \mid m \in \mathit{Code}(\mathrm{ip}_i\colon \texttt{C})$ et

$$\begin{cases} \overline{\Gamma}_i \vdash m : W(1) \\ \overline{k}_i = 1 \\ \overline{\Gamma}_{i+1} \vdash m : X(2) \end{cases} \tag{5.1}$$

  - or $\forall i,\ 0 < k_i \leq \overline{k}_i$ donc $k_i = 1$. On a donc

$$\begin{aligned} (5.1) \text{ et } k_i = 1 &\implies \Gamma_i \vdash m : W(1) \text{ et } \Gamma_{i+1} \vdash m : X(2) \\ &\implies k_{i+1} = 2 \end{aligned}$$

  - or $k = \max_i\{k_i\} > 1$
  - donc $\overline{k} \neq 1 \implies k \neq 1$

□

5.3 Vagues de code

5.3.1 Définition

Nous avons vu dans la section précédente qu'au cours d'une même exécution, le niveau d'une adresse pouvait varier. Cela permet de stratifier l'image du programme en vagues de code qui évoluent dans le temps. Nous allons maintenant définir une vague de code comme l'ensemble des adresses ayant eu le même niveau durant l'exécution.

Définition 57. Pour une exécution bien typée d'un programme $\mathtt{p}$ sur une entrée x, la *vague de code* k est un triplet $(\mathbf{R}_k, \mathbf{W}_k, \mathbf{X}_k)$. On définit $\mathbf{R}_k$ (resp. $\mathbf{W}_k$, $\mathbf{X}_k$) comme l'ensemble des adresses ayant eu le niveau de lecture (resp. écriture, exécution) k durant l'exécution :

$$\begin{aligned} \mathbf{R}_k &= \{m \mid \exists i\ \Gamma_i \vdash m : R(k)\} \\ \mathbf{W}_k &= \{m \mid \exists i\ \Gamma_i \vdash m : W(k)\} \\ \mathbf{X}_k &= \{m \mid \exists i\ \Gamma_i \vdash m : X(k)\} \end{aligned}$$

Selon le contexte, on définira les couches de code avec le typage classique Γ ou le typage monotone $\overline{\Gamma}$.

Propriété 58. *Soit $\boldsymbol{p} \in$ STAT, pour tout x on a $\forall k > 1$ $(\boldsymbol{R}_k, \boldsymbol{W}_k, \boldsymbol{X}_k) = (\emptyset, \emptyset, \emptyset)$*

5.3.2 Codes auto-modifiants en profondeur et en largeur

Les typages classiques et monotones attribuent des niveaux d'exécution potentiellement différents aux adresses mémoires, et donc les vagues de code qui en découlent sont potentiellement différentes. Cette différence est particulièrement visible dans le cas de codes auto-modifiants en largeur.

Nous allons illustrer ce phénomène avec les figures 5.1 et 5.2. Dans ces figures, la légende est la suivante :

- les blocs A, B, C et D représentent des portions de code ;
- les flèches pleines représentent la relation « génère et transfère le contrôle à » ;
- les flèches en pointillés représentent la relation « transfère le contrôle à ».

Lorsqu'un bloc n'est pointé par aucune flèche pleine (comme le bloc A), cela signifie qu'il s'agit d'un bloc présent dans l'image originale du programme. Les programmes statiques sont donc les programmes que l'on peut représenter avec un bloc unique.

Dans le cas de la figure 5.1, le niveau des différents blocs en typage classique et monotone sont identiques. En revanche, la situation est différente pour l'exemple de la figure 5.2 :

- les blocs A et B ont un niveau classique et monotone identiques (respectivement 1 et 2) ;
- en typage classique, le bloc C a un niveau de 1 car il fait partie du programme original. En typage monotone, le bloc C a un niveau de 2 car le bloc B avait déjà un niveau de 2 ;
- en typage classique, le bloc D a un niveau de 2 car il a été généré par un bloc de niveau 1 (le bloc C). En typage monotone, il a un niveau de 3 car il est généré par un bloc de niveau 2.

On voit donc que le nombre de vagues de code dépend du typage choisi, et que la différence entre le nombre de vagues classiques et monotones donne une mesure de la profondeur des auto-modifications.

Définition 59. La profondeur et la largeur de `p` sur x sont données respectivement par les fonctions Π et Λ :

$$\Pi(k, \overline{k}) = \frac{k}{\overline{k}}$$
$$\Lambda(k, \overline{k}) = 1 - \Pi(k, \overline{k})$$

Propriété 60 (Auto-modification en profondeur)**.** *Pour tous $k, \overline{k}$ correspondant à une exécution bien typée, la mesure de profondeur vérifie les propriétés suivantes :*

$$0 < \Pi(k, \overline{k}) \leq 1$$
$$k = \overline{k} \implies \Pi(k, \overline{k}) = 1$$

Cette dernière propriété traduit l'intuition selon laquelle un programme pour lequel $k = \overline{k}$ est purement auto-modifiant en profondeur (comme en figure 5.1).

FIGURE 5.1: Code auto-modifiant en profondeur

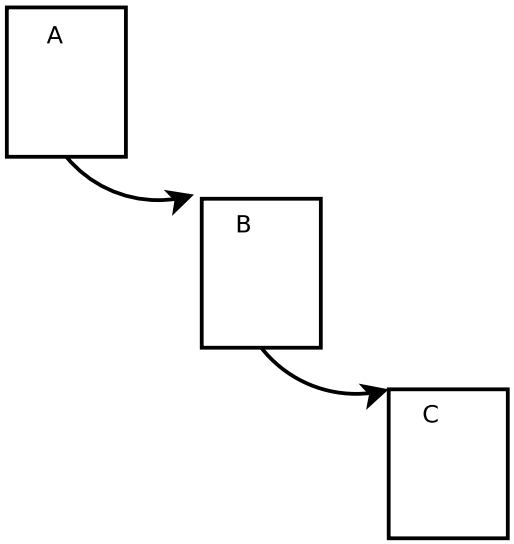
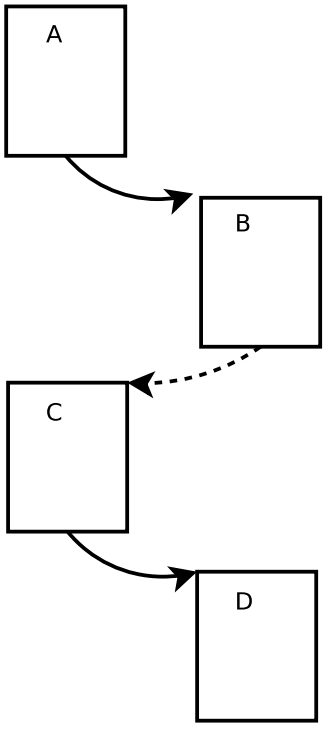


FIGURE 5.2: Code auto-modifiant en largeur



Propriété 61 (Auto-modification en largeur)**.** *Pour tous $k, \overline{k}$ correspondant à une exécution bien typée, la mesure de largeur vérifie les propriétés suivantes :*

$$0 \leq \Lambda(k, \overline{k}) < 1$$
$$\lim_{\overline{k} \to \infty} \Lambda(k, \overline{k}) = 1$$

Remarque 62. On peut voir intuitivement cette dernière propriété de la manière suivante : la largeur d'un programme se rapproche du maximum (100%) pour des grandes valeurs de $\overline{k} - k$. En particulier, il existe des programmes pour lesquels k est fini et $\overline{k}$ est infini, construits sur le modèle suivant :

```
while true do eval(nil)
```

On verra en chapitre 7 des applications de ces deux mesures.

5.4 Comportements auto-modifiants

Il est courant que les auteurs de programmes informatiques souhaitent dissimuler le fonctionnement interne de leurs programmes. C'est par exemple le cas d'entreprises souhaitant protéger leur propriété intellectuelle contre le piratage ou d'auteurs de programmes malveillants souhaitant retarder le travail d'analyse. On parle alors de programmes obfusqués ou même blindés [Fil07] quand ils emploient des techniques de protection avancées.

L'auto-modification peut être vue comme une forme d'obfuscation, mais ce n'est pas la seule possible. Nous allons ici proposer une classification de quelques techniques de protection, ainsi que des définitions formelles de ces techniques permettant de détecter et visualiser leur emploi.

5.4.1 Vérification d'intégrité et écrasement de code en C

Nous allons illustrer deux techniques de protection avec un programme écrit en langage C :

- La vérification d'intégrité consiste pour un programme à vérifier s'il s'exécute sans avoir été modifié. Ce type de protection peut être particulièrement efficace contre les techniques d'analyse fonctionnant par réécriture comme l'instrumentation dynamique de code (voir section 4.3).
- L'écrasement, qui consiste à exécuter du code puis à l'effacer de la mémoire [Fer08]. Il est particulièrement intéressant d'utiliser une protection de ce type en conjonction avec du code auto-modifiant, ce qui rend difficile une vision complète du programme à un instant donné.

Si on se donne une fonction `checksum()` qui calcule une somme de contrôle d'une zone mémoire (par exemple un contrôle de redondance cyclique), alors le programme

en C suivant permet d'illustrer à la fois la vérification d'intégrité et l'écrasement de code :

```
#include <stdio.h>
#include <string.h>

void scramble(unsigned char *s, unsigned int len) {
    memset(s, 0, len);
}

int self_check() {
    unsigned long out = checksum((void*)self_check);
    printf("checksum(self_check) = 0x%08lxL\n", out);
    if(out == 0xe4d6e331L) {
        printf("Vérification d'intégrité ok.\n");
        return 0;
    } else {
        printf("Echec de la vérification d'intégrité.\n");
        return 1;
    }
}

int main() {
    printf("Vérification d'intégrité:\n");
    self_check();

    printf("Ecrasement de la fonction self_check:");
    scramble((unsigned char*)self_check, 16);
    printf("checksum(self_check) = 0x%08lxL\n",
        checksum((void*)self_check));

    printf("self_check n'est plus en mémoire, crash imminent...\n");
    self_check();
}
```

Lorsqu'on l'exécute, ce programme imprime sur la sortie standard le message suivant :

```
Vérification d'intégrité:
checksum(self_check) = 0xe4d6e331L
Vérification d'intégrité ok.

Ecrasement de la fonction self_check:
```

```
checksum(self_check) = 0x00000000L

self_check n'est plus en mémoire, crash imminent...
Segmentation fault
```

La fonction `self_check()` vérifie que la somme de contrôle de la zone mémoire pointée par `self_check` (son propre pointeur de fonction) est égale à une valeur fixe déterminée à l'avance. Si ce n'est pas le cas, cela veut dire que le contenu de la mémoire pointé par `self_check` a changé.

Ensuite, la fonction `scramble()` met les 16 premiers octets de la zone mémoire pointée par `self_check` à zéro. Comme cette valeur ne correspond pas a une instruction valide, le deuxième appel à `self_check()` échoue : la fonction a bien été effacée de la mémoire.

Nous allons maintenant voir comment caractériser formellement ces comportements et comment les détecter.

5.4.2 Signatures logiques

Lorsque du code est exécuté au niveau $k' > 1$, il a été écrit par du code de niveau k avec $0 < k < k'$. On remarquera qu'en typage classique, on a $k' = k + 1$, ce qui n'est pas nécessairement vrai en typage monotone. On définit l'ensemble des adresses mémoires modifiées par le niveau k et exécutées au niveau k' de la manière suivante :

$$\mathit{Self}(k, k') =_{dfn} \mathbf{W}_k \cap \mathbf{X}_{k'} \text{ pour } 0 < k < k'$$

On déduit de la définition des vagues de code qu'un programme est auto-modifiant sur son entrée si et seulement si $\cup_{k<k'} \mathit{Self}(k, k') \neq \emptyset$.

Remarque 63. En typage classique, on pourrait s'attendre à une propriété de la forme $\mathbf{X}_{k+1} = \mathbf{W}_k$ (c'est-à-dire que la vague $k + 1$ est générée par la vague k). En réalité, ce n'est pas le cas car le codage d'une instruction peut s'étendre sur plusieurs adresses et la définition du niveau d'une instruction (définition 43) ne prend en compte que le plus grand des niveaux des adresses qui la composent.

Par exemple, on peut imaginer une commande `C` dont l'encodage s'étend sur deux adresses m_1 et m_2 telles que :

- $\Gamma \vdash m_1 : W(k)$
- $\Gamma \vdash m_2 : W(k + 1)$
- le niveau de `C` est $k_w = k + 2$

Cette instruction est *composite*, car elle a été partiellement écrite par $\mathbf{X}_k$ et par $\mathbf{X}_{k+1}$.

Remarque 64. Le fait que $\mathit{Self}(k, k')$ ne soit pas vide n'implique pas que des valeurs effectivement écrites par la couche k aient été effectivement exécutées par la couche k'. Imaginons par exemple qu'il existe une adresse m telle que $m \in \mathit{Self}(k_1, k')$ et

$m \in \mathit{Self}(k_2, k')$. Cette adresse a été écrite par deux couches distinctes, donc une des deux valeurs a écrasé l'autre. Pour savoir quelle valeur est importante, il faudrait inspecter la trace du programme (dans l'ordre chronologique) au niveau de chaque adresse. Or :

- les instructions ayant un niveau atomique, des informations sur les adresses individuelles sont perdues ;
- avec le système de typage classique, la numérotation des vagues (sur laquelle on se base pour détecter les signatures logiques) ne correspond pas à l'ordre chronologique d'exécution.

Nous allons maintenant définir la vérification d'intégrité et l'écrasement de code pour $0 < k \leq k'$.

Définition 65. La vague k vérifie l'intégrité de la vague k' si

$$\mathit{Check}(k, k') =_{dfn} (\mathbf{R}_k \cap \mathbf{X}_{k'}) \setminus \cup_{min(k,k') \leq k'' \leq max(k,k')} \mathbf{W}_{k''} \neq \emptyset$$

Définition 66. La vague k écrase la vague k' si

$$\mathit{Scrambled}(k, k') =_{dfn} \mathbf{W}_k \cap \mathbf{X}_{k'} \neq \emptyset \text{ pour } k \geq k'$$

Remarque 67. L'inégalité n'est pas stricte dans la définition de *Scrambled*, on peut donc détecter une signature $\mathit{Scrambled}(k, k)$ non vide. Cela signifie que la couche $\mathbf{X}_k$, au cours de son exécution, écrit sur elle-même. Si les adresses écrasées par $\mathbf{X}_k$ sont à nouveau exécutées, elles deviennent la couche $\mathbf{X}_{k+1}$. Une auto-modification sur place est donc détectée comme $\mathit{Scrambled}(k, k)$ puis $\mathit{Self}(k, k+1)$.

Nous pouvons également distinguer deux types d'auto-modifications, selon la présence ou non d'une lecture du code en mémoire.

Définition 68. La vague k modifie la vague k' en aveugle si

$$\mathit{Blind}(k, k') =_{dfn} \mathit{Self}(k, k') \setminus \mathbf{R}_k \neq \emptyset$$

Définition 69. La vague k déchiffre la vague k' si

$$\mathit{Decrypt}(k, k') =_{dfn} \mathit{Self}(k, k') \cap \mathbf{R}_k \neq \emptyset$$

Les quatre définitions que l'on vient de donner correspondent à des schémas de protection réels mais qui n'étaient définis jusqu'à présent que de manière informelle. Ces définitions ne dépendent que d'un typage que l'on peut obtenir de manière efficace avec une trace d'exécution, elles fournissent donc un moyen simple et non-ambigu de détecter si un programme emploie ces comportements particuliers.

Nous verrons au chapitre 6 une visualisation du comportement des programmes basée sur la notion de signature logique et au chapitre 7 des résultats expérimentaux visant à détecter ces signatures dans les traces de programmes malveillants.

5.5 Non-interférence

Nous souhaitons maintenant montrer des propriétés de non-interférence sur les traces grâce au typage dynamique de la mémoire. Ce type de propriétés sur les flux d'informations permet de garantir la confidentialité et l'intégrité des données [VIS96], y compris dans les cas où le code source n'est pas disponible [CNR09].

Dans cette section, nous souhaitons déterminer quelles adresses mémoires sont nécessaires au calcul d'une vague.

5.5.1 Pour une commande

Lemme 70 (Confinement)**.** *Soit une commande* $ip_i :\ \mathtt{C}_i$ *telle que*

$$ip_i :\ \mathtt{C}_i \vdash (ip_i, \mu_i) \to (ip_{i+1}, \mu_{i+1})$$

On a alors $\mu_{i+1}(m) = \mu_i(m)$ *pour tout* $m \notin Out_{\mu_i}(ip_i :\ \mathtt{C}_i)$

Démonstration. Supposons qu'il existe m tel que $\mu_{i+1}(m) \neq \mu_i(m)$. Comme on obtient μ_{i+1} par exécution de $\mathrm{ip}_i :\ \mathtt{C}_i$ sur μ_i, cela signifie que m a été modifiée par l'exécution de la commande. Donc par définition de *Out*, on a $m \in Out_{\mu_i}(\mathrm{ip}_i :\ \mathtt{C}_i)$. D'autre part $Out_{\mu_i}(\mathrm{ip}_i :\ \mathtt{C}_i)$ est un sous-ensemble fini de $dom(\mu_i)$, donc en dehors de ce sous-ensemble on a l'égalité. □

Remarque 71. Une hypothèse importante du lemme de confinement est que la mémoire n'est modifiée que par les commandes du programme. Cette hypothèse est vraie pour les langages en `Machine-GOTO` car on a défini les appels système de manière purement fonctionnelle, sans effet de bord sur la mémoire. Ce modèle est idéal car il suppose l'existence d'un stockage temporaire arbitrairement grand en dehors de la mémoire, et l'hypothèse concernant les appels systèmes n'est pas vraie pour les systèmes réels.

Lemme 72 (Équivalence)**.** *Soit une commande* $ip_i :\ \mathtt{C}_i$ *telle que*

$$ip_i :\ \mathtt{C}_i \vdash (ip_i, \mu_i) \to (ip_{i+1}, \mu_{i+1})$$

Soit un autre environnement μ'_i *tel que*

(a) $\mu'_i(m) = \mu_i(m)$ *pour tout* $m \in In_{\mu_i}(ip_i :\ \mathtt{C}_i) \cup Code_{\mu_i}(ip_i :\ \mathtt{C}_i)$

(b) $ip_i :\ \mathtt{C}_i \vdash (ip_i, \mu'_i) \to (ip'_{i+1}, \mu'_{i+1})$

On a alors

(1) $ip'_{i+1} = ip_{i+1}$

(2) $\mu'_{i+1}(m) = \mu_{i+1}(m)$ *pour tout* $m \in Out_{\mu_i}(ip_i :\ \mathtt{C}_i)$

Démonstration. D'après l'hypothèse (a), la commande exécutée est la même dans les deux environnements, ce qui signifie qu'elle lit les mêmes variables. Or le nombre de cases mémoires lues par une commande en `Machine-GOTO` ne dépend que de la valeur stockée en mémoire, et d'après la première hypothèse les valeurs stockées sont les mêmes, donc dans les deux environnements la commande lit les mêmes valeurs aux mêmes adresses. Comme la sortie d'une commande ne dépend que des valeurs lues, les adresses écrites et leur contenu sont identiques donc la conclusion (2) est vraie.

(1) est vraie car

$$\begin{aligned} \mathrm{ip}'_{i+1} &= next_{\mu'_i}(\mathrm{ip}_i) && \text{d'après (b)} \\ &= next_{\mu_i}(\mathrm{ip}_i) && \text{d'après (a)} \\ &= \mathrm{ip}_{i+1} && \text{par définition de } next \end{aligned}$$

□

Théorème 73 (Robustesse pour une commande)**.** *Soit une commande* $ip_i :\ \mathtt{C}_i$ *telle que*

$$ip_i :\ \mathtt{C}_i \vdash (ip_i, \mu_i, \overline{\Gamma}_i) \to (ip_{i+1}, \mu_{i+1}, \overline{\Gamma}_{i+1})$$

Soit un autre environnement μ'_i *tel que*

(a) $\mu'_i(m) = \mu_i(m)$ *pour tout* $m \in In_{\mu_i}(ip_i :\ \mathtt{C}_i) \cup Code_{\mu_i}(ip_i :\ \mathtt{C}_i)$

(b) $ip_i :\ \mathtt{C}_i \vdash (ip_i, \mu'_i, \overline{\Gamma}_i) \to (ip'_{i+1}, \mu'_{i+1}, \overline{\Gamma}'_{i+1})$

On a alors $\overline{\Gamma}'_{i+1} = \overline{\Gamma}_{i+1}$.

Démonstration. L'égalité $\overline{\Gamma}'_{i+1}(m) = \overline{\Gamma}_{i+1}(m)$ pour m appartenant à $In_{\mu_i}(\mathrm{ip}_i :\ \mathtt{C}_i)$, $Code_{\mu_i}(\mathrm{ip}_i :\ \mathtt{C}_i)$ ou $Out_{\mu_i}(\mathrm{ip}_i :\ \mathtt{C}_i)$ est donnée par le lemme d'équivalence. En effet, les règles de jugement du système de typage $\overline{\Gamma}$ ne dépendent que de ces trois ensembles pour une instruction donnée.

Pour m en dehors de ces ensembles, on a l'égalité grâce au lemme de confinement.

□

5.5.2 Pour une vague

Nous définissons l'ensemble $\mathbf{SX}(k)$ des indices pour lesquels l'instruction courante appartient à la vague k.

Définition 74. Pour une exécution bien typée $(\mathrm{ip}_i, \mu_i, \overline{\Gamma}_i)_{i \in \mathbb{N}^*}$,

$$\mathbf{SX}(k) =_{dfn} \{i \in \mathbb{N}^* \mid \overline{\Gamma}_i \vdash \mathrm{ip}_i : X(k)\}$$

Propriété 75. *Les éléments de* $\boldsymbol{SX}(k)$ *pour* k *donné sont contigus.*

$$\forall k, i_1, i_2 \mid i_1 \leq i_2 \text{ et } i_1, i_2 \in \boldsymbol{SX}(k),\ i_1 \leq i \leq i_2 \implies i \in \boldsymbol{SX}(k)$$

Démonstration. Cette propriété découle directement de la monotonicité du typage. Elle n'est pas vraie en typage classique. □

Dans la section précédente, nous avons employé la notion d'entrée, de sortie et d'intervalle de code pour une instruction donnée. Nous généralisons maintenant cette notion à l'ensemble d'une vague de code.

Lemme 76 (Convergence)**.** *Soit* $(ip_l, \mu_l, \overline{\Gamma}_l) \to \ldots \to (ip_{l+t}, \mu_{l+t}, \overline{\Gamma}_{l+t})$ *un environnement bien typé tel que* $l = min(\boldsymbol{SX}(k))$ *et* $l + t = max(\boldsymbol{SX}(k)) + 1$*. Cela signifie que l'environnement de départ correspond à l'état avant l'exécution de la première instruction de* $\boldsymbol{X}_k$ *et que l'environnement d'arrivée est l'état après l'exécution de la dernière instruction de* $\boldsymbol{X}_k$*.*

Si on note

$$Code_n = \bigcup_{l \leq i < l+n} In_{\mu_i}(ip_i : \mathcal{C}_i)$$

et que l'on définit de la même manière In_n *et* Out_n*, alors on a*

(1) $(Code_n)_{n \leq t}$ *est une suite croissante qui converge vers* $\boldsymbol{X}_k$

(2) $(In_n)_{n \leq t}$ *est une suite croissante qui converge vers* $\boldsymbol{R}_k$

(3) $(Out_n)_{n \leq t}$ *est une suite croissante qui converge vers* $\boldsymbol{W}_k$

Tout est maintenant en place pour établir la robustesse du système de typage $\overline{\Gamma}$. Le théorème de robustesse explicite que pour tout $m \in \mathbf{W}_k$, nous pouvons changer arbitrairement le contenu de l'adresse $m' \notin \mathbf{R}_k \cup \mathbf{X}_k$ sans que cela ait d'impact sur la valeur de m après avoir exécuté les instructions de la vague k.

Théorème 77 (Robustesse du typage)**.** *Soit* $(ip_l, \mu_l, \overline{\Gamma}_l) \to \ldots \to (ip_{l+t}, \mu_{l+t}, \overline{\Gamma}_{l+t})$ *un environnement bien typé tel que* $l = min(\boldsymbol{SX}(k))$ *et* $l + t = max(\boldsymbol{SX}(k)) + 1$*. Supposons que*

(a) $(ip_l, \mu'_l, \overline{\Gamma}_l) \to \ldots \to (ip'_{l+t}, \mu'_{l+t}, \overline{\Gamma}'_{l+t})$ *soit un autre environnement d'exécution*

(b) $\forall m \in \boldsymbol{R}_k \cup \boldsymbol{X}_k$ *on a* $\mu_l(m) = \mu'_l(m)$

Alors

(1) $\overline{\Gamma}_{l+t}(m) = \overline{\Gamma}'_{l+t}(m)$ *pour tout* m

(2) $ip_{l+t} = ip'_{l+t}$

(3) $\mu_{l+t}(m) = \mu'_{l+t}(m)$ *pour tout* $m \in \boldsymbol{W}_k$

Démonstration. On montre aisément :
- (1) par induction sur le théorème 73 ;
- (2) et (3) par induction sur le lemme d'équivalence et le lemme de convergence.

□

Propriété 78. *Avec les mêmes hypothèses que pour le théorème 77, on a la propriété suivante :*

(3') $\mu_{l+t}(m) = \mu'_{l+t}(m)$ *pour tout* $m \in \boldsymbol{R}_k \cup \boldsymbol{W}_k \cup \boldsymbol{X}_k$

Démonstration. Soit $m \in \mathbf{R}_k \cup \mathbf{X}_k$.

- si $m \in \mathbf{W}_k$, alors on a l'égalité d'après la conclusion (3) du théorème 77
- si $m \notin \mathbf{W}_k$, alors

$$\begin{aligned} \mu'_{l+t}(m) &= \mu'_l(m) && \text{par généralisation du lemme de confinement} \\ &= \mu_l(m) && \text{hypothèse (b) du théorème 77} \\ &= \mu_{l+t}(m) && \text{par généralisation du lemme de confinement} \end{aligned}$$

□

5.6 Typage statique

Dans les sections précédentes, nous avons proposé un système d'inférence dynamique de type et une définition des vagues de code comme l'ensemble des éléments d'un même type pour une entrée donnée. Nous allons maintenant nous intéresser au problème de l'inférence statique de type (c'est-à-dire pour un programme donné, indépendamment de son entrée).

5.6.1 Modèle avec variables

Nous nous plaçons dans un premier temps dans le cadre du langage `Machine-GOTO` où les cases mémoires sont des variables semblables à celles du langage `WHILE`. Cela a pour conséquence que :

1. une case mémoire m peut contenir n'importe quel $d \in \mathbb{D}$;
2. une affectation $m := d$ ne modifie que la case m (alors que précédemment $|enc(d)|$ cases étaient modifiées).

Le but du typage que nous proposons dans cette section est de bien typer uniquement les programmes statiques. Nous allons donc appliquer des règles de typage semblables à celles vues en section 5.2, en se limitant à un seul niveau et sans tenir compte des lectures. Pour cela, on définit deux types *code* et *data* que l'on infère avec les règles de typage suivantes :

$$\frac{\texttt{l: C}}{\Gamma \vdash \texttt{l} : \mathit{code}} \qquad \frac{\texttt{l: C}}{\Gamma \vdash \texttt{l+1} : \mathit{code}}$$

$$\frac{\texttt{l: if expr goto l'}}{\Gamma \vdash \texttt{l'} : \mathit{code}} \qquad \frac{\texttt{l: m := expr}}{\Gamma \vdash \texttt{m} : \mathit{data}}$$

On peut comprendre ces règles de typage de la manière suivante :
- une instruction est du code ;
- le successeur d'une instruction est du code ;
- une cible de saut est du code ;
- une cible d'affectation est une donnée.

Définition 79. Un programme $\texttt{p} = \texttt{l}_1\texttt{: C}_1\texttt{; } \ldots\texttt{; l}_n\texttt{: C}_n$ est bien typé s'il n'existe pas d'adresse m telle que :

$$\Gamma \vdash m : \mathit{code} \text{ et } \Gamma \vdash m : \mathit{data}$$

Propriété 80. *Les programmes bien typés sont statiques.*

$$\{\texttt{p} \in \textsc{Progs} \mid \Gamma \vdash \texttt{p}\} \subset \textsc{Stat}$$

Démonstration. Supposons qu'il existe un programme `p` bien typé et une entrée x telle que `p` soit auto-modifiant sur x. Par définition de l'auto-modification interne (définition 11) :
- $\exists i\ \sigma_i^p \neq \sigma_0^p$
- donc $\exists \texttt{l}\ \sigma_i^p(\texttt{l}) \neq \sigma_0^p(\texttt{l})$

On en déduit deux propriétés :

1. une commande `l := expr` a été exécutée. On peut supposer sans perte de généralité que cette commande était dans le texte original du programme, donc d'après les règles de typage on a $\sigma \vdash \texttt{l} : \mathit{data}$;
2. la nouvelle commande `l: C` a été exécutée (par définition de $(\sigma_i^p)_{i\in\mathbb{N}}$). Il n'existe que trois façons d'exécuter cette commande :
   - soit `l` est dans le listing de `p`,
   - soit `l` est le successeur d'une commande dans le listing de `p`.
   - soit `l` est une cible de saut.

   Or nous avons vu que dans ces trois cas, on avait $\sigma \vdash \texttt{l} : \mathit{code}$.

On a donc à la fois $\sigma \vdash \texttt{l} : \mathit{code}$ et $\sigma \vdash \texttt{l} : \mathit{data}$, ce qui contredit l'hypothèse `p` bien typé. ; □

5.6.2 Indécidabilité dans le cas général

Grâce à l'hypothèse sur la taille infinie des cases mémoires, le système de typage trivial décrit ci-dessus permet de rendre décidable la détection de programmes potentiellement auto-modifiants. En effet, sans cette hypothèse on peut aisément imaginer des programmes `Machine-GOTO` à la fois valides, bien typés et auto-modifiants.

Par exemple, il suffit de créer un programme `p` qui :

1. écrit $\mathit{enc}(\texttt{cons}(\texttt{nil}, \underline{\texttt{C}}))$ à une adresse m ;
2. saute à l'adresse $m' \neq m$ contenant l'encodage de la commande `C`.

Ce programme serait à la fois auto-modifiant et bien typé, car

$$\Gamma \vdash m : \mathit{data} \text{ et } \Gamma \vdash m' : \mathit{code}$$

En x86, la situation est comparable au cas général `Machine-GOTO` à cause de la présence de pointeurs. En effet, on pourrait créer un programme qui :

1. écrit à l'adresse pointée par le registre `X` (par exemple `mov [X], d`)
2. saute à l'adresse pointée par le registre `Y` $\neq$ `X` (`jmp [Y]`)

Ce programme serait auto-modifiant si `X` et `Y` contiennent la même valeur, ce qui est indécidable.

En `Machine-GOTO` comme en x86, nous sommes donc contraints à recourir à un typage dynamique afin de lever l'indécidabilité liée aux flux de données.

5.7 Conclusion

Nous avons introduit un système de typage dynamique permettant de détecter les auto-modifications dans une trace d'exécution. Les règles de typage, explicitées en `Machine-GOTO`, s'appliquent naturellement à des traces en langage machine. Ainsi, le chapitre suivant introduira un prototype permettant d'analyser dynamiquement les programmes en x86.

Nous avons proposé de classer les instructions exécutées en vagues de code, permettant ainsi de suivre la structure logique et l'origine des auto-modifications. On a ensuite défini différents comportements en mémoire liés à la protection dynamique de code, ainsi que des signatures logiques du premier ordre permettant de les détecter.

Enfin, nous avons montré (théorème 77) un résultat de non-interférence pour une trace donnée, permettant de décider un sur-ensemble des dépendances pour une vague.

6 Applications à la sécurité logicielle

Nous allons ici présenter les aspects pratiques liés aux systèmes de typages définis précédemment. Dans un premier temps, nous allons introduire le prototype utilisé pour réaliser les tests expérimentaux. Nous allons ensuite décrire quelques scénarios d'utilisation pour l'analyse de programmes protégés (notamment les programmes malveillants) ainsi que pour la détection de vulnérabilités.

6.1 TraceSurfer

Afin de produire des résultats expérimentaux, nous avons implémenté les systèmes de typage décrits dans le chapitre 5 dans un prototype, TraceSurfer. Ce programme est composé de deux modules indépendants :

- un module d'extraction de trace, sous forme d'un *plug-in* pour le programme d'instrumentation dynamique Pin [LCM+05]. Ce module permet d'extraire des traces de binaires x86 pour les systèmes d'exploitation Windows et Linux, sans disposer de leur code source ou d'information de débogage. Au total, ce module est composé d'environ 2 400 lignes de C++.
- un module d'analyse de trace, implémentant les systèmes de typage et la détection des signatures logiques vus en chapitre 5. Ce module est composé d'environ 750 lignes de Python.

Ces deux modules sont indépendants, on pourrait en particulier remplacer le module d'extraction de trace par un outil basé sur TEMU [YS10] ou Ether [DRSL08] pour répondre à des contraintes de transparence ou de performance, tant que les traces générées contiennent les informations nécessaires au typage :

- l'adresse et la taille de chaque instruction exécutée ;
- pour chaque instruction, l'adresse effective et la taille de chaque lecture/écriture en mémoire.

Le chapitre 7 regroupe les résultats expérimentaux permettant d'évaluer la performance de TraceSurfer pour différents scénarios d'utilisation. Dans la suite de ce chapitre, les différents graphes et résultats ont été produits automatiquement par TraceSurfer. Nous allons illustrer son utilisation pour :

- la visualisation du comportement de programmes auto-modifiants (section 6.2) ;
- l'analyse de programmes protégés par *packing* (section 6.3) ;
- l'analyse d'attaques par injection de code (section 6.4).

6.2 Visualisation

6.2.1 Définition des graphes d'auto-référence

La définition des vagues de codes (section 5.3) et la caractérisation des comportements en mémoire (section 5.4.2) permettent d'obtenir une visualisation du comportement des programmes comme un graphe.

Définition 81. Un *graphe d'auto-référence* est un multigraphe dirigé construit à partir d'un environnement d'exécution bien typé $(\mathrm{ip}_i, \mu_i, \Gamma_i)_{i\in\mathbb{N}}$ de la manière suivante :

- le noeud i correspond à la vague $\mathbf{X}_i$;
- les arêtes sont les relations entre les vagues, comme définies en section 5.4.2. Ainsi, on aura une arête A entre les noeuds k et k' si :

$$A(k, k') \neq \emptyset \text{ avec } A \in \{\mathit{Check},\ \mathit{Scrambled},\ \mathit{Blind},\ \mathit{Decrypt}\}$$

Dans l'exemple vu en section 5.4.1, le programme comporte une première vague effectuant de la vérification d'intégrité et de l'écrasement de code, ainsi qu'une deuxième vague constituée par le dernier appel à `self_check()` provoquant un crash. On obtient le graphe en figure 6.1.

Cette visualisation est particulièrement utile pour analyser des programmes automodifiants comportant un grand nombre de vagues avec des relations complexes. En figure 6.2, on peut voir le graphe d'auto-référence d'un programme statique de 154 instructions protégé par PELock. La trace du programme ainsi protégé comporte 13,5 millions d'instructions et 9 vagues de code.

FIGURE 6.1: Vérification d'intégrité et écrasement

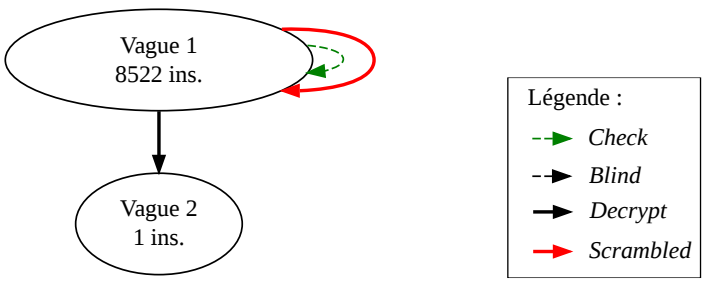
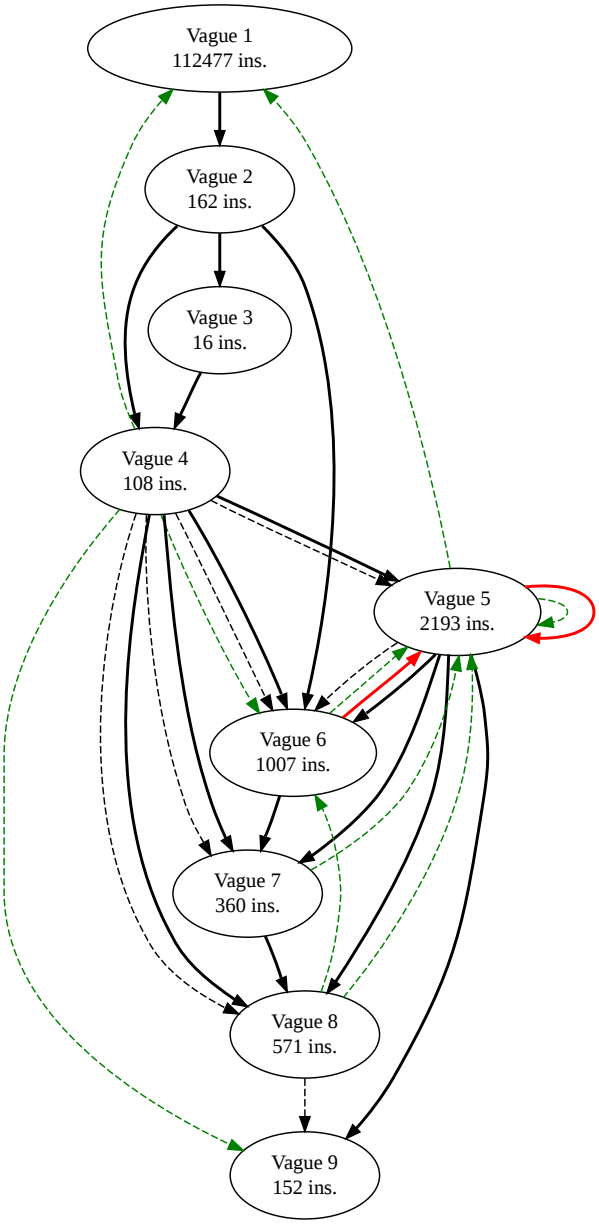


FIGURE 6.2: Analyse de la protection PELock



6.2.2 Relation avec les signatures logiques

Un graphe d'auto-référence donné correspond exactement à une conjonction de formules du premier ordre. Par exemple, le graphe en figure 6.1 correspond à la formule ϕ suivante :

$$\begin{aligned}\phi = & \mathbf{X}_1 \neq \emptyset \\ & \wedge \mathbf{X}_2 \neq \emptyset \\ & \wedge \forall k > 2,\ \mathbf{X}_k = \emptyset \\ & \wedge \mathit{Check}(1,1) \neq \emptyset \\ & \wedge \mathit{Scrambled}(1,1) \neq \emptyset \\ & \wedge \mathit{Decrypt}(1,2) \neq \emptyset\end{aligned}$$

6.2.3 Utilisation comme signature

On peut imaginer utiliser les graphes d'auto-référence comme une forme de signature permettant de caractériser certains comportements. On pourrait alors avoir plusieurs applications potentielles, comme par exemple la détection de comportements malveillants ou encore la vérification de la conformité à une spécification.

Pour la détection de comportement malveillants, il s'agirait de chercher à caractériser certains programmes malveillants par leur graphe d'auto-référence, et de lancer une alerte lorsqu'un sous-graphe d'un comportement malveillant est détecté dans le graphe d'un programme inconnu. L'idée d'utiliser des graphes comme signatures de programmes n'est pas nouvelle, il s'agit du fondement de l'analyse morphologique [BKM09] et d'autres méthodes basées sur les graphes de flot de contrôle [BMM06] ou encore les graphes de flux d'information [YSE+07]. Cette approche serait toutefois à utiliser en conjonction avec d'autres techniques, car le graphe d'auto-référence d'un programme malveillant n'est pas nécessairement suffisant pour les discriminer. En particulier, le graphe de programmes malveillants statiques ne comporte qu'un noeud, ce qui n'est pas suffisamment caractéristique pour faire office de signature. D'autre part, un graphe d'auto-référence correspond à un programme et une entrée donnés, on peut donc imaginer des programmes pour lesquels il n'y a pas de portion invariante du graphe, et donc pas de signature exploitable.

On peut également s'intéresser à la détection de comportements indésirables vus comme la violation d'une spécification (c'est-à-dire une forme de contrôle d'accès). En effet, nous verrons en section 6.4 que l'on peut modéliser les vulnérabilités par injection de code comme du code auto-modifiant non-intentionnel. On pourrait donc imaginer que les programmes soient accompagnés d'un certificat fourni par le développeur qui spécifie si le programme est censé être auto-modifiant. Par exemple, du code auto-modifiant détecté dans un serveur web spécifié comme statique par le développeur constituerait une violation de la politique de sécurité (en plus de

permettre de détecter la présence d'une vulnérabilité exploitée). On retrouve une idée semblable pour la protection contre l'injection de code dans [ABEL05] (en se basant sur les graphes de flot de contrôle) ou encore une approche basée sur l'exécution de données dans le logiciel de capture de programmes malveillants Nepenthes [BKH+06]. On remarquera que la dépendance d'un graphe à une entrée est ici un avantage.

TABLE 6.1: Récapitulatif sur les graphes d'auto-référence

Application	Signature	Programmes autorisés
Détection	Invariant d'un programme malveillant	Programmes qui n'ont pas pour sous-graphe une signature connue
Contrôle d'accès	Fournie par le développeur	Programmes dont le comportement est sous-graphe de la signature fournie

6.3 Analyse de programmes packés

6.3.1 Protection par packing

Nous avons vu dans le chapitre 3 le principe de la détection par signature. Nous avons également introduit le principe du polymorphisme, qui permet de générer des programmes équivalents avec une signature différente. Le *packing* est une méthode de protection extrêmement courante pour la protection des programmes malveillants qui permet d'obtenir des versions polymorphiques auto-modifiantes de n'importe quel programme.

Définition formelle

Pour définir le packing de manière formelle, nous allons d'abord introduire la notion de transformation de programme.

Définition 82. Une *transformation de programme* est une fonction calculable et totale $T : \text{PROGS} \times \mathbb{D} \to \mathbb{D}$ telle que :

1. la restriction à son premier argument est injective, c'est-à-dire pour $x \in \mathbb{D}$ donné on a $\mathtt{p} \neq \mathtt{q} \implies T(\mathtt{p}, x) \neq T(\mathtt{q}, x)$
2. $T(\mathtt{p}, x) \neq \underline{\mathtt{p}}$ pour tout $\mathtt{p}$ et pour tout x.

$T^{-1} : \mathbb{D} \times \mathbb{D} \to \textsc{Progs}$ est la transformation inverse de T. On ne s'intéresse ici aux transformations symétriques, c'est-à-dire telles que :

$$T^{-1}(T(\mathtt{p}, x), x) = \mathtt{p} \text{ pour tout } \mathtt{p}, x \in \textsc{Progs} \times \mathbb{D}$$

Nous établissons maintenant que la restriction de la transformation inverse à l'ensemble $T(\textsc{Progs}, \mathbb{D})$ est calculable, c'est-à-dire pour toute donnée qui correspond effectivement à un programme transformé.

Propriété 83. *Soient T et T^{-1} une transformation et sa transformation inverse. Il existe un programme* `unzip` *qui calcule T^{-1} tel que pour tous $d, x \in \mathbb{D}^2$:*

$$\exists \mathtt{p} \mid d = T(\mathtt{p}, x) \implies [\![\mathtt{unzip}]\!](d, x) = \mathtt{p}$$

Démonstration. Par définition de T, il existe un programme `zip` qui calcule T tel que $\forall \mathtt{p}, x\ [\![\mathtt{zip}]\!](\mathtt{p}, x)\downarrow$. Étant donné que T est injective, il suffit donc d'énumérer les éléments de l'ensemble $\{[\![\mathtt{zip}]\!](\mathtt{p}, x)\}_{\mathtt{p} \in \textsc{Progs}}$ jusqu'à trouver d (pour d et x donnés). □

Définition 84 (Fonction de packing)**.** Soient `zip` et `unzip` deux programmes tels que $[\![\mathtt{zip}]\!]$ est une transformation de programmes et $[\![\mathtt{unzip}]\!] = [\![\mathtt{zip}]\!]^{-1}$. Soient `pre` et `post` deux programmes tels que $[\![\mathtt{pre}]\!](d)\downarrow$ et $[\![\mathtt{post}]\!](d)\downarrow$ pour toute entrée d.

Une *fonction de packing* $f_x : \textsc{Progs} \to \textsc{Smc}$ est une fonction totale calculable qui à tout programme associe un programme auto-modifiant `q` de la forme :

```
q =
    d := read();
    pre(d);
    eval(unzip(data, x));
    post(d)
```

où `data` est la constante $[\![\mathtt{zip}]\!](\mathtt{p}, x)$.

Remarque 85. La définition ci-dessus correspond à une modélisation arbitraire des packers. Cette modélisation est composée de plusieurs éléments :

- l'évaluation d'une constante décodée qui correspond au programme original, il s'agit du coeur du fonctionnement d'un packer ;
- deux programmes `pre` et `post` qui n'ont pas d'effet sur le programme décodé, on autorise ainsi le programme original à être noyé au sein d'un flux d'instructions. Dans certains cas concrets, le programme `pre` exécute des milliards d'instructions afin de retarder intentionnellement l'exécution du programme original (voir par exemple le packer Morphine dans la figure 7.1 ;
- une entrée d qui permet à `pre` et `post` d'implémenter des comportements variables.

Cette modélisation semble englober tous les comportements observés par l'expérience sur une sélection de packers (voir chapitre 7).

Définition 86. Une fonction de packing est dite :
- *simple* si le programme `pre` est statique ;
- *acceptable* si le programme `post` est vide.

On dit qu'un programme est *packé* s'il est l'image d'un autre programme par une fonction de packing.

Propriété 87. *Soient $\mathtt{p}$ un programme statique et f_x une fonction de packing acceptable et simple, le programme $f_x(\mathtt{p})$ a un niveau d'exécution de 2, quelle que soit son entrée.*

Transparence, polymorphisme et terminaison

La fonction de packing ci-dessus ne garantit pas strictement que $[\![\mathtt{p}]\!] = [\![f_x(\mathtt{p})]\!]$. En effet, on peut imaginer des programmes qui dépendent de leur représentation concrète pour calculer leur sortie, et le problème de la transparence du packing se pose de la même manière que pour l'instrumentation (voir section 4.3.2).

En supposant que f_x soit transparente pour `p` donné, on a bien des programmes polymorphiques car :
- $[\![\mathtt{p}]\!] = [\![f_x(\mathtt{p})]\!]$
- $\underline{\mathtt{p}} \neq \underline{f_x(\mathtt{p})}$

Comme on l'a vu au chapitre 3 avec le code de César, il est important de choisir une transformation de programme qui maximise la taille de l'ensemble $T(\mathtt{p}, \mathbb{D})$ afin d'obtenir un nombre maximal de variantes $\{f_x(\mathtt{p})\}_{x\in\mathbb{D}}$.

Par exemple, soit `p` tel que $enc(\underline{\mathtt{p}}) = b_1, b_2, \ldots, b_n$ et σ une permutation de l'ensemble $\{1, 2, \ldots, n\}$. On peut alors s'intéresser à la transformation $T(\mathtt{p}, \sigma) = d$ telle que $enc(d) = b_{\sigma(1)}, b_{\sigma(2)}, \ldots, b_{\sigma(n)}$, car il existe $n!$ permutations possibles.

Enfin, nous établissons que la terminaison d'un programme packé est équivalente à la terminaison du programme original.

Propriété 88. *Soient un programme $\mathtt{p}$ et une fonction de packing f_x. Pour toute entrée d, on a :*

$$[\![f_x(\mathtt{p})]\!](d)\downarrow \iff [\![\mathtt{p}]\!](d)\downarrow$$

Démonstration. Par définition d'une fonction de packing (définition 84), on a :

$$[\![\mathtt{pre}]\!](d)\downarrow \text{ et } [\![\mathtt{post}]\!](d)\downarrow$$

D'autre part, d'après la propriété 83 on a $[\![\mathtt{unzip}]\!](\mathtt{data}, x)\downarrow$ car $\mathtt{data} = [\![\mathtt{zip}]\!](\mathtt{p}, x)$. □

6.3.2 Approche classique pour l'unpacking générique

Le problème de l'unpacking générique peut être posé ainsi : étant donné un programme packé $f_x(\mathtt{p})$, retrouver $\mathtt{p}$. Lorsque la fonction de packing utilisée f_x est connue on peut envisager des techniques d'unpacking statique ou semi-statique, qui ne rentrent pas dans le cadre de l'unpacking générique.

L'approche classique pour l'unpacking générique sur l'architecture x86 comprend les étapes suivantes :

1. lors de l'exécution du programme, on construit un ensemble d'adresses exécutées et un ensemble d'adresses écrites ;
2. à chaque étape de calcul (c'est-à-dire à chaque instruction), on calcule l'intersection entre les adresses à exécuter et les adresses déjà écrites ;
3. on répète les opérations précédentes jusqu'à ce que l'intersection ne soit pas vide, auquel cas on a détecté du code auto-modifiant ;
4. on retranscrit alors l'état actuel de la mémoire dans un fichier (on parle de *dump mémoire*) ;
5. enfin, on essaie de reconstruire un exécutable valide à partir de cette représentation de la mémoire.

De nombreuses implémentations sont basées sur ce modèle, les différences se situant essentiellement dans la manière de suivre l'exécution du programme :

- Renovo [KPY07] et Pandora [Boh08] fonctionnent par émulation ;
- Ether [DRSL08] fonctionne par virtualisation matérielle ;
- Saffron [QA08] et OmniUnpack [MCJ07] utilisent un programme en mode kernel et les permissions sur les pages mémoire pour détecter l'auto-modification ;
- VxStripper [Jos09] et PolyUnpack [RHD+06] utilisent une variante de la première étape. Au lieu de détecter un changement dans l'évolution du programme avec une écriture, ces outils comparent la représentation en mémoire avec la représentation sur le disque. Un changement entre ces deux représentations indique également qu'il y a eu une auto-modification.

Il existe aussi des solutions non génériques, soit car elles nécessitent une connaissance a priori de la fonction de packing f_x [Lop08, GFC08] soit parce qu'elles utilisent des heuristiques spécifiques à un système d'exploitation.

Enfin, on notera qu'il existe un certain nombre de contre-mesures techniques destinées à faire échouer les approches classiques d'unpacking générique, comme par exemple :

- les boucles pour ralentir l'émulation [Fer08] ;
- l'écrasement de code (voir section 5.4.2), afin que le programme retranscrit à partir de l'état de la mémoire soit incomplet ;
- l'utilisation de code interprété [SLGL09] afin d'échapper à la détection de code auto-modifiant natif.

6.3.3 Contribution

Le système de typage défini dans le chapitre 5 permet de détecter les auto-modifications dans la trace d'exécution de n'importe quel programme. C'est en particulier vrai pour les programmes packés. À partir de la visualisation des vagues de code sous forme de graphes, nous pouvons extraire des informations sur le programme original en faisant des hypothèses sur la fonction de packing.

Dans les sections suivantes, nous nous intéressons au problème de l'unpacking d'un programme $\mathtt{q} = f_x(\mathtt{p})$. La procédure d'unpacking consiste à retrouver $\mathtt{p}$ à partir de $\mathtt{q}$, f_x étant inconnue. Nous allons distinguer plusieurs cas, en faisant des hypothèses sur la fonction de packing utilisée et sur le programme original.

Fonction de packing acceptable et simple

Nous considérons le problème de l'unpacking de $\mathtt{q}$ en faisant les hypothèses suivantes :

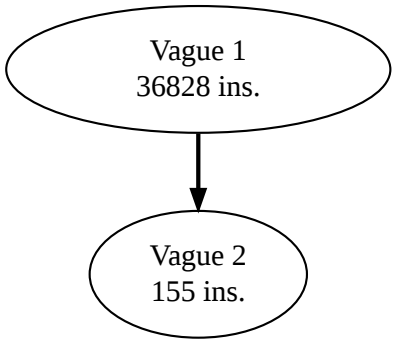
- f_x est une fonction de packing acceptable et simple (définition 86) ;
- $\mathtt{p}$ est statique.

Dans ce cas, la procédure d'unpacking termine sur toute entrée d (il suffit d'arrêter le calcul après avoir trouvé une deuxième vague de code) et :

- le niveau d'exécution est $k = 2$ (propriété 87) ;
- la vague de code $\mathbf{X}_1$ fait partie des programmes `pre` et `unzip` (définition 84) ;
- seule la vague de code $\mathbf{X}_2$ fait partie du programme `p` original.

On peut voir le résultat de l'analyse d'un programme protégé par une fonction de packing simple en figure 6.3. Dans ce cas précis, le packer (vague 1) est ASPack [1] et le programme protégé (vague 2) est `hostname.exe` [2].

FIGURE 6.3: Protection par packing acceptable et simple



1. http://www.aspack.com/
2. Un des programmes par défaut sur une installation du système d'exploitation Windows XP.

Fonction de packing acceptable

Dans le cas général, une fonction de packing peut comporter un nombre quelconque de vagues de code. Arrêter le calcul lorsqu'une deuxième vague est détectée comme précédemment n'est donc pas satisfaisant. Cette fois nous considérons le problème de l'unpacking de `q` avec les hypothèses suivantes :

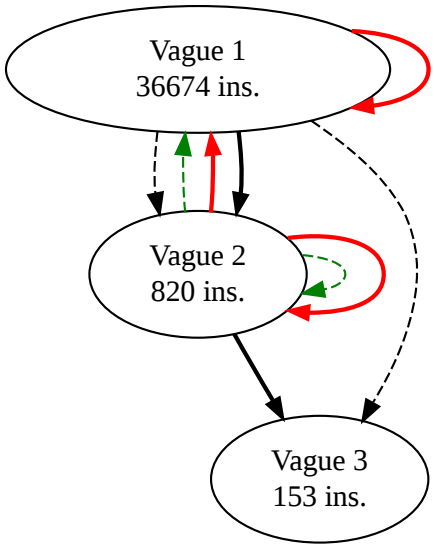
- f_x est une fonction de packing acceptable (définition 86) ;
- `p` est statique et $[\![\mathtt{p}]\!](d)\downarrow$ pour tout d.

D'après la propriété 88, `q` termine toujours donc la procédure d'unpacking termine également toujours. Dans ce cas :

- le niveau d'exécution est $k \geq 2$;
- les vagues de code $\mathbf{X}_1, \ldots, \mathbf{X}_{k-1}$ font partie de `pre` et de la transformation inverse `unzip` (définition 84) ;
- seule la vague de code $\mathbf{X}_k$ fait partie du programme `p` original.

La figure 6.4 illustre la protection de `hostname.exe` (vague 3) par PECompact [3] (vagues 1 et 2).

FIGURE 6.4: Protection par packing acceptable



3. http://www.bitsum.com/pecompact.shtml

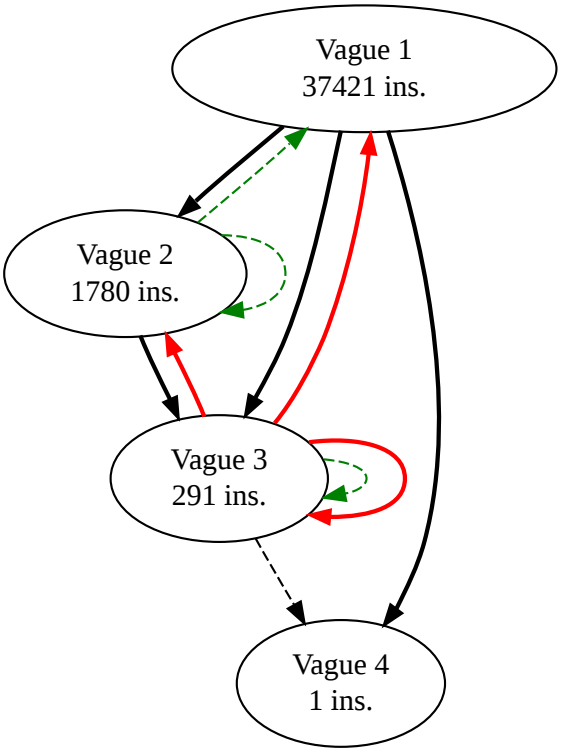
Fonction de packing quelconque

Enfin, si l'on fait les hypothèses suivantes :
– f_x est quelconque ;
– `p` est statique et $[\![\texttt{p}]\!](d)\downarrow$ pour tout d.

Alors la procédure d'unpacking termine toujours mais nous ne sommes plus en mesure de distinguer quelle vague de code correspond au programme original `p`. L'hypothèse f_x acceptable garantissait que le graphe comportait un noeud sans successeur correspondant au programme original. Comme le programme `post` est maintenant quelconque, cela signifie qu'il existe un noeud dans le graphe qui contient à la fois `p` et une partie de `post`, et que ce noeud peut avoir des successeurs si `post` a des relations avec d'autres vagues de code. C'est par exemple le cas si `post` est auto-modifiant.

La figure 6.5 illustre la protection de `hostname.exe` par Yoda Protector [4]. Nous ne sommes pas en mesure de déterminer la vague exacte contenant le code de `hostname.exe`, mais on notera toutefois qu'il s'agit forcément de la vague 2 ou 3, car son niveau doit être strictement supérieur à 1 et la vague 4 est trop petite pour le contenir.

FIGURE 6.5: Protection par packing quelconque



4. http://yodap.sourceforge.net/

6.4 Détection de vulnérabilités

6.4.1 Introduction aux attaques par débordement de tampon

Comme nous l'avons mentionné en section 6.2.3, nous allons ici montrer que l'on peut voir une certaine classe de vulnérabilités comme du code auto-modifiant non-intentionnel. Nous allons nous intéresser aux attaques par injection de code dans les programmes en x86, qui sont typiquement exploitées grâce à des débordements de tampon [Spa89, CPM+98]. Un tampon est une zone mémoire utilisée comme stockage temporaire, et il y a débordement lorsqu'on essaie de copier dans le tampon plus de données qu'il ne peut en recevoir. Des données critiques peuvent alors être écrasées, ce qui conduit à plusieurs cas de figure :

- un crash ou un comportement imprévisible du programme ;
- une redirection du flot d'exécution vers une zone mémoire différente.

Ce dernier cas est particulièrement critique, car un attaquant peut alors rediriger le flot d'exécution vers une zone mémoire qu'il contrôle (on parle d'injection de code) ou vers une librairie (on parle d'attaques par retour dans la *libc*).

Dans tous les cas, l'attaquant a alors pris le contrôle du programme ciblé et il n'y a pas nécessairement eu d'effet visible. Tous les programmes sont potentiellement vulnérables à ce type d'attaques lorsqu'ils lisent des données, ce qui en fait un vecteur d'exploitation et d'infection extrêmement important.

6.4.2 Exemple en langage C

Le programme en langage C suivant est vulnérable à une attaque par débordement de tampon :

```
#include <string.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buffer[4];
    strcpy(buffer, argv[1]);
    return 0;
}
```

Il fonctionne de la manière suivante :

- il déclare un tampon, `buffer`, avec une taille de 4 octets ;
- il copie son premier argument dans le tampon ;
- enfin, il rend la main.

Ce programme ne fait rien de particulier tant que la taille de `argv[1]` est inférieure à 4 octets. En revanche, si `argv[1]` a une taille supérieure, il y aura un débordement de tampon.

Lorsque ce programme est compilé en x86, l'allocation du tampon a lieu sur la pile car `buffer` est une variable locale. Or, la pile contient également l'adresse de retour de la fonction `main`. Ainsi, si `argv[1]` a une taille suffisante, il sera possible d'écraser cette adresse de retour avec une valeur contrôlée par l'attaquant.

Si on suppose que le programme ciblé s'exécute sans mesure de protection particulière (comme un espace d'adressage randomisé ou une pile non-exécutable), on peut construire un argument de la manière suivante :

- 16 octets de bourrage quelconques destinés à écraser les variables locales du programme (`buffer`, `argc`, `argv[0]` et `argv[1]`) ;
- puis la nouvelle adresse de retour x ;
- puis une « zone d'atterrissage » constituée d'un certain nombre d'instructions `nop`[5] ;
- enfin, la charge utile à exécuter pour prendre le contrôle du programme. Typiquement, la charge utile donne accès à une ligne de commande, ou cherche à se connecter à un serveur contrôlé par l'attaquant.

On calcule la nouvelle adresse de retour de la manière suivante :

$$x = base + n_{bourrage} + |x| + \frac{n_{nops}}{2}$$

où :

- *base* est l'adresse estimée de `buffer` ;
- $n_{bourrage}$ est le nombre d'octets de bourrage (ici, 16) ;
- $|x|$ est la taille du pointeur x, par exemple 4 octets sur une architecture 32 bits ;
- n_{nops} est le nombre de `nop` dans la zone d'atterrissage.

Le but est que la chaîne passée en argument contienne une valeur x pointant sur n'importe quel octet de la zone d'atterrissage. Ainsi, si le contrôle est transféré à cette zone, toutes les instructions `nop` seront exécutées, puis la charge utile le sera également.

Par exemple, voici ce qui se passe lorsqu'on exécute le programme vulnérable sur une entrée bien construite, avec une charge utile se contente de lister le contenu du répertoire courant [One96] :

```
$ ./vuln  $'AAAAAAAAAAAAAAAA\x80\xf3\xff\xbf\x90\x90\x90\x90\x90
...
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd
\x80\xe8\xdc\xff\xff\xff/bin/ls'
vuln.c     vuln
```

5. `nop` signifie *no operation*, ou « opération nulle »

6.4.3 Analyse par TraceSurfer

Le graphe d'auto-référence du programme vulnérable vu précédemment varie selon son entrée :

- sur une entrée normale (de taille inférieure ou égale à 4 octets), le programme est statique donc le graphe ne comporte qu'un noeud et pas d'arête (figure 6.6) ;
- sur une entrée conçue spécialement pour exploiter la vulnérabilité, le programme sera auto-modifiant et le graphe comportera deux vagues, ou plus si la charge utile est également auto-modifiante (figure 6.7).

FIGURE 6.6: Programme vulnérable avec entrée normale

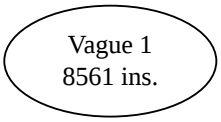
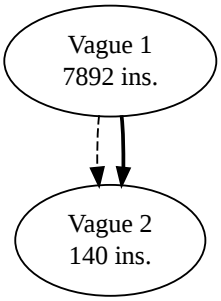


FIGURE 6.7: Programme vulnérable avec entrée exploitant la faille



TraceSurfer permet de discriminer le cas où du code a été injecté, car il apparaît comme du code auto-modifiant. Si l'on sait que le programme est fait pour être statique (par exemple, s'il est accompagné d'un certificat signé par le développeur déclarant le programme comme statique), alors la présence de code auto-modifiant est non-intentionnelle et indique qu'une attaque a potentiellement été menée sur le programme.

On peut remarquer que les techniques de protection comme DEP [6] visent également à empêcher l'exécution involontaire de certaines zones mémoires censées contenir des données, comme la pile ou le tas.

6. *Data Execution Prevention*

6.5 Conclusion

Nous avons vu que TraceSurfer, l'implémentation des systèmes de typage définis dans le chapitre 5, permettait d'analyser les fonctions de packing, utilisées notamment comme une forme d'obfuscation visant à complexifier la compréhension du code.

Un autre scénario d'utilisation est envisageable, basé sur la surveillance dynamique d'un programme. Ainsi, il est possible de lever une alerte lorsque du code auto-modifiant non-intentionnel est détecté, indiquant qu'une faille a été rencontrée et potentiellement exploitée.

7 Résultats expérimentaux

Dans ce chapitre, nous proposons de tester le prototype présenté au chapitre précédent sur divers types de programmes auto-modifiants ou non.

Dans un premier temps, nous testons TraceSurfer sur des programmes packés (section 7.1) et sur des compilateurs à la volée (section 7.2). Ce test permet de tester la performance du prototype et sa validité sur des programmes dont on sait par construction qu'ils sont auto-modifiants. Nous mettons ces deux tests en perspective, afin mettre en évidence la différence de comportement entre un packer et un compilateur à la volée.

Dans un deuxième temps, nous effectuons un test à grande échelle sur des programmes malveillants (section 7.3), et comparons les résultats avec ceux d'un test similaire réalisé sur des programmes sains (section 7.4).

7.1 Analyse de programmes packés

7.1.1 Jeu de test

Nous avons réalisé une expérience sur un jeu d'échantillons synthétiques :
- en sélectionnant un témoin, le programme `hostname.exe` ;
- en collectant le plus de packers possibles (28 différents en tout) et en les appliquant au témoin.

Nous avons donc obtenu 28 programmes packés différemment dont la terminaison et la sortie sont connues, et dont on suppose qu'ils doivent comporter au moins deux vagues de code. Cette expérience avait plusieurs objectifs :
- savoir si l'analyse par instrumentation fonctionnait sur des programmes packés ;
- mesurer la performance de cette analyse sur des échantillons semi-contrôlés ;
- valider le typage dynamique sur des programmes auto-modifiants « réels » ;
- mesurer le niveau de protection employé dans l'ensemble de packers retenus.

7.1.2 Évaluation des performances

La figure 7.1 rassemble les résultats concernant la réussite de l'analyse et ses performances. En particulier, on retiendra que :
- l'instrumentation échoue (elle n'est pas transparente) dans 4 cas, ce qui donne un taux de réussite de 85,71% ;
- en moyenne, on analyse un programme en 45,73s avec un écart-type de 127,05s ;

– toutefois, si l'on exclut le packer Morphine pour lequel on a des valeurs extrêmes, on a une moyenne de 22,33s et un écart-type de 19,30s.

7.1.3 Résultats du typage

La figure 7.3 synthétise le résultat du typage des traces de programmes packés.

– 13 des 28 packers testés (46,4%) sont simples, car ils ne comportent que 2 vagues ;
– 10 des programmes packés sont auto-modifiants en profondeur (voir section 5.3.2) ;
  – aucun n'est complètement auto-modifiant en largeur, même si certains comme ACProtect ont un grand facteur de largeur ;
– on détecte les comportements sophistiqués *Check* et *Scrambled* dans la majorité des programmes packés (respectivement dans 75% et 50% des cas).

D'autre part, pour 2 des packers (PEPack et VMProtect), on a un niveau d'exécution de 1, alors que le programme se termine normalement avec la même sortie que le programme original. Cela signifie que la trace est complète (il n'y a pas eu de crash) et que le programme original s'est exécuté sans toutefois être auto-modifiant. On remarque également que pour ces deux programmes, l'augmentation du nombre d'instructions exécutées n'est que de 0,3% et 0,8% alors qu'en moyenne, l'augmentation est de 475% pour les autres packers. Il semble donc que ces deux programmes ne soient pas packés au sens de la définition 84.

7.2 Analyse de compilateurs à la volée

Nous avons ensuite réalisé une expérience sur des interpréteurs dotés de compilateurs JIT [1], ou compilateurs à la volée.

7.2.1 Introduction aux compilateurs à la volée

Un interpréteur classique est basé sur une boucle d'exécution, ou *fetch-decode-execute loop*. Le corps de la boucle correspond aux opérations suivantes :

1. lire les données correspondant à l'instruction suivante ;
2. décoder ces données pour remonter à la syntaxe abstraite de l'instruction ;
3. exécuter l'instruction et mettre à jour l'environnement.

1. *Just-In-Time*

FIGURE 7.1: Résultats de l'instrumentation

Nom du binaire	Succès	Instructions	Trace (Mo)	t_I (s)	t_A (s)
hostname.exe (original)	✓	$2,74.10^6$	23,50	12,47	37,23
!EPack_1.0.exe	✓	$8,72.10^7$	533	25,50	691,16
acprotect-hostname.exe	✓	$1,20.10^7$	107	88,11	174,95
aspack-hostname.exe	✓	$3,16.10^6$	28	12,52	42,00
enigma_protector_1.16.exe	✓	$2,19.10^7$	180	26,41	266,05
exefog_1.1.exe	✓	$3,32.10^6$	30	12,44	44,64
expressor-hostname.exe	✓	$8,85.10^6$	82	37,11	126,31
fsg.exe	✓	$3,33.10^6$	30	12,30	45,22
mew11.exe	✓	$3,32.10^6$	30	11,92	45,00
molebox-hostname.exe	✓	$3,43.10^7$	313	54,81	479,70
morphine_1.9.exe	✗	$1,64.10^8$	1 558	710,67	2 461,06
nakedpack.exe	✓	$5,53.10^6$	48	12,80	68,22
npack-hostname.exe	✓	$3,38.10^6$	30,67	12,59	47,59
nspack.exe	✓	$3,82.10^6$	34,00	13,27	50,38
packman_1.0.exe	✓	$2,91.10^6$	26,11	12,17	38,59
pec2-hostname.exe	✓	$3,52.10^6$	31,76	12,88	47,44
pelock-hostname.exe	✓	$1,32.10^7$	110,53	39,52	165,48
pepack.exe	✓	$2,75.10^6$	24,78	12,11	37,13
pespin-hostname.exe	✗	$5,84.10^6$	42,01	24,88	59,16
petite.exe	✗	$2,17.10^6$	19,54	17,28	28,77
rlpack_1.17_full_version.exe	✓	$3,42.10^6$	30,95	12,08	46,09
rlpack-hostname.exe	✓	$3,91.10^6$	35,09	12,39	56,77
telock_0.98.exe	✗	$9,87.10^5$	8,84	7,84	13,75
themida_1.8.5.2.exe	✓	$8,22.10^7$	665,96	69,61	1 060,23
upx-hostname.exe	✓	$2,90.10^6$	25,96	12,22	38,41
vmprotect-hostname.exe	✓	$2,76.10^6$	24,89	11,70	37,13
winupack-hostname.exe	✓	$3,92.10^6$	36,25	12,58	53,78
Yodas_Crypter_v1.3.exe	✓	$3,53.10^6$	31,52	12,44	46,69
yp-1.02-hostname.exe	✓	$4,23.10^6$	36,64	13,53	53,69

Légende :
- la colonne « Succès » indique si le programme instrumenté a la même sortie que le programme original ;
- t_I est le temps d'exécution du programme instrumenté, en secondes ;
- t_A est le temps d'analyse de la trace par notre outil, en secondes.

FIGURE 7.2: Résultats de l'analyse

Nom du binaire	k	$\overline{k}$	*Blind*	*Decrypt*	*Check*	*Scrambled*
hostname.exe (original)	1	1				
!EPack_1..exe	2	2		✓	✓	
acprotect-hostname.exe	18	882	✓	✓	✓	✓
aspack-hostname.exe	2	3	✓	✓	✓	✓
enigma_protector_1.16.exe	5	24	✓	✓	✓	✓
exefog_1.1.exe	3	5		✓	✓	✓
expressor-hostname.exe	2	3		✓	✓	
fsg.exe	2	2	✓	✓	✓	
mew11.exe	2	2	✓	✓	✓	
molebox-hostname.exe	3	5	✓	✓	✓	✓
morphine_1.9.exe	3	3	✓	✓	✓	
nakedpack.exe	2	2	✓			
npack-hostname.exe	2	2		✓		
nspack.exe	3	4	✓	✓	✓	
packman_1..exe	2	2	✓	✓		✓
pec2-hostname.exe	3	4	✓	✓	✓	✓
pelock-hostname.exe	9	16	✓	✓	✓	✓
pepack.exe	1	1			✓	
pespin-hostname.exe	4	38	✓	✓	✓	✓
petite.exe	2	2	✓			✓
rlpack_1.17_full_version.exe	2	2	✓	✓	✓	
rlpack-hostname.exe	2	2		✓		
telock_.98.exe	2	2		✓		
themida_1.8.5.2.exe	11	164	✓	✓	✓	✓
upx-hostname.exe	2	2	✓	✓		
vmprotect-hostname.exe	1	1			✓	
winupack-hostname.exe	3	4	✓	✓	✓	✓
Yodas_Crypter_v1.3.exe	4	4	✓	✓	✓	✓
yp-1.02-hostname.exe	4	6	✓	✓	✓	✓

Légende :
– k est le niveau d'exécution maximal en typage classique (chapitre 5, définition 50) ;
– $\overline{k}$ est le niveau d'exécution maximal en typage monotone.

Malheureusement, ce type de fonctionnement est assez lent et peut amener à effectuer souvent les mêmes opérations. Le principe d'un compilateur JIT est d'accélérer l'exécution du programme interprété en compilant des fragments de ce programme en langage natif et en exécutant directement les fragments de code compilés. L'opération de compilation peut être coûteuse, donc l'interpréteur peut chercher à maximiser son efficacité en ne compilant que les instructions exécutées le plus fréquemment comme les boucles.

L'opération qui consiste à compiler un fragment de code et à l'exécuter correspond à notre définition du code auto-modifiant, puisque l'image du programme original (c'est-à-dire l'interpréteur) est étendue avec de nouvelles instructions (correspondant aux fragments compilés du programme interprété). Nous allons confirmer cette intuition en analysant des compilateurs JIT avec TraceSurfer.

7.2.2 Jeu de test

Nous avons sélectionné le plus grand nombre possible d'interpréteurs dotés de compilateurs à la volée, pour des langages variés. Nous les avons ensuite exécutés avec pour argument un programme de test `p` et une entrée x particulière. Le choix de $(\mathtt{p}, x)$ n'est pas anodin, car si l'interpréteur ne détecte pas de point chaud il risque de décider de ne pas compiler de fragments du programme. Autant que possible, nous avons sélectionné des tests standard [2] comme spectral-norm et binary-trees.

On remarquera la présence de Pin dans les interpréteurs testés, car son fonctionnement par instrumentation dynamique correspond effectivement au comportement d'un compilateur JIT. À la différence d'un interpréteur classique, implémenté dans un langage machine pour interpréter un langage source de haut niveau, les langages source et destination de Pin sont identiques (le x86). Pour le test, nous avons donc utilisé Pin (muni de TraceSurfer) pour analyser un programme lui-même instrumenté avec Pin.

Le but de cette expérience est de détecter une différence de comportement entre un programme packé et un programme interprété avec une machine virtuelle dotée d'un compilateur JIT. Dans les deux cas on détecte du code auto-modifiant et dans le cadre de la classification automatique de programmes inconnus, il peut être intéressant d'avoir une indication sur la nature de l'auto-modification : est-elle implémentée à des fins d'optimisation ou à des fins de protection du code ?

7.2.3 Résultats

Les résultats en figure 7.3 confirment une différence importante entre les compilateurs JIT et les programmes packés. En effet, à l'exception de la machine virtuelle

2. Issus du « Computer Language Benchmarks Game », http://shootout.alioth.debian.org/

HotSpot de Sun et de l'interpréteur LLVM, tous les interpréteurs testés ont un niveau d'exécution de 2 et une largeur importante (93,19% en moyenne), alors qu'en moyenne les programmes packés ont un niveau d'exécution de 3,61 pour une largeur de 23,52%. Cela confirme l'intuition qu'un compilateur JIT typique fonctionne par allers et retours entre les portions compilées du programme interprété et la machine virtuelle, alors qu'un packer fonctionne plutôt en couches récursives pour complexifier l'analyse.

Pour les cas avec des valeurs extrêmes :

- pour lli, l'interpréteur de LLVM, on a $k = \overline{k} = 2$. Il semble donc qu'il compile intégralement le programme interprété, puis exécute ce programme compilé sans faire appel à une machine virtuelle ;
- la machine virtuelle HotSpot de Sun a un niveau d'exécution de 3. Si l'interpréteur était statique, cela signifierait que le code compilé dynamiquement aurait à son tour généré du code natif et l'aurait exécuté. Cela serait extrêmement surprenant, car cela irait à l'encontre des contrôles effectués sur les programmes Java avant leur exécution [LY99]. En réalité, une partie de l'interpréteur est générée dynamiquement pour des raisons d'optimisation, ce dont on trouve une confirmation dans [Dmi01, chap. 7].

FIGURE 7.3: Résultats de l'analyse

Interpréteur testé	Langage	Test	k	$\overline{k}$	$\Pi(k, \overline{k})$	$\Lambda(k, \overline{k})$
lli	LLVM	spectral-norm	2	2	100,00%	0,00%
Google v8	JavaScript	spectral-norm	2	92	2,17%	97,83%
Sun HotSpot	Java	fibonacci	3	44	6,82%	93,18%
PyPy	Python	spectral-norm	2	31	6,45%	93,55%
Mono	.NET	binary-trees	2	562	0,36%	99,64%
LuaJIT	Lua	spectral-norm	2	8	25,00%	75,00%
Pin	x86	empty-tool / ls	2	2979	0,07%	99,93%

7.3 Analyse de programmes malveillants

7.3.1 Introduction aux techniques anti-virtualisation

En plus de l'analyse de code auto-modifiant décrite dans les chapitres précédents, nous profitons également de l'instrumentation dynamique pour détecter l'utilisation de techniques anti-virtualisation basées sur le processeur [Mus09]. Ces techniques, baptisées des *redpills* [Rut04, PMR+09], permettent de détecter la présence d'un

hyperviseur matériel de manière invisible pour celui-ci à cause de la présence d'instructions sensibles non-privilégiées [RI00].

Les malwares ont la réputation d'employer ce type de techniques pour détecter s'ils s'exécutent sur une machine d'analyse ou sur une machine réelle [BCK+10, KYH+09]. Nous proposons ici de mesurer par l'expérience la présence effective de ce type de techniques dans des programmes malveillants. [3]

Il existe de nombreuses autres techniques permettant de détecter la présence d'un environnement d'analyse [RKK07], nous nous limitons ici aux techniques :

- basées sur le processeur ;
- qui cherchent à détecter la présence d'un hyperviseur matériel. [4]

7.3.2 Jeu de test et protocole expérimental

Pour cette expérience, nous avons sélectionné 95 613 binaires [5] uniques [6] provenant de captures par un pot de miel et d'échanges avec d'autres universités.

Ces binaires ont ensuite été analysés un à un par instrumentation avec TraceSurfer. Chaque analyse était lancée dans une machine virtuelle Windows XP propre afin d'éviter les interactions entre codes malveillants. L'exécution était interrompue au bout de 2 minutes si le binaire ne s'était pas arrêté.

L'expérience dans son ensemble a été menée sur un cluster de 12 machines, plus une machine de supervision. Chaque machine de calcul comporte deux processeurs quad-core Xeon L5420 et 16 Go de mémoire vive et permet d'exécuter simultanément 2 machines virtuelles.

7.3.3 Résultats

L'analyse a duré environ 67h30 (soit 1 400 binaires de l'heure) et a permis d'extraire 76 993 traces non vides. Les raisons potentielles qui expliquent la présence d'une trace vide sont :

- des binaires syntaxiquement invalides, qui ne s'exécutent donc pas du tout ;
- des binaires qui plantent au lancement ou incompatibles avec le système d'exploitation utilisé pour les tests ;
- des binaires pour lesquels TraceSurfer n'est pas transparent et ne parvient pas à générer une trace.

Dans la suite de cette section, les proportions sont exprimées par rapport au nombre de traces non vides.

3. Les techniques détectées sont détaillées dans [GMR09], nous ne donnerons ici que les résultats de l'expérience.
4. C'est-à-dire qui utilise les technoloies Intel VT-x [NSL+06] ou AMD-V [Adv05].
5. Identifiés comme des fichiers Windows PE par la commande Unix `file`.
6. Relativement à leur hachage `md5`.

Les résultats de l'analyse (figures 7.4, 7.5 et 7.6) permettent de constater :

- l'omniprésence de code auto-modifiant dans le jeu de test, avec seulement 5,29% de programmes à 1 vague. Ce nombre est une borne supérieure, car un programme qui semble être statique à un instant t peut se réveler auto-modifiant à un instant $t + n$;
- le nombre important de programmes à 5 vagues (56,81%) et de programmes exhibant la signature *Check*. Cette observation contredit l'intuition et l'observation faite en section 7.1 (la plupart des packers sont simples). Elle indique un biais dans les binaires récoltés par le pot de miel, constitués en grande partie d'échantillons du ver polymorphique Allaple [7] ;
- la technique anti-virtualisation prévalente est la technique redpill originale (la plus connue et sans doute la mieux documentée) [Rut04], basée sur l'instruction SIDT. Son utilisation est toutefois très minoritaire, on ne détecte l'usage d'une technique anti-virtualisation que dans 234 binaires (soit 0,30% des binaires analysés).

FIGURE 7.4: Comportements détectés dans les programmes malveillants

Comportement	Nombre de binaires	Proportion
Decrypt	63 186	82,07%
Blind	56 520	73,41%
Check	45 569	59,19%
Scrambled	3 332	4,33%

7.4 Analyse de programmes sains

7.4.1 Jeu de test et protocole expérimental

Pour cette expérience, nous avons sélectionné les 467 exécutables [8] uniques présents sur une installation par défaut de Windows XP.

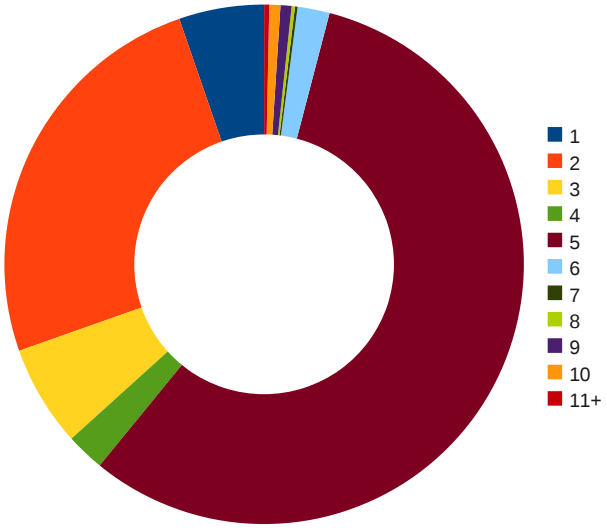
Le protocole expérimental et l'environnement de test était exactement le même qu'en section 7.3.

7.4.2 Résultats

388 traces non vides (soit 83,08% des exécutables) ont pu être extraites en environ 30 minutes. Les résultats de l'analyse (figures 7.7 et 7.8) mettent en évidence les éléments suivants :

7. http://www.f-secure.com/v-descs/allaple_a.shtml
8. Identifiés par leur extension `.exe`.

FIGURE 7.5: Répartition par nombre de vagues



- une grande majorité (84,02%) des binaires testés sont statiques et ceux qui sont auto-modifiants ne comportent que 2 vagues. Une analyse manuelle des programmes dans lesquels 2 vagues ont été détectés a montré qu'il s'agissait de l'effet du compilateur JIT de l'environnement .NET (voir section 7.2) ;
- aucun des binaires testés n'emploie de technique de protection avancée comme la vérification d'intégrité ou l'écrasement de code ;
- aucun des binaires testés n'emploie de technique anti-virtualisation (voir section 7.3.1), ce qui confirme l'intuition que ces techniques ne sont employées que comme une mesure de protection contre l'analyse.

Une comparaison de ces résultats avec ceux de la section 7.3 met en évidence des différences de comportement très importantes entre des programmes potentiellement malveillants et des programmes sains.

FIGURE 7.6: Répartition par technique anti-virtualisation

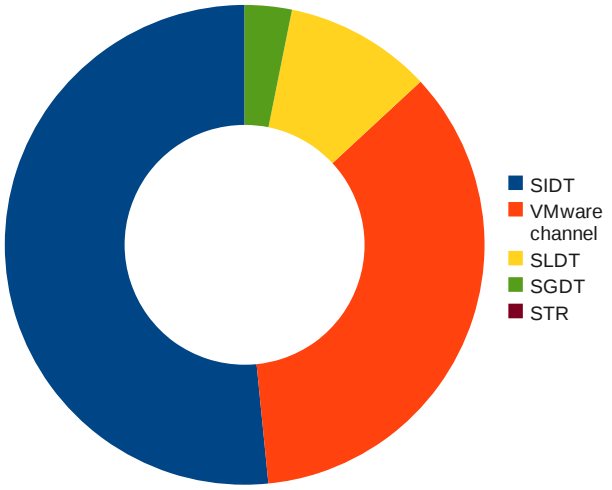


FIGURE 7.7: Comportements détectés dans les programmes sains

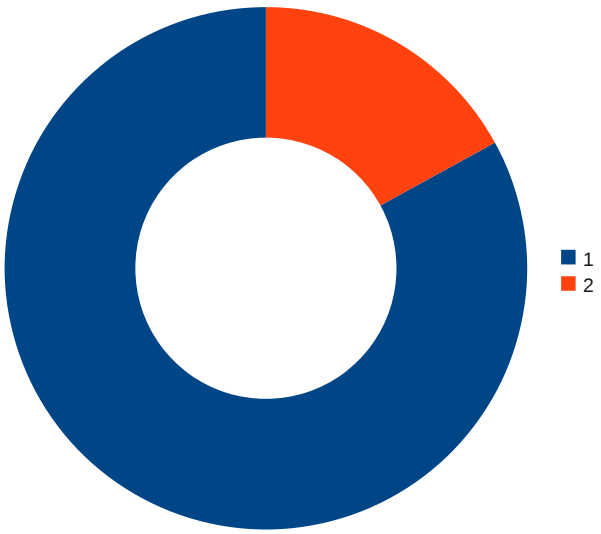
Comportement	Nombre de binaires	Proportion
Decrypt	61	15.72%
Blind	1	0.26%
Check	0	0.00%
Scrambled	0	0.00%

7.5 Conclusion

Nous avons mis en évidence au travers de tests sur des programmes variés que TraceSurfer permettait de détecter efficacement :

- la présence de code auto-modifiant (volontaire ou non) ;
- les signatures logiques de comportements liés à la protection de code. Nous avons pu confirmer expérimentalement que ces techniques étaient très présentes dans les programmes packés et les malwares, et très peu dans les programmes sains ;
- une différence de fonctionnement entre un programme packé et un programme compilé à la volée grâce à la mesure de largeur des auto-modifications.

FIGURE 7.8: Répartition par nombre de vagues



Conclusion

Nous avons montré au chapitre 1 que la recherche en virologie informatique suivait naturellement l'évolution technique des programmes malveillants. La notion de comportement, encore peu formalisée, a une importance croissante pour raisonner sur les malwares.

Nous avons pour cela proposé au chapitre 2 deux modèles de calcul simples, définis formellement, dans lesquels on intégrait la notion d'environnement et d'appel système. Ces modèles permettent de prendre en compte l'architecture des systèmes réels, qui constitue une contrainte dans le cadre de l'analyse de codes binaires.

Le chapitre 4 se base sur ces modèles pour introduire formellement la notion de comportement des programmes et de politique de sécurité. Nous avons montré une propriété fondamentale des politiques de sécurité qui découle du théorème de Rice : toute politique non-triviale sur les comportements est indécidable. Ce résultat permet notamment de généraliser le résultat d'indécidabilité de la détection de virale à n'importe quel comportement. Nous avons également vu qu'il était possible d'utiliser des approximations pour rendre les politiques de sécurité décidables, sous réserve d'interdire les programmes auto-modifiants.

Le chapitre 3 a présenté deux définitions des programmes auto-modifiants selon le modèle de calcul utilisé. Nous avons toutefois discuté l'unicité du phénomène étudié, et toutes les propriétés que nous avons montré par la suite étaient identiques pour les deux définitions de l'auto-modification. Cela implique que ces deux définitions caractérisent le même phénomène sous-jacent. Nous avons également montré une différence fondamentale du point de vue de la calculabilité entre les programmes statiques et les programmes auto-modifiants.

Le chapitre 5 propose une méthode pour la détection effective d'auto-modifications dans des programmes en langage machine. Cette méthode, qui repose sur un système de typage dynamique, nécessite toutefois d'exécuter (ou simuler) le programme analysé pour une entrée particulière. Il serait souhaitable de généraliser ce système de typage afin de raisonner statiquement sur les propriétés des programmes, mais nous avons vu que c'était impossible avec les modèles de calculs proposés et avec les modèles réels, à moins de les rendre triviaux. Il semble toutefois intéressant de chercher un modèle restreint, analysable statiquement, permettant de raisonner sur les binaires. Dans un tel système, on pourrait raisonner sur les programmes exécutables sans être confronté à la complexité de la simulation de programmes potentiellement malveillants.

Enfin, nous avons vu au chapitre 6 (et validé par l'expérience au chapitre 7)

différents cas où il était intéressant de détecter et analyser la présence de codes auto-modifiants pour la sécurité des applications. Ces différents scénarios mettent en évidence l'omniprésence de ces techniques au sein des problématiques de sécurité : parfois implémentées volontairement pour masquer des actions potentiellement malveillantes, souvent de manière accidentelle et indiquant la présence d'une vulnérabilité. Les codes auto-modifiants sont pourtant légitimes et parfois indispensables, car employés à bon escient, ils permettent d'améliorer de manière spectaculaire la performance des programmes. Paradoxalement, nous avons trouvé qu'une manière simple d'analyser les programmes auto-modifiants était d'utiliser de l'instrumentation par auto-modification, à la fois en JavaScript (section 4.3) et en x86 (section 6.1).

Les codes auto-modifiants sont des programmes au fonctionnement singulier dont le comportement est mal connu. Leur nature réflexive est difficile à appréhender sur un plan théorique, et ils sont complexes à construire et contrôler. Cette thèse constitue un effort de formalisation sur leur nature et leur impact sur la sécurité des applications, à la fois en bien et en mal.

Bibliographie

[ABEL05] M. Abadi, M. Budiu, Ú. Erlingsson & J. Ligatti – « Control-flow integrity », in *Proceedings of the 12th ACM conference on Computer and communications security*, ACM New York, NY, USA, 2005, p. 340–353. 69

[Adl88] L. Adleman – « An abstract theory of computer viruses », in *Advances in Cryptology – CRYPTO'88*, vol. 403, Lecture Notes in Computer Science, 1988. 7, 9

[Adv05] Advanced Micro Devices, Inc. – « AMD64 Virtualization Codenamed "Pacifica" Technology », Tech. report, 2005. 87

[Auc96] D. Aucsmith – « Tamper resistant software: An implementation », in *Information Hiding*, Springer, 1996, p. 317–333. 25

[Ayc03] J. Aycock – « A brief history of just-in-time », *ACM Computing Surveys (CSUR)* **35** (2003), no. 2, p. 113. 25

[BCK+10] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda & G. Vigna – « Efficient detection of split personalities in malware », in *Network and Distributed System Security (NDSS)*, 2010. 87

[BDB00] V. Bala, E. Duesterwald & S. Banerjia – « Dynamo: a transparent dynamic optimization system », in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, ACM, 2000, p. 12. 25

[Bel05] F. Bellard – « QEMU, a fast and portable dynamic translator », USENIX, 2005. 25, 41

[BKH+06] P. Baecher, M. Koetter, T. Holz, M. Dornseif & F. Freiling – « The nepenthes platform: An efficient approach to collect malware », in *Recent Advances in Intrusion Detection*, Springer, 2006, p. 165–184. 69

[BKM06] G. Bonfante, M. Kaczmarek & J. Marion – « On abstract computer virology from a recursion theoretic perspective », *Journal in computer virology* **1** (2006), no. 3, p. 45–54. 9

[BKM09] — , « Architecture of a morphological malware detector », *Journal in Computer Virology* **5** (2009), no. 3, p. 263–270. 68

[BMM06] D. Bruschi, L. Martignoni & M. Monga – « Detecting self-mutating malware using control-flow graph matching », *Detection of Intrusions and Malware & Vulnerability Assessment* (2006), p. 129–143. 68

[BMR09] G. Bonfante, J. Marion & D. Reynaud – « A computability perspective on self-modifying programs », *Software Engineering and Formal Methods* (2009). 23

[Boh08] L. Bohne – « Pandora's bochs: Automatic unpacking of malware », 2008. 72

[BPSZ08] D. Brumley, P. Poosankam, D. Song & J. Zheng – « Automatic patch-based exploit generation is possible: Techniques and implications », in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008, p. 143–157. 10

[Bru75] J. Brunner – *The Shockwave Rider*, Harper & Row, 1975. 3

[Bru04] D. Bruening – « Efficient, transparent, and comprehensive runtime code manipulation », Thèse, Massachusetts Institute of Technology, 2004. 41

[Bus03] G. Bush – *The National Strategy to Secure Cyberspace*, Morgan James Pub, 2003. 6

[CES86] E. Clarke, E. Emerson & A. Sistla – « Automatic verification of finite-state concurrent systems using temporal logic specifications », *ACM Transactions on Programming Languages and Systems (TOPLAS)* **8** (1986), no. 2, p. 263. 36

[CJS+05] M. Christodorescu, S. Jha, S. Seshia, D. Song & R. Bryant – « Semantics-aware malware detection », in *2005 IEEE Symposium on Security and Privacy*, 2005, p. 32–46. 8

[CKJ+05] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, H. Veith et al. – « Malware normalization », Tech. report, University of Wisconsin, 2005. 10

[CNR09] A. Chaudhuri, P. Naldurg & S. Rajamani – « A type system for data-flow integrity on Windows Vista », *ACM SIGPLAN Notices* **43** (2009), no. 12, p. 9–20. 59

[Cod68] E. Codd – *Cellular automata*, Academic Press, Inc. Orlando, FL, USA, 1968. 8

[Coh86] F. Cohen – « Computer viruses », Thèse, University of Southern California, 1986. 7, 8, 11, 38

[CPM+98] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle & Q. Zhang –

« StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks », in *Proceedings of the 7th conference on USENIX Security Symposium-Volume 7*, Usenix Association, 1998, p. 5. 76

[CR04] T. M. Chen & J.-M. Robert – « The evolution of viruses and worms », in *Statistical Methods in Computer Security*, 2004. 3

[CSV07] H. Cai, Z. Shao & A. Vaynberg – « Certified self-modifying code », *ACM SIGPLAN Notices* **42** (2007), no. 6, p. 77. 25

[Dmi01] M. Dmitriev – « Safe class and data evolution in large and long-lived Java [tm] applications », Thèse, University of Glasgow, 2001. 86

[DMR+09] R. Deibert, A. Manchanda, R. Rohozinski, N. Villeneuve & G. Walton – « Tracking GhostNet: Investigating a Cyber Espionage Network », Tech. report, Information Warfare Monitor, Munk Centre, March 2009. 6

[DRSL08] A. Dinaburg, P. Royal, M. Sharif & W. Lee – « Ether: Malware analysis via hardware virtualization extensions », in *Proceedings of the 15th ACM conference on Computer and communications security*, ACM, 2008, p. 51–62. 65, 72

[FC08] B. Ford & R. Cox – « Vx32: Lightweight user-level sandboxing on the x86 », in *2008 USENIX Annual Technical Conference*, 2008. 39

[Fer08] P. Ferrie – « Anti-unpacker tricks », in *Proc. of the 2nd International CARO Workshop*, 2008. 55, 72

[Fil05] E. Filiol – *Computer viruses: from theory to applications*, Springer, 2005. 8

[Fil07] — , *Techniques virales avancées*, Springer, 2007. 8, 55

[Fut99] Y. Futamura – « Partial evaluation of computation process - an approach to a compiler-compiler », *Higher-Order and Symbolic Computation* **12** (1999), no. 4, p. 381–391. 33

[Gar99] L. Garber – « Melissa virus creates a new type of threat », *Computer* **32** (1999), no. 6, p. 16–19. 4

[GCK05] J. Giffin, M. Christodorescu & L. Kruger – « Strengthening software self-checksumming via self-modifying code », in *21st Annual Computer Security Applications Conference*, 2005, p. 10. 25

[GFC08] F. Guo, P. Ferrie & T. Chiueh – « A study of the packer problem and its solutions », in *Recent Advances in Intrusion Detection*, Springer, 2008, p. 98–115. 72

[GGPS96] L. Goldberg, P. Goldberg, C. Phillips & G. Sorkin – « Constructing computer virus phylogenies », in *Combinatorial Pattern Matching*, Springer, 1996, p. 253–270. 10

[GLM+08] P. Godefroid, M. Levin, D. Molnar et al. – « Automated whitebox fuzz testing », in *Proceedings of the Network and Distributed System Security Symposium*, 2008. 9

[GMR09] W. Guizani, J. Marion & D. Reynaud – « Server-Side Dynamic Code Analysis », in *4th International Conference on Malicious and Unwanted Software*, 2009. 47, 87

[GPR04] T. Garfinkel, B. Pfaff & M. Rosenblum – « Ostia: A delegating architecture for secure system call interposition », in *Proceedings of the Network and Distributed Systems Security Symposium*, 2004, p. 187–201. 39

[Hee09] S. Heelan – « Automatic generation of control flow hijacking exploits », 2009. 10

[HSD+08] T. Holz, M. Steiner, F. Dahl, E. Biersack & F. Freiling – « Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm », in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, USENIX Association, 2008, p. 1–9. 5

[Hyp10] M. H. Hyppönen – « Les virus à l'assaut des téléphones mobiles », *Dossier pour la science n°66* (2010). 5

[Jac09] G. Jacob – « Malware behavioral models: bridging abstract and operational virology », Thèse, Université de Rennes 1, December 2009. 8

[Jon97] N. Jones – *Computability and complexity: from a programming perspective*, The MIT Press, 1997. 13, 17

[Jos09] S. Josse – « Analyse et détection dynamique de codes viraux dans un contexte cryptographique », Thèse, École Polytechnique, 2009. 72

[Kac08] M. Kaczmarek – « Des fondements de la virologie informatique vers une immunologie formelle », Thèse, Institut National Polytechnique de Lorraine, December 2008. 9, 23

[Kah87] G. Kahn – « Natural semantics », *Symposium on Theoretical Aspects of Computer Science* (1987), p. 22–39. 17

[KBA02] V. Kiriansky, D. Bruening & S. P. Amarasinghe – « Secure execution via program shepherding », in *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA), USENIX Association, 2002, p. 191–206. 39

[KKSV05] J. Kinder, S. Katzenbeisser, C. Schallhart & H. Veith – « Detecting malicious code by model checking », *Detection of Intrusions and Malware and Vulnerability Assessment* (2005), p. 174–187. 36

[Kle52] S. Kleene – *Introduction to metamathematics*, Wolters-Noordhoff, 1952. 9

[KPY07] M. Kang, P. Poosankam & H. Yin – « Renovo: A hidden code extractor for packed executables », in *Proceedings of the 2007 ACM workshop on Recurring malcode*, ACM, 2007, p. 53. 72

[KYH+09] M. Kang, H. Yin, S. Hanna, S. McCamant & D. Song – « Emulating emulation-resistant malware », in *Proceedings of the 1st ACM workshop on Virtual machine security*, ACM, 2009, p. 11–22. 87

[Lan84] C. Langton – « Self-reproduction in cellular automata », *Physica D: Nonlinear Phenomena* **10** (1984), no. 1-2, p. 135–144. 8

[Law96] K. Lawton – « Bochs: A portable pc emulator for unix/x », *Linux Journal* (1996), no. 29es, p. 7. 41

[LCM+05] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi & K. Hazelwood – « Pin: building customized program analysis tools with dynamic instrumentation », in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ACM, 2005, p. 190–200. 41, 65

[Les07] M. Lesk – « The new front line: Estonia under cyberassault », *IEEE Security & Privacy* (2007), p. 76–79. 7

[Lew10] J. A. Lewis – « The cyber war has not begun », Tech. report, Center for Strategies and International Studies, March 2010. 6

[Liv08] *Défense et Sécurité Nationale - Le Livre Blanc* – La Documentation Française, 2008. 6

[Lop08] M. A. Lopez – « Unpacking, a hybrid approach », in *2nd International Computer Antivirus Researchers Organization*, 2008. 72

[LY99] T. Lindholm & F. Yellin – *Java virtual machine specification*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999. 39, 86

[MCJ07] L. Martignoni, M. Christodorescu & S. Jha – « Omniunpack: Fast, generic, and safe unpacking of malware », in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007. 72

[MK10] J.-Y. Marion & M. Kaczmarek – « Boulevard du cybercrime », *Dossier pour la science n°66* (2010). 5

[MKK07] A. Moser, C. Kruegel & E. Kirda – « Exploring multiple execution paths for malware analysis », in *IEEE Symposium on Security and Privacy, 2007. SP'07*, 2007, p. 231–245. 9

[MPRB09] L. Martignoni, R. Paleari, G. Roglia & D. Bruschi – « Testing CPU emulators », in *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA), Chicago, Illinois, U.S.A.*, ACM, 2009, p. 261–272. 41

[MS01] D. Moore & C. Shannon – « The Spread of the Code-Red Worm », Tech. report, The Cooperative Association for Internet Data Analysis, 2001. 4

[Mus09] V. Mussot – « Transparence et instrumentation », Tech. report, Institut Universitaire de Technologie de Metz, 2009. 41, 86

[Mye99] A. Myers – « JFlow: Practical mostly-static information flow control », in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1999, p. 228–241. 39

[Myr09] M. Myreen – « Formal verification of machine-code programs », Thèse, University of Cambridge, 2009. 10

[Myr10] — , « Verified just-in-time compiler on x86 », *ACM SIGPLAN Notices* **45** (2010), no. 1, p. 107–118. 25

[NA09] S. Nagaraja & R. Anderson – « The snooping dragon: social-malware surveillance of the Tibetan movement », Tech. report, University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CLTR-746, 2009. 6

[Nec97] G. Necula – « Proof-carrying code », in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1997, p. 106–119. 39

[Neu66] J. V. Neumann – *Theory of self-reproducing automata*, University of Illinois Press, Champaign, IL, USA, 1966. 8

[NS03] N. Nethercote & J. Seward – « Valgrind: A Program Supervision Framework », *Electronic Notes in Theoretical Computer Science* **89** (2003), no. 2, p. 44–66. 41

[NSL+06] G. Neiger, A. Santoni, F. Leung, D. Rodgers & R. Uhlig – « Intel virtualization technology: Hardware support for efficient processor virtualization », *Intel Technology Journal* **10** (2006), no. 3, p. 167–177. 87

[OMR08] M. Oberhumer, L. Molnár & J. Reiser – « UPX: Ultimate Packer for eXecutables », (2008), http://upx.sourceforge.net. 25

[One96] A. One – « Smashing the stack for fun and profit », *Phrack magazine* **7** (1996), no. 49, p. 1996–11. 77

[Plo81] G. Plotkin – « A structural approach to operational semantics. Report DAIMI FN-19 », Tech. report, Computer Science Department, Aarhus University, 1981. 16

[PMR+09] R. Paleari, L. Martignoni, G. Roglia, D. Bruschi, U. di Milano & U. di Udine – « A fistful of red-pills: How to automatically generate procedures to detect CPU emulators », in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009. 41, 86

[PSY09] P. Porras, H. Saidi & V. Yegneswaran – « Conficker C Analysis », Tech. report, SRI International, 2009. 5

[QA08] D. Quist & C. Ames – « Temporal reverse engineering », *Blackhat USA, Las Vegas, Nevada* (2008). 72

[Ran08] M. Ranum – « Cyberwar is bull$#!t », in *Hack In The Box Security Conference*, 2008. 6

[Rey10] D. Reynaud – « Dynamic Binary Instrumentation: an Application to JavaScript Deobfuscation », *HITB Magazine* (2010), p. 12–16. 35, 43

[RHD+06] P. Royal, M. Halpin, D. Dagon, R. Edmonds & W. Lee – « Polyunpack: Automating the hidden-code extraction of unpack-executing malware », in *22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006, p. 289–300. 72

[RI00] J. Robin & C. Irvine – « Analysis of the Intel Pentium's ability to support a secure virtual machine monitor », in *Proceedings of the 9th conference on USENIX Security Symposium-Volume 9*, USENIX Association, 2000, p. 10. 87

[Ric53] H. Rice – « Classes of recursively enumerable sets and their decision problems », *Transactions of the American Mathematical Society* **74** (1953), no. 2, p. 358–366. 38

[RKK07] T. Raffetseder, C. Kruegel & E. Kirda – « Detecting system emulators », *Information Security* (2007), p. 1–18. 41, 87

[Rog67] H. Rogers – *Theory of recursive functions and effective computability*, McGraw-Hill New York, 1967. 31, 32

[Rut04] J. Rutkowska – « Red Pill... or how to detect VMM using (almost) one CPU instruction », *Invisible Things* (2004). 86, 88

[SBY+] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam & P. Saxena – « BitBlaze: A new approach to computer security via binary analysis », *Information Systems Security*, p. 1–25. 9

[SH82] J. F. Shoch & J. A. Hupp – « The "worm" programs - early experience with a distributed computation », *Commun. ACM* **25** (1982), no. 3, p. 172–180. 3

[SLGL09] M. Sharif, A. Lanzi, J. Giffin & W. Lee – « Automatic reverse engineering of malware emulators », in *2009 30th IEEE Symposium on Security and Privacy*, IEEE, 2009, p. 94–109. 72

[Smu94] R. Smullyan – *Diagonalization and self-reference*, Clarendon press, 1994. 23

[Spa89] E. H. Spafford – « The Internet worm program: an analysis », *SIGCOMM Comput. Commun. Rev.* **19** (1989), no. 1, p. 17–57. 4, 76

[Szo05] P. Szor – *The art of computer virus research and defense*, Addison-Wesley Professional, 2005. 24, 25

[VIS96] D. Volpano, C. Irvine & G. Smith – « A sound type system for secure flow analysis », *Journal of computer security* **4** (1996), no. 2/3, p. 167–188. 59

[YS10] H. Yin & D. Song – « TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution », (2010). 65

[YSD+09] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula & N. Fullagar – « Native client: A sandbox for portable, untrusted x86 native code », *30th IEEE Symposium on Security and Privacy* (2009). 39, 40

[YSE+07] H. Yin, D. Song, M. Egele, C. Kruegel & E. Kirda – « Panorama: Capturing system-wide information flow for malware detection and analysis », in *Proceedings of the 14th ACM conference on Computer and communications security*, ACM, 2007, p. 127. 68

INSTITUT NATIONAL
POLYTECHNIQUE
DE LORRAINE

AUTORISATION DE SOUTENANCE DE THESE
DU DOCTORAT DE L'INSTITUT NATIONAL
POLYTECHNIQUE DE LORRAINE

o0o

VU LES RAPPORTS ETABLIS PAR :
Monsieur Marc DACIER, Professeur, EUROCOM, Valbonne Sophia Antipolis
Monsieur Marc POUZET, Professeur, Université Paris-Sud 11, Orsay

Le Président de l'Institut National Polytechnique de Lorraine, autorise :

Monsieur REYNAUD-PLANTEY Daniel

NANCY BRABOIS
2, AVENUE DE LA
FORET-DE-HAYE
BOITE POSTALE 3
F - 54501
VANDŒUVRE CEDEX

à soutenir devant un jury de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée :

"Analyse de codes auto-modifiants pour la sécurité logicielle"

en vue de l'obtention du titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

Spécialité : **« Informatique »**

Fait à Vandoeuvre, le 06 octobre 2010
Le Président de l'I.N.P.L.,
F. LAURENT



TEL. 33/03.83.59.59.59
FAX. 33/03.83.59.59.55

Résumé. Les programmes auto-modifiants fonctionnent de manière singulière car ils sont capables de réécrire leur propre code en cours d'exécution. Absents des modèles de calcul théoriques, ils sont pourtant omniprésents dans les ordinateurs et les systèmes d'exploitations actuels. Ils sont en effet utilisés par les chargeurs d'amorçages, pour la compilation à la volée ou encore l'optimisation dynamique de code. Ils sont également omniprésents dans les programmes malveillants, dont les auteurs ont bien compris qu'ils constituaient des objets complexes à analyser. Ils sont enfin virtuellement présents dans tous les autres programmes mais de manière non-intentionnelle. En effet, on peut voir certaines classes de vulnérabilités, par exemple les failles par débordement de tampon, comme la possibilité d'exécuter accidentellement des données – ce qui est un comportement caractéristique des programmes auto-modifiants.

Dans cette thèse, nous proposons un modèle théorique permettant de caractériser un certain nombre de comportements auto-modifiants avancés. Nous mettons également au point un prototype, TraceSurfer, permettant de détecter efficacement ces comportements à partir de l'analyse de traces et de les visualiser sous forme de graphes d'auto-référence. Enfin, nous validons par l'expérience à la fois le modèle théorique et l'outil en les testant sur un grand nombre de programmes malveillants.

Mots clés : *Analyse binaire, comportement, politique de sécurité, programmes malveillants, recherche de vulnérabilités, auto-référence.*

Abstract. Self-modifying programs run in a very specific way: they are capable to rewrite their own code at runtime. Remarkably absent from theoretical computation models, they are present in every modern computer and operating system. Indeed, they are used by bootloaders, for just-in-time compilation or dynamic optimizations. They are also massively used by malware authors in order to bypass antivirus signatures and to delay analysis. Finally, they are unintentionally present in every program, since we can model code injection vulnerabilities (such as buffer overflows) as the ability for a program to accidentally execute data.

In this thesis, we propose a formal framework in order to characterize advanced self-modifying behaviors and code armoring techniques. A prototype, TraceSurfer, allows us to detect these behaviors by using fine-grained execution traces and to visualize them as self-reference graphs. Finally, we assess the performance and efficiency of the tool by running it on a large corpus of malware samples.

Keywords: *Binary analysis, behaviour, security policy, malware, vulnerability research, self-reference.*