HAL
open science

# Analyse et détection dynamique de codes viraux dans un contexte cryptographique

Sébastien Josse

## To cite this version:

Sébastien Josse. Analyse et détection dynamique de codes viraux dans un contexte cryptographique. Computer Science [cs]. Ecole Polytechnique X, 2009. English. NNT : . pastel-00005279

**HAL Id: pastel-00005279**

**https://pastel.hal.science/pastel-00005279**

Submitted on 21 Jul 2009

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# THÈSE

présentée à

L'ÉCOLE POLYTECHNIQUE

pour obtenir le titre de

DOCTEUR

Specialité

MATHÉMATIQUES APPLIQUÉES ET INFORMATIQUE

par

Sébastien JOSSE

Titre de la thèse

## Analyse et détection dynamique de code viraux dans un contexte cryptographique

*et application à l'évaluation de logiciels antivirus*

Soutenue le 10 Avril 2009 devant le jury composé de :

| | | | | |
|---|---|---|---|---|
| Président du jury | Jean-Marc | Steyaert | Professeur | Ecole Polytechnique |
| Directeur de thèse | Éric | Filiol | Professeur | ESIEA LAVAL |
| Rapporteurs | Hervé | Debar | Professeur | France Telecom R&D |
| | Jean-Jacques | Quisquater | Professeur | UCL |
| Examinateurs | Robert | Erra | Docteur | ESIEA PARIS |
| | Caroline | Fontaine | Docteur | IRISA |
| | Cédric | Lauradoux | Docteur | UCL |
| | Frédéric | Raynal | Docteur | Sogeti ESEC |

# Remerciements

Les années consacrées à cette thèse constituent à mes yeux une expérience aussi éprouvante que passionnante. Réalisée sur mon temps libre, alors que je mène à temps plein une vie professionnelle et familiale déjà très dense, j'ai souvent douté pouvoir la mener à son terme. Aux vues des obstacles franchis et à présent que l'exercice est terminé, je prends conscience du soutien important dont nombre de collègues, proches et amis m'ont gratifié. À tous je veux dire ma gratitude.

Je voudrais remercier en premier lieu mon directeur de thèse, Eric Filiol, pour avoir guidé mes premiers pas dans les arènes de la recherche. C'est son expérience, sa confiance et son amitié qui m'ont permis de mener à terme cette aventure. J'ai particulièrement apprécié sa franchise et son soutien inconditionnel face aux difficultés de tous ordres qui peuvent jalonner la route du thésard. Je tiens également à le remercier de son implication et de la disponibilité dont il a su faire preuve dans la direction de mon travail. Enfin, Eric a su me communiquer une partie de la passion scientifique qui l'habite et de son goût du travail, parfois acharné. J'espère avoir l'opportunité de poursuivre avec lui cette collaboration le plus longtemps possible.

Mes remerciements vont ensuite aux autres membres du jury, pour avoir bien voulu lire et valider mes travaux. À Jean-Jacques Quisquater et Hervé Debar je veux dire ma gratitude pour m'avoir fait l'honneur d'être les rapporteurs de cette thèse. Je les remercie pour leur bienveillance et leurs conseils avisés. Je remercie aussi très chaleureusement Jean-Marc Stayaert pour avoir bien voulu présider le jury. Sa bonne humeur contagieuse m'a permis de passer cette épreuve dans les meilleurs conditions. Je remercie enfin Robert Erra, Frédéric Raynal, Caroline Fontaine et Cédric Lauradoux pour leur examen attentif du manuscrit et la qualité de leurs questions et remarques, qui devraient me permettre de progresser dans mes futures activités de recherche.

Je voudrais remercier Guillaume Dabosville et Guillaume Geffard, avec qui j'ai eu l'opportunité de collaborer sur plusieurs travaux. Leur décontraction et leur amitié m'ont été précieuses.
Je remercie également l'ensembles des membres du Laboratoire de Virologie et Cryptologie Opérationnelles, avec qui j'ai pu collaborer, et en particulier Philippe Beaucamps et Grégoire Jacob pour les intéressantes conversations que nous avons eu sur le sujet.

Je tiens à remercier Dominique Chauveau, Lionel Van Aertryck, Alexandre Gazet et David Boucher pour avoir pris le temps de relire certaines de mes soumissions. Leurs remarques m'ont clairement permis d'en améliorer la qualité.
Je remercie Alain Marcay et Benoît Jeannin pour avoir respectivement encouragé puis rendu possible le financement de certains déplacements, parfois à l'autre bout du monde.

Je remercie également les personnes des secteurs civil et de Défense avec qui j'ai eu l'occasion de travailler ces trois dernières années au cours de missions d'étude sur les problématiques de protection logicielle et de rétro-ingénierie, pour l'enrichissement que m'a apportée leur collaboration. Je pense en particulier à François Daudé, Olivier Vivolo, Gaëtan Le Gouelvic, Antoine Monsifrot et Briag Morange.

Je remercie enfin ma femme, Stéphanie, qui a su s'armer de patience durant la rédaction de cette thèse. Puisse-t-elle me pardonner ce temps que je ne lui ai pas consacré.

# Introduction

## Motivations

L'utilisateur final d'un produit antivirus ne sait pas toujours quelle confiance il peut placer dans son produit antivirus pour parer convenablement la menace virale. Pour répondre à cette question, il est nécessaire de formuler la problématique de sécurité à laquelle doit répondre un tel produit et de disposer d'outils et de critères méthodologiques, techniques et théoriques permettant d'évaluer la robustesse des fonctions de sécurité et la pertinence des choix de conception au regard d'une menace virale identifiée.

Une bonne compréhension de la menace virale rend indispensable une veille technologique continue. Nous avons observé ces dernières années une tendance à l'utilisation de plus en plus systématique d'applications de protection logicielle du marché par les auteurs de virus. Par ailleurs, la mise à la disposition du plus grand nombre[1] de générateurs automatiques de variantes d'un même programme viral (kits de génération de virus) de plus en plus évolués [Szö05] impose un nouveau défi à relever pour les éditeurs de produits antivirus et une remise en cause profonde de leur choix de conception. Je pense que ces outils ne sont pas encore représentatifs de l'état de l'art dans le domaine de la protection logicielle, dont l'essor récent est probablement dû aux nouvelles applications civiles de protection de contenu (gestion numérique des droits). Si les concepteurs de programmes malicieux utilisent volontier les packers d'exécutables, ils n'utilisent pas encore (pour les codes viraux identifiés) d'outils plus évolués, comme les chaînes de compilation spécialisées, dont la distribution reste encore très discrétionnaire. Nous avons choisi d'analyser la menace potentielle que représenterait l'utilisation de ce type d'outils spécialisés par les auteurs de virus.

Les programmes viraux font un usage intensif de la cryptographie, tant pour contrôler l'intégrité de leur code ou de leur environnement d'exécution, que pour protéger leurs données critiques en confidentialité et échapper à la détection. Ces mécanismes crypographiques doivent être évalués dans un contexte cryptographique particulier, appelé contexte d'attaque boîte blanche, dans lequel l'environnement d'exécution est supposé complètement maîtrisé par l'attaquant. Dans ce contexte cryptographique, l'attaquant est en mesure de mener une analyse dynamique du programme viral et d'instrumenter le système d'exploitation.

L'identification de la menace virale et sa modélisation sont des étapes cruciales avant toute organisation d'une ligne de défense. L'évaluation d'un produit anti-virus doit se faire sur la base de l'énoncé d'une problématique de sécurité, permettant de définir des objectifs de sécurité. Un tel produit propose plusieurs fonctions de sécurité, permettant de couvrir les objectifs de sécurité.
Parmi les différentes fonctions de sécurité proposées par un produit antivirus, l'une d'entre elles revêt une importance cruciale et concentre le savoir-faire des éditeurs de produits antivirus : la fonction de détection virale.

Les schémas 1 et 2 présentent de manière synthétique l'architecture générale et le fonctionnement du moteur de détection virale d'un produit antivirus. Nous pouvons remarquer que le processus de détection se fonde toujours sur une base de connaissance.

[1] Près de 200 kits de génération de virus sont téléchargeables sur le site web Vx Heavens [Hea08].

Nous avons représenté sur le schéma 1 le processus d'élaboration du schéma de détection d'un virus. Nous pouvons observer que l'extraction d'information est le plus souvent précédée d'une étape d'analyse, automatique ou manuelle, lors de laquelle la protection fournie par un outil du marché est supprimée ou contournée (unpacking), puis d'une phase de désassemblage, permettant d'extraire une représentation intermédiaire du code. Cette représentation est ensuite normalisée, pour aboutir à un schéma de détection, destiné à être stocké dans une base d'information. Nous adoptons la définition suivante d'un schéma de détection :

**Définition 0.0.1** *(Schéma de détection [Fil06]). Un schéma de détection d'un malware $\mathcal{M}$ est défini comme la donnée d'un motif et d'une fonction de détection $\{\mathcal{S}_{\mathcal{M}}, f_{\mathcal{M}}\}$.*

Nous pouvons voir une base de signatures virales comme un ensemble de schémas de détection $\{\mathcal{S}_{\mathcal{M}}, f_{\mathcal{M}}\}$. Cette base de signatures peut être constituée de différents types d'informations :
- des séquences d'octets ;
- des séquences d'instructions assembleur ;
- des séquences d'appels à des fonctions externes du système d'exploitation ;
- des données liées aux interactions du programme avec certains objets du système d'exploitation, comme le système de fichiers ou la base de registre ;
- des graphes de flots de contrôle, c'est-à-dire la représentation sous forme de graphes des branchements entre blocs de code de programmes (ou de portions de programmes).

Un schéma de détection d'un virus $\mathcal{M}$ peut être vu [Fil07b] comme la donnée :
- du langage $L_{\mathcal{M}}$ produit par une grammaire $G_{\mathcal{M}}$,
- d'un automate à états finis, capable de reconnaître le langage $L_{\mathcal{M}}$.

Le langage $L_{\mathcal{M}}$ peut être constitué de séquences d'octets ou d'instructions assembleur. Il peut être constitué de séquences d'appels à des fonctions externes. Il peut être constitué de termes représentant des graphes de flots de contrôle [BKM08].
Avec ce formalisme (celui des grammaires formelles), le problème de détection se ramène au problème d'appartenance d'un mot au langage $L_{\mathcal{M}}$. La fonction de détection $f_{\mathcal{M}}$ est un automate à états finis capable de reconnaître le langage $L_{\mathcal{M}}$, ce dernier définissant le motif de détection $\mathcal{S}_{\mathcal{M}}$.

Un schéma de détection d'un virus $\mathcal{M}$ peut être vu [FJ07] comme la donnée :
- de la distribution d'une loi de probabilité caractérisant l'information ou d'un modèle $\lambda_{\mathcal{M}}$ formé sur des données d'apprentissage (c'est-à-dire que les paramètres du modèle sont estimés sur les données d'apprentissage) ;
- d'une règle de décision générée sur la base de données d'apprentissage, permettant d'évaluer la vraisemblance d'une observation, étant donné le modèle $\lambda_{\mathcal{M}}$.

Avec ce formalisme (celui des modèles combinatoires ou probabilistes), le problème de détection se ramène au problème de la vraisemblance d'une observation, étant donné un modèle $\lambda_{\mathcal{M}}$. La fonction de détection $f_{\mathcal{M}}$ est un algorithme de classification ou un test, caractérisé par un seuil, permettant de reconnaître le fait d'être régit par une loi ou un modèle $\lambda_{\mathcal{M}}$, ce dernier définissant le motif de détection $\mathcal{S}_{\mathcal{M}}$.

Un schéma de détection d'un virus $\mathcal{M}$ peut être vu [DPCJD07] comme la donnée :
- d'une information utile $\alpha([\![\mathcal{M}]\!])$ extraite de la sémantique $[\![\mathcal{M}]\!]$ du virus $\mathcal{M}$ ;
- d'une fonction d'abstraction $\alpha$ qui extrait l'information sémantique utile des programmes.

Avec ce formalisme (celui de l'interprétation abstraite), le problème de la détection se ramène au problème de l'approximation des sémantiques concrètes d'un programme, par le biais d'une fonction d'abstraction $\alpha$. La fonction de détection $f_{\mathcal{M}}$ est une fonction d'abstraction, permettant de reconnaître le fait pour un programme d'exhiber certains comportements qui, sous l'abstraction $\alpha$, correspondent aux comportements du virus. L'abstraction des comportements du virus définissent le motif de détection $\mathcal{S}_{\mathcal{M}}$.

Il est fondamental que le processus d'extraction d'information puisse permettre, à partir de l'ensemble des virus d'une même souche, de définir un unique schéma de détection. Pour y parvenir, ce processus doit comprendre une phase de normalisation, permettant d'obtenir un unique schéma de détection pour tous les virus d'une même

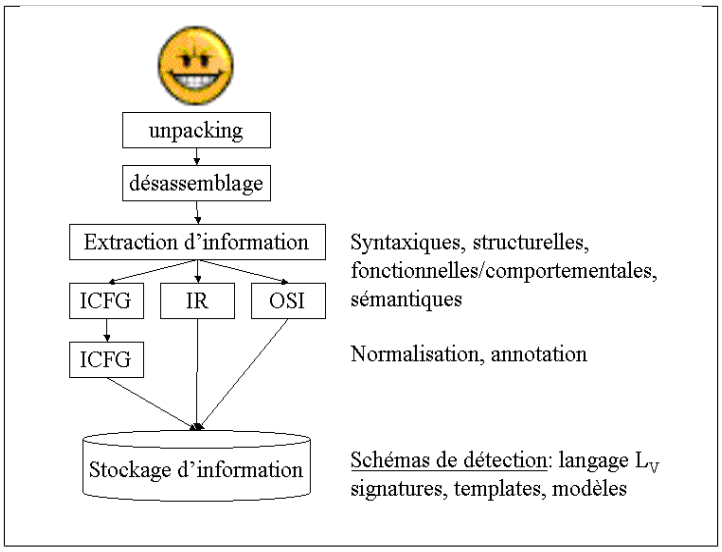



FIG. 1 – Moteur de détection : de l'extraction d'information au schéma de détection

famille virale. Pour y parvenir, des transformations doivent être appliquées sur la représentation intermédiaire du code. L'ensemble de ces règles de réécriture constitue une stratégie de réécriture [Vis01].

Un ensemble de règles de transformation induit une relation de réécriture. Si la relation est confluente et qu'elle se termine, alors il y a une unique forme normale pour chaque programme. Ce n'est malheureusement pas le cas en général : un ensemble de règles de transformation peuvent donner naissance à :

- des branches infinies (consider par exemple l'*inlining* d'une fonction récursive),
- des inversions (une transformation est défaite, par exemple en appliquant des règles de distribution ou de commutativité),
- des situations de non-confluence, situations dans lesquelles un programme peut être transformé en deux programmes différents.

Nous voyons donc que le choix d'une stratégie de réécriture est un élément fondamental dans la conception d'un moteur de détection.

Nous avons représenté sur le schéma 2 le synoptique de fonctionnement général d'une fonction de détection. Le processus de détection extrait l'information du programme $P$ et décide que le programme $P$ est infecté lorsqu'il existe une signature $\mathcal{S}_{\mathcal{M}}$ dans la base d'information telle que :

$$f_{\mathcal{M}}(P, \mathcal{S}_{\mathcal{M}}) = 1.$$

Dans le cadre des grammaires formelles, le processus de détection consiste à transformer le programme $P$ en un mot $x$, en utilisant le même procédé d'extraction d'information que celui utilisé pour extraire la signature $\mathcal{S}_{\mathcal{M}}$ d'un virus $\mathcal{M}$, puis à vérifier si $x \in \mathcal{S}_{\mathcal{M}}$, c'est-à-dire si $x$ appartient au langage $L_{\mathcal{M}}$. La fonction de détection $f_{\mathcal{M}}(P, L_{\mathcal{M}}) = 1$ si et seulement si elle reconnaît $x$ comme appartenant au langage $L_{\mathcal{M}}$.

Dans le cadre probabiliste, le processus de détection consiste à extraire une distribution statistique du programme $P$ ou de ses interactions avec son environnement, en utilisant le même procédé d'extraction d'information que celui utilisé pour extraire l'information $\lambda_{\mathcal{M}}$, puis à vérifier que cette distribution ne s'écarte pas de manière trop significative de la distribution $\lambda_{\mathcal{M}}$. La fonction $f_{\mathcal{M}}(P, \lambda_{\mathcal{M}}) = 1$ si et seulement si la vraisemblance de l'observation $x$ dépasse un certain seuil, étant donné le modèle $\lambda_{\mathcal{M}}$.

Dans le cadre formel de l'interprétation abstraite, le processus de détection consiste à extraire l'information sémantique du programme $P$, par le biais de la fonction d'abstraction $\alpha$ déjà utilisée pour extraire l'information utile

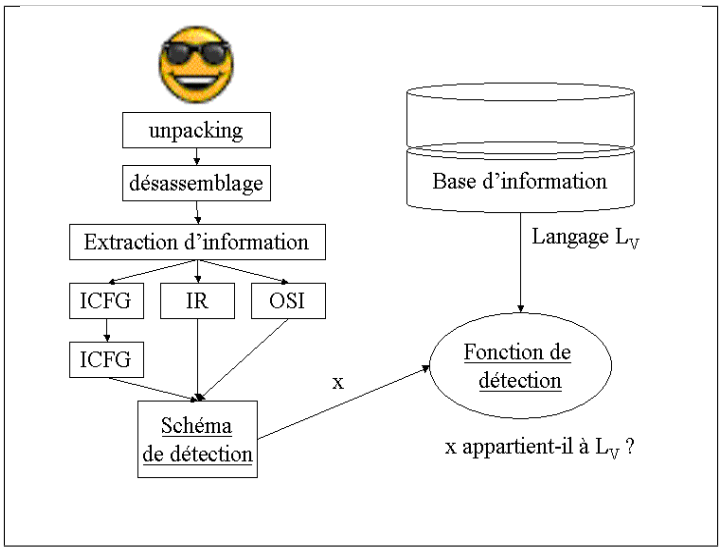



FIG. 2 – Moteur de détection : la fonction de détection

du virus $\mathcal{M}$, puis à vérifier que $\alpha(S[\![\mathcal{M}]\!]) \subseteq \alpha(S[\![P]\!])$.

Même si une partie des actions effectuées lors de l'analyse d'un programme viral peut être automatisée, les phases d'extraction d'informations et d'élaboration d'un schéma de détection (destiné à être stocké dans une base d'information) sont toujours en partie manuelles. Il est souvent nécessaire de maintenir plusieurs populations :

- une population de programmes viraux, classés par catégories (même langage de programmation par exemple) et par famille (même souche virale ou variantes) ;
- une population de programmes bénins, permettant d'élaborer des schémas de référence correspondant à un comportement normal. Cette population est utilisée pour étalonner la fonction de détection, sur la base de données d'apprentissage ou de test. Elle est utilisée par exemple pour déterminer empiriquement un seuil $S$ à partir duquel la classification opère sans erreur.

À l'issue de cette phase manuelle d'analyse, il est possible de renseigner une base de connaissance, décrivant les dommages causés par un virus, puis de proposer aux administrateurs système des contre-mesures permettant de se prémunir de cette menace virale et lorsque c'est possible une procédure de détection/désinfection.

Les processus de rétro-ingénierie et de détection de code viral sont donc différents, même si une grande partie des tâches automatisables est commune. Les problématiques associées sont les suivantes :

- dans le cas de l'analyse de code viral, le processus consiste à caractériser et à contourner ou supprimer les protections du code viral, avant d'extraire l'information permettant de l'identifier. Les moyens mis en oeuvre pour y parvenir peuvent être coûteux en terme de ressources (temps processeur et espace mémoire) ;
- dans le cas de la détection virale, le processus consiste à utiliser cette information pour reconnaître un virus. Les moyens mis en oeuvre pour y parvenir sont contraints en terme de ressources.

## Correction et complétude

Les méthodes d'analyse de programmes peuvent être classées en fonction de l'ensemble des propriétés qu'elles peuvent établir avec une certaine confiance. Deux propriétés fondamentales permettent de les caractériser : la correction et la complétude. Les concepts de correction et de complétude ont un sens précis en logique mathématique :

- un système de preuves est dit correct s'il ne prouve que des sentences vraies ;
- il est dit complet s'il prouve toutes les sentences vraies.

En d'autres termes, une analyse est correcte si et seulement si prouvable($P$) $\Rightarrow$ vraie($P$) et complète si et seulement si vraie($P$) $\Rightarrow$ prouvable($P$). Pour qualifier en ces termes l'analyse d'un programme, nous devons préciser ce que l'on cherche à prouver. Nous pouvons par exemple désirer prouver qu'un programme comporte une erreur [SC07]. Avec ce choix, une analyse est correcte pour montrer qu'un programme comporte une erreur si et seulement si elle est complète pour monter qu'un programme n'en comporte pas (correction et complétude correspondent à des implications dites converses). La correction d'une telle analyse correspond à l'absence de faux positif : nous pouvons certifier la présence d'erreurs. En revanche, l'analyse peut affirmer à tort qu'il n'y a pas d'erreur.
Correction :

$$\begin{array}{rcl} \text{Affirmer(P comporte une erreur)} & \Rightarrow & \text{P comporte une erreur} \\ \text{P ne comporte pas d'erreur} & \Rightarrow & \text{Affirmer(P ne comporte pas d'erreur)} \end{array}$$

A contrario, et c'est le choix le plus souvent rencontré dans la littérature relative à la détection d'erreur [XNHA05], nous pouvons vouloir prouver qu'une méthode d'analyse permet de détecter toutes les erreurs d'une certaine classe (ou de manière équivalente certifier qu'un programme n'en comporte pas). Avec cette définition de la correction d'une analyse, nous certifions l'absence de faux négatif.
Correction :

$$\begin{array}{rcl} \text{Affirmer(P ne comporte pas d'erreur)} & \Rightarrow & \text{P ne comporte pas d'erreur} \\ \text{P comporte une erreur} & \Rightarrow & \text{Affirmer(P comporte une erreur)} \end{array}$$

Ces concepts peuvent être appliqués au contexte de la compilation et des transformations d'optimisation : ici, la notion de correction est souvent comprise au sens de préservation des sémantiques du programme, ce qui est la principale exigence imposée à un compilateur. Elle s'applique en premier lieu à l'analyse statique des programmes. Dans ce cas, une sur-approximation des sémantiques du programme sert de base au raisonnement et garantit la correction d'une transformation imposée à notre vision abstraite du programme. Cependant, si la fonction abstraite est correcte, elle peut ne pas préserver la précision de la fonction concrète qu'elle approche. Certaines optimisations potentielles ne seront pas appliquées.
À l'inverse, il est possible d'imposer des transformations d'optimisation agressives, au détriment de la conservation des sémantiques du programme. L'analyse peut affirmer à tort la possibilité d'une optimisation. Elle sera complète si elle certifie que toutes les optimisations potentielles ont été détectées.
Correction :

$$\begin{array}{rcl} \text{Affirmer(Optimisation possible)} & \Rightarrow & \text{Optimisation possible} \\ \text{Optimisation impossible} & \Rightarrow & \text{Affirmer(Optimisation impossible)} \end{array}$$

Complétude :

$$\begin{array}{rcl} \text{Affirmer(Optimisation impossible)} & \Rightarrow & \text{Optimisation impossible} \\ \text{Optimisation possible} & \Rightarrow & \text{Affirmer(Optimisation possible)} \end{array}$$

Ces notions s'étendent naturellement au contexte de la détection virale.

- Un schéma de détection *correct* permet de s'assurer qu'un programme bénin ne sera pas considéré à tort comme malicieux. Le langage décrivant cette famille virale est suffisamment précis pour éviter ce risque, appelé risque de première espèce ($\alpha$, le risque de fausse alarme, est proche de zéro). La fonction de détection ne reconnaît que les mots appartenant à ce langage. Cependant, la complétude de la fonction de détection se basant sur ce schéma de détection peut être insuffisante pour détecter toutes les variantes du virus à cette obfuscation près.
- Un schéma de détection *complet* vis-à-vis d'une classe de transformations d'obfuscation garantit que toutes les variantes du virus seront détectées. Le langage décrivant cette famille virale est complet, dans la mesure où toutes les variantes possibles du virus appartiennent à ce langage. Le langage décrivant cette famille virale est suffisamment complet pour éviter le risque de seconde espèce ($\beta$, le risque de non détection, est proche de zéro). Cependant, la fonction de détection peut être trop grossière pour éviter le risque de première espèce. Le risque de première espèce est que des programmes bénins appartiennent également à ce langage.

Nous voyons donc que les notions de correction et de complétude sont suffisamment générales pour trouver un sens dans le cadre de différents formalismes, tous très utiles pour qualifier un algorithme d'analyse ou de détection de code :

- les grammaires formelles,

- l'interprétation abstraite,
- les modèles combinatoires ou probabilistes.

Ces concepts sont traduits en critères et exigences portant sur les fonctions d'extraction d'informations et de détection et nous servent de base, en terme de terminologie, pour formaliser de manière unifiée les axes d'effort de la lutte antivirale. Remarquons que ces concepts s'appliquent également à la caractérisation d'autres systèmes de détection que les antivirus, notamment les systèmes de détection/prévention d'intrusion (ID/PS). Une analyse complète des différences, similarités et de la complémentarité de ces approches est donnée dans [MM07]. Selon les auteurs, un antivirus est à rapprocher d'un HIPS utilisant une méthode de détection d'intrusion basée sur la connaissance (*knowledge* ou *misuse-based*).

## Autres critères

D'autres critères ont été proposés pour qualifier un schéma de détection [Fil06]. Parmi ceux-ci, nous pouvons citer notamment :

- l'information mutuelle ou transinformation d'un schéma de détection $\{\mathcal{S}_\mathcal{M}, f_\mathcal{M}\}$, qui décrit l'incertitude qu'un analyste doit affronter lors d'une analyse en boîte noire relativement à un moteur de détection $\mathcal{D}$ donné, pour extraire le schéma de détection $\{\mathcal{S}_\mathcal{M}, f_\mathcal{M}\}$ : la transinformation décrit et quantifie ce qu'apporte à l'analyste le fait de disposer de $\mathcal{D}$ ;
- la rigidité $\mathcal{R}(\mathcal{S}_\mathcal{M}, f_\mathcal{M})$ d'un schéma de détection $\{\mathcal{S}_\mathcal{M}, f_\mathcal{M}\}$, définie comme la difficulté plus ou moins grande qu'aura un plagiaire à le contourner.

Cette seconde notion me paraît suffisamment générale pour pouvoir être exprimée dans différents formalismes. Nous verrons au chapitre 7 qu'elle peut s'exprimer de manière équivalente dans le cadre des modèles statistiques par la notion de simulabilité des tests statistiques. Elle correspond à la difficulté, pour un attaquant, d'extraire un schéma de contournement exploitable :

**Définition 0.0.2** *(Schéma de contournement [Fil06]). Un schéma de contournement d'un malware $\mathcal{M}$ est défini comme la donnée d'un motif et d'une fonction de contournement $\{\mathcal{S}_\mathcal{M}, \overline{f_\mathcal{M}}\}$.*

La fonction de contournement $\overline{f_\mathcal{M}}$ est en fait la négation de la fonction $f_\mathcal{M}$, soit $1 \oplus f_\mathcal{M}$. Cette fonction décrit les différentes possibilités de modification pouvant être effectuées sur $\mathcal{S}_\mathcal{M}$ pour contourner un schéma de détection donné.

## Problématique

Le problème de l'évaluation d'un programme antivirus est le thème central de cette thèse. Cette évaluation doit se faire au regard d'une menace virale identifiée. Parmi les mécanismes implémentés par les programmes viraux pour masquer leur fonctionnement ou échapper à la détection, certains sont de nature cryptographique, d'autres ne le sont pas (techniques de furtivité, de diversification). Partant du constat que l'analyse des programmes viraux est menée dans un environnement potentiellement totalement contrôlé par l'analyste, il convient de définir un modèle de l'attaquant permettant de représenter notamment la possibilité pour l'analyste :

- d'instrumenter le système hôte ;
- de faire usage de méthodes dynamiques (ou hybrides statique/dynamique).

Le contexte WBAC (*White Box Attack Context*) est un contexte cryptographique, permettant de définir un tel modèle de l'attaquant et de qualifier les mécanismes cryptographiques au regard de cet attaquant. Je pense que c'est dans ce contexte que la résistance d'un mécanisme viral à la cryptanalyse ou à la rétro-ingénierie doit être évaluée.
La question de la pertinence de la cryptographie boîte blanche dans le contexte viral est étudiée au chapitre 2.
Cette thèse traite la double problématique de l'analyse et de la détection de code viral. J'ai pris le parti de focaliser sur les méthodes dynamiques d'extraction d'information fondées sur une émulation du matériel supportant le système d'exploitation.

Concernant l'analyse d'efficacité en boîte blanche (i.e. sur la base des spécifications détaillées) d'un moteur de détection, il est parfois utile de considérer que les résultats des opérations suivantes :
- unpacking ;
- désassemblage ;
- extraction du graphe de flot ;
- détection du junk code.

sont fournis par des oracles. Cela permet de centrer l'analyse d'efficacité du moteur de détection sur l'algorithme de détection, indépendamment de ces problèmes difficiles à résoudre de manière acceptable sans recours à des heuristiques.
J'ai pris le parti de ne pas négliger les problèmes de *correction* et de *complétude* de ces opérations, dont la bonne réalisation est un pré-requis indispensable lorsque l'on souhaite valider par le test l'efficacité de l'implémentation d'un algorithme de détection.

Les problèmes du désassemblage et de l'extraction du graphe CFG d'un exécutable sont traités dans le chapitre 3. Le chapitre 5 traite notamment du problème de l'unpacking d'un exécutable. Le chapitre 6 est dédiée au problème de récupération d'information d'interaction bas niveau avec l'OS indispensable à l'analyse des programmes furtifs.

Une évaluation du moteur de détection selon des critères objectifs requiert un effort de modélisation. La modélisation par interprétation abstraite permet de capturer les aspects sémantiques de l'information extraite d'un programme, ainsi que de valider la correction et l'efficacité des transformations imposées à cette information. Elle permet en particulier de formaliser l'analyse statique et dans une certaine mesure la rétro-ingénierie par utilisation de méthodes d'analyse dynamique ou hybride statique/dynamique. Ce formalisme est présenté dans le chapitre 4.
Une modélisation statistique du problème de détection offre une vision complémentaire et s'applique à l'analyse de comportements viraux, sur la base d'informations statistiques. Elle permet notamment de capturer les aspects liés aux interaction du programme avec son environnement. Ce formalisme est présenté dans le chapitre 7.
Je présente dans le chapitre 8 une revue des critères permettant de qualifier la fonction de détection en termes de correction (pas de faux positif, i.e. pas de fausse alarme) et de complétude (pas de faux négatif, i.e. pas de non-détection) vis-à-vis des transformations d'obfuscation. On s'intéresse également à la rigidité du schéma de détection vis-à-vis d'un ensemble de transformations d'obfuscation.

## VxStripper

VxStripper est un outil d'analyse virale. Je l'ai conçu pour la rétro-analyse de programmes protégés et potentiellement hostiles en environnement Windows. Il implémente un émulateur et plusieurs fonctions d'analyse spécialisées, permettant d'observer le programme cible et son environnement d'exécution en toute sécurité pour l'utilisateur. Il propose d'ores et déjà les fonctionnalités suivantes :
- Analyse forensique : VxStripper permet de surveiller plusieurs objets et structures internes critiques du système d'exploitation, à plusieurs points de l'exécution du programme cible. Ce contrôle de l'environnement permet de mesurer les modifications subtiles induites par le programme cible sur son environnement d'exécution. Cette fonctionnalité est particulièrement adaptée à l'analyse de programmes qui interagissent à bas niveau avec le système d'exploitation, comme les virus furtifs (rootkits), ou induisent une corruption du système à bas niveau (virus de *boot*, par exemple).
- Unpacking : VxStripper permet de supprimer presque automatiquement le loader de protection de nombreux packers du marché. Cette fonctionnalité est un prérequis à l'analyse de code viral, dans la mesure où de nombreux virus sont protégés par ce type de solution.
- API Hooking : VxStripper permet de tracer les interactions du programme cible avec l'ensemble des API du système d'exploitation, en incluant les API natives du système invité. Cette fonctionnalité permet à l'utilisateur d'instrumenter sélectivement le code des fonctions d'API, de manière difficilement détectable pour le programme en cours d'analyse.

Les fonctionnalités suivantes sont encore en cours de développement :
- Désobfuscation : VxSripper devrait implémenter plusieurs transformations d'optimisation, permettant de supprimer tout ou partie des transformations implémentées par les moteurs d'obfuscation virale. Couplée avec

une analyse différentielle statique de binaire [DR05, SAB07a, SAB07b], cette fonctionnalité devrait permettre à l'utilisateur d'identifier plus facilement la famille d'appartenance d'un programme viral polymorphe, ou généré automatiquement à partir d'un kit VGK (*Virus Generator Kit*).
- Analyse interactive : VxStripper partage avec un débogueur certaines fonctionnalités, cruciales lors de l'analyse dynamique d'un programme. Lors de l'analyse d'un programme, l'utilisateur a la possibilité de définir un point d'arrêt virtuel, conditionnant l'exécution d'une ou plusieurs des fonctions d'analyse décrite ci-dessus. Même si son interface utilisateur n'est pas encore aussi ergonomique que celle d'un outil d'analyse comme IDA Pro, VxStripper offre d'hors et déjà un bon niveau d'interactivité, par le biais d'une interface en ligne de commande (shell). Une fonction de visualisation du graphe PCG (Program Control Graph) devrait permettre de visualiser le graphe du programme cible, avant et après application des transformations de désobfuscation.

VxStripper n'est pas un produit, dans la mesure où ses spécifications ne sont pas encore figées. Il doit plus être considéré comme un prototype. Il me permet de valider par l'expérimentation certaines des idées avancées dans cette thèse. Les algorithmes à la base de son fonctionnement ont étés présentés aux conférences AVAR[2] 2006 et EICAR[3] 2007. VxStripper est utilisé comme outil dans notre laboratoire pour les activités suivantes :
- Analyse de code viral ;
- Maintient d'une base virale, constituée exclusivement de virus non packés par utilisation d'un des outils spécialisés du marché[4] . VxStripper nous permet également de statuer rapidement sur la non corruption d'un exécutable (celui-ci s'exécute correctement et ne s'arrête pas sur une erreur du système). Cet outil nous permet également de vérifier que la description d'un virus (identifié sur une des bases d'informations maintenues par les éditeurs de produits antivirus) correspond aux interactions observées du virus avec son environnement d'exécution (notamment les appels aux fonctions d'API) ;
- Étude des protections logicielles implémentées par certaines applications, notamment concernant les fonctions d'ancrage au système d'exploitation et les mécanismes de furtivité ;
- Étude des produits antivirus (et des HIPS), particulièrement concernant la manière dont ils instrumentent les API du système d'exploitation. Cet outil est en particulier utilisé dans les tests présentés dans l'annexe A. Il permet de récupérer une information fiable sur les caractéristiques de l'installation d'un produit antivirus, (concernant en particulier les pilotes de périphériques et le registre du système d'exploitation), sur la manière dont se fait la mise à jour du produit (instrumentation des fonctions d'API réseau). Il permet de caractériser les mécanismes de protection éventuellement implémentés par le produit antivirus (ancrage, mécanismes de furtivité). Il permet l'utilisation de méthodes forensiques pour cacher un code viral dans les secteurs défectueux du système de fichier, ou pour l'écrire directement en mémoire vive. Il est également parfois utilisé pour la mise au point des programmes de test, notamment les programmes furtifs.

Je pense que VxStripper pourrait être utile également à l'élaboration de schéma de détection, fondés par exemple sur un estimateur de la distribution $DSys$, par utilisation de méthodes forensiques ou furtives d'extraction d'information, donc sans interaction avec le système (voir section 6.3.11).

## Résultats

Nous démontrons la pertinence de la cryptographie boîte blanche dans le contexte viral. Nous définissons en outre des axes d'amélioration des implémentations boîte blanche de l'AES et du DES. Cette étude complète le panorama de la menace virale, concernant en particulier les virus spécialisés dans l'extorsion [YY96, YY04, Gaz08]. Une partie de cette étude a été publiée dans les actes de la conférence EICAR 2008 [Jos08].

Nous avons étudié les aspects techniques et théoriques liés à la protection et à la rétro-ingénierie logicielle, au regard de la théorie de la complexité et de l'optimisation des programmes. Nous avons complété le panorama de la menace virale par l'étude et l'évaluation des mécanismes non cryptographiques d'un produit de protection logicielle représentatif de l'état de l'art dans ce domaine. Une partie de cette étude, réalisée en collaboration avec G. Dabosville, a été publiée dans les actes de la conférence C&ESAR 2007 [DJ07]. Elle a également fait l'objet d'une publication dans le journal MISC [Jos05].

[2] AVAR, *Association of anti-Virus Asia Researchers*, http ://www.aavar.org/
[3] EICAR, *European Institute for for Computer Antivirus Research*, http ://www.eicar.org/
[4] Remarquons que l'analyse reste en partie manuelle, même si elle est considérablement simplifiée par l'utilisation de cet outil.

Nous proposons une redéfinition des notions d'obfuscation et de désobfuscation dans le cadre formel de l'interprétation abstraite. Cet outil formel nous permet à la fois de qualifier le processus d'extraction d'informations sémantiques et de hiérarchiser ces abstractions au regard d'une classe de mécanismes d'obfuscation. Sur cette base formelle, nous sommes en mesure de qualifier en terme de correction et de complétude les transformations de désobfuscation appliquées à un programme lors d'une analyse manuelle ou lors du processus d'extraction d'un schéma de signature.

Nous avons défini un ensemble de techniques de récupération d'information d'intégrité et de cohérence de certaines parties du système d'exploitation par utilisation d'un émulateur de la plate-forme matérielle supportant le système d'exploitation. L'outil VxStripper, qui implémente ces heuristiques et techniques forensiques, permet de surveiller les composants critiques du système d'exploitation et de récupérer automatiquement de l'information sur un virus furtif ou tout autre programme interagissant à un bas niveau avec l'exécutif du système d'exploitation.
Le processus de récupération d'information est fiable et sécurisé. Cette information est précieuse pour l'élaboration de schéma de détection caractérisant ce type d'infection informatique. Une partie de cette étude a été publiée dans les actes de la conférence EICAR 2007 [Jos07a].

Nous proposons un nouvel algorithme de déchiffrement par émulation pour déprotéger presque automatiquement un virus packé par utilisation d'un des outils spécialisés du marché. Cet algorithme est implémenté dans l'outil VxStripper. Une partie de cette étude a été publiée dans les actes de la conférence AVAR 2006 [Jos06b].

Nous proposons une modélisation statistique du problème de la détection virale. Cette vision complémentaire s'applique à l'analyse de comportements viraux sur la base d'informations statistique. Elle permet notamment de capturer certains aspects liés aux interactions du programme avec son environnement ou à la distribution du spectre des instructions.
Cet outil formel nous permet de qualifier en boîte blanche les schémas de détection utilisés par les moteurs de détection probabilistes. Une partie de cette étude, réalisée en collaboration avec E. Filiol, a été publiée dans les actes de la conférence EICAR 2007 [FJ07].

Nous proposons enfin une approche méthodologique et une plate-forme de tests pour l'analyse de robustesse des fonctions et mécanismes d'un produit anti-virus. Nous avons développé plusieurs outils complémentaires et prototypé plusieurs des composants d'un produit anti-virus de manière à pouvoir implémenter et étalonner convenablement les tests de robustesse en boîte noire. Une partie de cette étude a été publiée dans les actes de la conférence EICAR 2006 [Jos06a]. Elle a également fait l'objet de deux publications dans le journal MISC [FEG+07, FGJ+08], réalisées en collaboration avec G. Geffard et en association avec le Laboratoire de Virologie et Cryptologie (LVC) de l'Ecole Supérieure et d'Application des Transmissions (ESAT).
L'outil VxStripper [Jos07b] est un outil indispensable pour la réalisation de plusieurs des tests. Il permet notamment de maintenir une base virale exploitable dans le cadre des tests en boîte noire.
Les outils formels sont utilisés pour qualifier en boîte blanche - ou sur la base des information de spécification et de conception - la conformité et l'efficacité d'un moteur de détection.

## Organisation de la thèse

Avant de présenter au chapitre 8 le cadre méthodologique et le banc de tests que nous avons mis en oeuvre pour tester les produits antivirus, nous présentons les autres travaux effectués pour mieux comprendre les différentes étapes qui jalonnent la prise en compte d'un virus par un produit antivirus, de l'analyse manuelle du code viral par l'expert à l'algorithme de détection :

- Le chapitre 1 donne un apperçu des fondamentaux théorique de la virologie informatique et introduit le problème de la détection virale et les principaux problèmes ouverts au regard de la théorie de la complexité.
- Le chapitre 2 présente l'utilisation malicieuse qui peut être faite des mécanismes cryptographiques WBAC et des nouveaux défis pour l'analyse de code malicieux implémentant ces mécanismes.
- Le chapitre 3 introduit les aspects techniques et théoriques liés à la protection et à la rétro-ingénierie logicielle, au regard de la théorie de la complexité et de l'optimisation des programmes. Il complète le panorama de la menace virale établi au chapitre 2 par l'étude et l'évaluation des mécanismes non cryptographiques d'un

produit de protection logicielle représentatif de l'état de l'art dans ce domaine.
- Le chapitre 4 formalise la rétro-ingénierie comme approximation discrète des sémantiques d'un programme. Ce modèle va nous être précieux pour qualifier l'information sémantique extraite lors du processus d'analyse d'un programme viral.
- Le chapitre 5 décrit un algorithme permettant de récupérer le corps d'un virus chiffré et de reconstruire l'exécutable.
- Le chapitre 6 est dédié à la récupération d'information d'interaction bas niveau avec l'OS indispensable à l'analyse des programmes furtifs.
- Le chapitre 7 focalise sur les algorithmes de détection probabilistes, dont la décision se fonde sur la base d'informations statistiques.
- Le chapitre 8 présente une approche méthodologique, une plate-forme de tests et une revue des critères permettant de qualifier un algorithme de détection et plus globalement d'évaluer la robustesse des fonctions de sécurité d'un produit antivirus.

# Première partie

# Analyse de la menace virale

Chapitre 1

# Fondements théoriques de la virologie informatique

Ce chapitre donne un aperçu des fondamentaux théoriques de la virologie informatique et introduit le problème de la détection virale et les principaux problèmes ouverts de cette discipline au regard de la théorie de la complexité. Un état de l'art plus complet est fourni dans [Fil04], dont une partie de ce chapitre est issue. Plusieurs problèmes ouverts ont été identifiés dans ce domaine [FHZ06]. Je positionne mes travaux au sein des axes de recherche identifiés de cette discipline [FHZ06].

## 1.1 Introduction

Historiquement, les formalismes proposés par F. Cohen [Coh86] et L. M. Adleman [Adl90] sont les premières modélisations explicites de ce type de programme. Leurs résultats sont des fondamentaux de la virologie informatique et permettent notamment de statuer sur la complexité en pire cas du problème général de la détection virale.
Nous avons choisi de présenter également des travaux plus récents effectués dans ce domaine de l'informatique théorique. Ceux-ci permettent d'expliquer ou de compléter les modèles initiaux en les raffinant au regard d'une menace virale en constante évolution.

Ce chapitre permet à la fois d'introduire des concepts que nous supposerons connus dans le reste de la thèse et de positionner nos travaux au sein des axes de recherche identifiés de cette discipline.

Le reste de ce chapitre est organisé de la manière suivante :
- la section 1.2 présente les différents formalismes qui s'appliquent à la définition de virus ;
- la section 1.3 présente plusieurs formalisations, au regard de la théorie de la complexité, du problème de la détection virale ;
- la section 1.4 conclue sur les conséquences pratiques de ce résultat et sur le positionnement de mes travaux au sein des axes de recherche identifiés de cette discipline.

Les principaux concepts et résultats théoriques en calculabilité, décidabilité et complexité utilisés dans ce chapitre sont présentés en annexe E.

## 1.2 Définition formelle d'un virus informatique

Fred Cohen et Leonard Adleman apparaissent comme les pères fondateurs de la virologie informatique, dans la mesure où ils ont produit les premiers résultats fondamentaux de cette discipline. Nous rappelons les définitions théoriques qu'ils ont proposées, ainsi que les problèmes intéressants soulevés par leurs travaux, qui font aujourd'hui encore l'objet de travaux de recherche. Les modèles théoriques qu'ils ont proposés tendent à devenir mal adaptés à la

formalisation de la menace virale actuelle et à ses rapides évolutions. Nous rappelons donc également les principales définitions théoriques de la notion de virus informatique (ou de code malicieux) qui ont été proposées par la suite, et qui permettent d'expliquer ou de compléter les modèles initiaux en les raffinant au regard d'une menace virale en constante évolution.

### 1.2.1 Principaux formalismes

#### Formalisme de Cohen

La première définition abstraite de la notion de virus informatique fut établie par Cohen en 1986 [Coh86]. Son approche sur fonde sur les machines de Turing. F. Cohen définit les notions de virus et d'ensemble viral $(M, V) \in \mathcal{V}$ de la manière suivante :

**Définition 1.2.1** *(Virus, Cohen). Soit $M$ une machine de Turing et $V$ un ensemble non vide de programmes. Un virus est défini par la propriété suivante : si $v$ est sur la bande de $M$ à l'instant $t$, alors il existe un instant $t'$ et un programme $v' \in V$ tel que $v'$ est sur la bande de $M$ à l'instant $t'$.*

**Définition 1.2.2** *(Ensemble viral). Considérons la paire $(M, V)$, où $M$ est une machine de Turing et $V$ est un ensemble non vide de programmes. $(M, V)$ est un ensemble viral (i.e. $(M, V) \in \mathcal{V}$) si et seulement si pour tout virus $v \in V$, pour tous les historiques de la machine $M$, on a : $\forall t \in \mathbb{N}$ et pour toute cellule $j$ de la machine $M$, si*
- *la tête de lecture pointe sur la cellule $j$ à l'instant $t$ et*
- *$M$ est dans son état initial à l'instant $t$ et*
- *les cellules de la bande à partir de la cellule $j$ contiennent le virus $v$,*

*alors, il existe un virus $v' \in V$, à l'instant $t' > t$ et à l'index $j'$ tel que :*
- *l'index $j'$ est suffisamment loin de la position de $v$ (dont la première instruction est à l'index $j$),*
- *les cellules de la bande en partant de l'index $j'$ contiennent le virus $v'$ et*
- *à un instant $t''$ tel que $t < t'' < t'$, $v'$ est écrit par $M$.*

*De manière abrégée, on peut écrire que $V$ est un ensemble viral relativement à la machine $M$, si et seulement si*

$$[(M, V \in \mathcal{V}]$$

*et que $v$ est un virus relativement à la machine $M$ si et seulement si*

$$[v \in V] \text{ tel que } [(M, V \in \mathcal{V}].$$

#### Formalisme d'Adleman

L'approche adoptée par Adleman [Adl90] est un peu différente. Elle se fonde sur les fonctions récursives et définit un virus comme une fonction récursive totale $v$ qui s'applique à tout programme $z$ (numérotation de Gödel des programmes) de manière à rendre compte des fonctionnalités virales caractéristiques que sont l'imitation fonctionnelle du programme hôte, le déclenchement d'une charge finale ou l'infection d'un autre programme.

Considérons que le système est constitué de $n$ données $x = (x_1, \ldots, x_n)$ et de $m$ programmes $y = (y_1, \ldots, y_m)$. L'environnement système est donc défini par le couple $(x, y)$.
Si $\varphi_z(x, y) = (x', y'_1, \ldots, y'_m)$ correspond à l'exécution du programme $z$ sur le système, lors de l'infection d'autres programmes du système par le programme infecté $v(z)$, on note : $v(\varphi_z(x, y)) = (x', \epsilon_v(y'_1), \ldots, \epsilon_v(y'_m))$, où $\epsilon_v$ est une fonction de sélection calculable définie par : $\epsilon_v(y) = y \vee v(y)$.
Adleman définit un virus comme suit :

**Définition 1.2.3** *(Virus, Adleman [Adl90]). Un fonction calculable totale $v$ est un virus (au sens d'Adleman) si $\forall (x, y)$, elle vérifie l'une des propriétés suivante :*

1. $\varphi_{v(z)}(x, y) = \varphi_{v(z')}(x, y))$
2. $\varphi_{v(z)} = v(\varphi_z(x, y))$

3. $\varphi_{v(z)}(x,y) = \varphi_z(x,y)$

La propriété $\varphi_{v(z)}(x,y) = \varphi_{v(z')}(x,y)$ (*injure*) traduit le fait que l'exécution du code viral $v(z)$ est faite indépendemment du programme infecté ($z$ ou $z'$) : le programme infecté ignore la tâche initialement conçue pour le programme hôte et exécute du code viral indépendant.

La propriété $\varphi_{v(z)} = v(\varphi_z(x,y))$ (*infect*) correspond au cas de l'infection. Tout programme est potentiellement réécrit relativement à $v$ ; les données sont laissées inchangées.

La propriété $\varphi_{v(z)}(x,y) = \varphi_z(x,y)$ (*imitate*) traduit le fait que l'exécution du programme infecté $v(z)$ correspond exactement à l'exécution du programme sain $z$. Le programme infecté imite donc parfaitement le programme d'origine, à des fins de furtivité.

**Remarques** :

1. La propriété $\varphi_{v(z)} = v(\varphi_z(x,y))$ suppose que tout programme peut être potentiellement modifié par l'action de $v$ mais que les données ne sont pas altérées.
2. Cette propriété s'exprime de manière équivalent de la manière suivante (voir definition 1.2.5) : $\varphi_{v(z)}(x,y) = \varphi_z(x, y[v(\underline{S(y)})])$, où la notation $y[v(\underline{S(y)})]$ représente le remplacement du programme sélectionné $y' = S(y)$ par sa forme infectée $v(y') = v(S(y))$ dans le système.
3. En revanche, la propriété $\varphi_{v(z)}(x) = v(\varphi_z(x))$ (définition 1.2.10) n'est pas équivalente. Elle suppose que toute partie de $\varphi_z(x)$ (code et données) peut être ré-écrite par $v$.

En 20 ans, le nombre de variétés de virus a considérablement augmenté. Parmi celles-ci, nous pouvons citer notamment les virus métamorphes et les virus furtifs. Même si la notion de plus grand ensemble viral inclue celle de métamorphisme, un formalisme adapté à la description de ces nouveaux virus (pouvant être vu comme un raffinement des modèles de Cohen et Adleman) était nécessaire. Des travaux plus récents se proposent de revisiter ces modèles, et de décrire certaines familles de virus, comme les virus métamorphes, les virus furtifs, etc.
Les principaux modèles utilisés sont les machines de Turing, les machines à registre et les fonctions récursives. Dans tous les cas, le théorème $s_n^m$, le théorème universel et le théorème de récursion [RJ87] sont des outils dans l'élaboration de nouveaux modèles ou le raffinement des modèles existants.

### Formalisme de Zhou et Zuo

Zhou et Zuo [ZZZ05, ZZ04] ont généralisé le modèle d'Adleman, de manière à rendre compte de certaines propriétés des virus, telles que la furtivité (qui était déjà prise en compte partiellement dans la propriété d'imitation de la définition d'Adleman) ou le métamorphisme.

Considérons deux prédicats récursifs $\mathcal{T}(x,y)$ et $\mathcal{I}(x,y)$ disjoints : $\nexists (x,y) | \mathcal{T}(x,y) \wedge \mathcal{I}(x,y)$. On suppose de plus que l'ensemble $\{(x,y) | \neg(\mathcal{T}(x,y) \wedge \mathcal{I}(x,y))\}$ est infini.

1. Le prédicat $\mathcal{T}(x,y)$ est la condition de déclenchement d'une charge finale $D(x,y)$, fonction récursive virale indépendante du programme infecté $v(z)$ :
$$\text{si } \mathcal{T}(x,y),\ \varphi_{v(z)}(x,y) = D(x,y).$$
2. Le prédicat $\mathcal{I}(x,y)$ est la condition d'infection. S'il est vérifié, le virus utilise une fonction récursive de sélection $S(y)$ pour choisir un programme $z'$, l'infecte puis exécute le programme original $z$ :
$$\text{si } \mathcal{I}(x,y),\ \varphi_{v(z)}(x,y) = \varphi_z(x, y[v(\underline{S(y)})]).$$

### Noyau d'un virus

On définit le noyau d'un virus de la manière suivante :

**Définition 1.2.4** *(Noyau d'un virus, Zhou & Zuo [ZZ04]) Le noyau d'un virus $v$ est un quadruplet $(\mathcal{T}(x,y), \mathcal{I}(x,y), S(y), D(x,y))$ où $\mathcal{T}(x,y)$ est un prédicat récursif, appelé condition de déclenchement d'une charge finale, $\mathcal{I}(x,y)$ est un prédicat récursif appelé condition d'infection, $S(y)$ est une fonction récursive appelée fonction de sélection et $D(x,y)$ est une fonction récursive appelée charge finale.*

### Virus non résident

Les deux prédicats $\mathcal{T}(x,y)$, $\mathcal{I}(x,y)$ et les fonctions $S(y)$ et $D(x,y)$ permettent de définir un virus non résident de manière unique.

**Définition 1.2.5** *(Virus non résident, Zhou & Zuo [ZZ04]). Soient $\mathcal{T}$ et $\mathcal{I}$ deux ensembles calculables disjoints. Une fonction calculable $v$ est un virus non résident si $\forall z$ :*

$$\varphi_{v(z)}(x,y) = \begin{cases} D(x,y) & \textit{si } \mathcal{T}(x,y) \\ \varphi_z(x, y[v(\underline{S(y)})]) & \textit{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \textit{sinon.} \end{cases}$$

### Virus résident

**Définition 1.2.6** *(Virus résident, Zhou & Zuo [ZZ04]). Soient $\mathcal{T}$ et $\mathcal{I}$ deux ensembles calculables disjoints. La paire de fonctions calculables (récursives totales) $(v,s)$ est un virus résident si $\forall z$ :*

$$\varphi_{v(z)}(x,y) = \begin{cases} D(x,y) & \textit{si } \mathcal{T}(x,y) \\ \varphi_z(y[v(\underline{S(s)})]) & \textit{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \textit{sinon.} \end{cases}$$

*et*

$$\varphi_{v(s)}(x,y) = \begin{cases} D'(x,y) & \textit{si } \mathcal{T}'(x,y) \\ \varphi_s(y[v(\underline{S(y)})]) & \textit{si } \mathcal{I}'(x,y) \\ \varphi_s(x,y) & \textit{sinon.} \end{cases}$$

### Virus polymorphe

**Définition 1.2.7** *(Virus polymorphe avec deux formes, Zhou & Zuo [ZZ04]). Soient $\mathcal{T}$ et $\mathcal{I}$ deux ensembles calculables disjoints. La paire de fonctions calculables (récursives totales) $(v,v')$ est un virus polymorphe avec deux formes si $\forall z$ :*

$$\varphi_{v(z)}(x,y) = \begin{cases} D(x,y) & \textit{si } \mathcal{T}(x,y) \\ \varphi_z(y[v'(\underline{S(y)})]) & \textit{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \textit{sinon.} \end{cases}$$

*et*

$$\varphi_{v'(z)}(x,y) = \begin{cases} D(x,y) & \textit{si } \mathcal{T}(x,y) \\ \varphi_z(y[v(\underline{S(y)})]) & \textit{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \textit{sinon.} \end{cases}$$

**Définition 1.2.8** *(Virus polymorphe avec une infinité de formes, Zhou & Zuo [ZZ04]). Soient $\mathcal{T}$ et $\mathcal{I}$ deux ensembles calculables disjoints. Une fonction calculable $v$ est un virus polymorphe avec une infinité de formes si $\forall n, z$ :*

$$\varphi_{v(n,z)}(x,y) = \begin{cases} D(x,y) & \textit{si } \mathcal{T}(x,y) \\ \varphi_z(n+1, y[v(\underline{S(y)})]) & \textit{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \textit{sinon.} \end{cases}$$

*et*

$$\forall n, n', \quad \textit{si } n \neq n' \textit{ alors } v(n,z) \neq v(n',z).$$

**Théorème 1.2.1** *(Existence des virus polymorphe avec une infinité de formes, Zhou & Zuo [ZZ04]). Il existe des virus polymorphes avec une infinité de formes.*

**Preuve** : considérons la fonction récursive totale $g(n, z, z')$ telle que :

$$\varphi_{g(n,z,z')}(x,y) = \begin{cases} < n, y > & \text{si } \mathcal{T}(x,y) \\ \varphi_z(x, y[\varphi_{z'}(n+1, \underline{S(y)})]) & \text{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \text{sinon.} \end{cases}$$

D'après le théorème de récursion, il existe un programme $z'_0$ telle que : $g(n, z, z') = \varphi_{z'_0}(n, z)$. Posons : $v(n, z) = g(n, z, z'_0) = \varphi_{z'_0}(n, z)$.

$$\varphi_{v(n,z)} = \varphi_{g(n,z,z'_0)}(x,y) = \begin{cases} < n, y > & \text{si } \mathcal{T}(x,y) \\ \varphi_z(x, y[v(n+1, \underline{S(y)})]) & \text{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \text{sinon.} \end{cases}$$

si $\mathcal{T}(x, y)$, $n \neq n'$ alors $\varphi_{v(n,z)}(x, y) =< n, y > \neq < n', y >= \varphi_{v(n',z)}(x, y)$.
□

Ces définitions complètent celles données par Adleman (définition 1.2.3) et permettent de rendre compte de la notion de mutation. En effet, étant donné un programme $z$, l'ensemble des mutation du programme infecté est donné par $\{v(n, z)\}_n$, l'entier $n$ permettant d'indexer les générations de virus.

Zhou & Zuo ont également formalisé la notion de virus furtif [ZZ04] :

### Virus furtif

**Définition 1.2.9** *(Virus furtif, Zhou & Zuo [ZZ04]). Soient $\mathcal{T}$ et $\mathcal{I}$ deux ensembles calculables disjoints. La paire de fonctions calculables (récursives totales) $(v, s)$ est un virus furtif s'il existe une fonction récursive $h$ telle que $\forall z$ :*

$$\varphi_{v(z)}(x,y) = \begin{cases} D(x,y) & \textit{si } \mathcal{T}(x,y) \\ \varphi_z(x, y[v(\underline{S(y)}), h(\underline{s})]) & \textit{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \textit{sinon.} \end{cases}$$

*et*

$$\varphi_{h(s)}(x) = \begin{cases} \varphi_s(y) & \textit{si } x = v(y) \\ \varphi_s(x) & \textit{sinon.} \end{cases}$$

La notion de furtivité a ensuite été revisitée sous l'angle statistique par E. Filiol [Fil07a]. En comparant les techniques de furtivité aux techniques stéganographiques, E. Filiol propose une formalisation du problème de la détection des virus furtifs dans le cadre de la théorie de l'information. Nous présentons ce formalisme de manière plus détaillée au chapitre 6 (section 6.3.11).

### Formalisme de Bonfante, Kaczmarek & Marion

Plus récemment, Bonfante, Kaczmarek & Marion ont proposé un formalisme [BKM05, BKM06] qui est proche de ceux proposés par Adleman et Zhou & Zuo, et permet d'unifier les formalismes appliqués jusqu'à présent. Ils ont choisi de décrire un virus comme le point fixe d'une fonction semi-calculable. À travers la définition de fonction de duplication, ils proposent en outre une approche constructive permettant de formaliser de nouveaux types de virus.

**Définition 1.2.10** *(Virus, Bonfante, Kaczmarek & Marion [BKM05]). Soit $b$ une fonction semi-calculable, appelée fonction de propagation. Un programme $v$ est un virus relativement à la fonction de propagation $b$ si $\forall z, x$ :*

$$\varphi_v(z,x) = \varphi_{b(v,z)}(x).$$

Soit $f(y,x)$ une fonction calculable. Considérons la fonction $g(y,z,x) = f(y,z)$. La fonction $g(y,z,x)$ est calculable. D'après le théorème du point fixe, il existe un programme $v$ tel que pour tout $x$, $\varphi_v(z,x) = g(v,z,x) = f(v,z)$. Soit $e$ un programme calculant $g(y,z,x)$. On a :

$$g(y,z,x) = \varphi_e(y,z,x).$$

D'après le théorème $s_n^m$, il existe une fonction élémentaire $s(e,y,z)$ telle que :

$$\varphi_e(y,z,x) = \varphi_{s(e,y,z)}(x).$$

On a donc : $\varphi_v(z,x) = f(v,z) = g(v,z,x) = \varphi_e(v,z,x) = \varphi_{s(e,v,z)}(x)$. On a donc montré l'existence d'un programme $v$ et d'une fonction élémentaire $b(v,z) = s(e,v,z)$ tels que :

$$\varphi_v(z,x) = \varphi_{b(v,z)}(x).$$

On en déduit, compte tenu de la définition 1.2.10, le résultat d'existence suivant :

**Théorème 1.2.2** *(Existence d'un Virus, Bonfante, Kaczmarek & Marion [BKM05]) Soit $f(y,x)$ une fonction calculable, il existe un virus $v$ tel que $f(v,z) = \varphi_v(z,x)$.*

#### Fonction de duplication

La fonction calculable $f(y,x)$ peut être définie de manière à ce que $f(y,x)$ contienne au moins une occurence de $y$. Cette propriété assure la propriété d'auto-reproduction du virus construit à partir de $f(y,x)$.

**Définition 1.2.11** *Une fonction de duplication $\Delta$ est une fonction calculable telle que $\forall v, z$, $\Delta(v,z)$ contienne au moins une occurence de $v$. Un duplicateur est un programme viral $v$ tel que $\varphi_v(z,x) = \Delta(v,z)$.*

Le théorème 1.2.2 nous assure donc de l'existence d'un virus de type duplicateur dès lors que l'on est capable d'établir l'existence d'une fonction de duplication. Bonfante & al. proposent donc la construction de quelques fonctions de duplication, représentatives de ce qui peut se rencontrer dans la nature :

- La fonction de duplication $\Delta(v,z) = v$ permet de cloner le virus.
- Soit $\delta(v,z)$ une fonction de duplication telle que $\forall v, z, |\delta(v,z)| \leq |z|$ , la fonction de duplication $\Delta(v,z) = (\delta(v,z_1), \ldots, \delta(v,z_n)$ permet d'écraser la cible $z$ avec une copie du virus sans en modifier la taille.
- Considérons la fonction de duplication $\delta(v,z) = v||z$, consistant à concaténer le corps du virus au programme $z$. La fonction de duplication $\Delta(v,z) = (\delta(v,z_1), \ldots, \delta(v,z_n))$ permet d'insérer le code viral dans un programme $z$ sans en écraser le code.

### 1.2.2 Comparaison des formalismes

La définition 1.2.10 est la définition la plus générale proposée actuellement. Elle complète et recouvre les définitions proposées antérieurement. En effet, on a les résultats suivants :

**Proposition 1.2.1** *Si $v$ est un virus de Cohen (définition 1.2.1), alors $v$ est un virus de Bonfante, Kaczmarek & Marion (définition 1.2.10).*

**Proposition 1.2.2** *Si $v$ est un virus d'Adleman (définition 1.2.3), alors $v$ est un virus de Bonfante, Kaczmarek & Marion (définition 1.2.10).*

**Proposition 1.2.3** *Si $v$ est un virus de Zhou et Zuo (définition 1.2.5), alors $v$ est un virus de Bonfante, Kaczmarek & Marion (définition 1.2.10).*

Les définitions proposées par Adleman ou par Zhou et Zuo sont plus restrictives que la première définition proposée par Cohen et fondée *stricto-sensu* sur le théorème de récursion. En effet, il est possible d'exhiber un virus $v$ qui est un virus de Cohen (ou de Bonfante, Kaczmarek & Marion) et qui n'est pas un virus d'Adleman (ou de Zhou et Zuo respectivement).

Le formalisme de Bonfante, Kaczmarek & Marion est une traduction fidèle de la définition de Cohen, fondée

sur les machines de Turing, en terme de fonction récursive. En conclusion, la définition générale de la notion de virus informatique proposée par Cohen est toujours valable, même si les formalismes proposés récemment semblent plus constructifs et permettent de raffiner la formalisation du problème de la détection virale au regard de la théorie de la complexité. Nous admettrons donc la définition la plus générale de virus comme un programme qui, s'il est exécuté par une machine de Turing, se recopie plus loin sur la bande, ou de manière équivalente, comme le point fixe d'une fonction récursive partielle.

## 1.3 Le problème de la détection virale

Nous avons présenté dans la section 1.2 différents formalismes qui s'appliquent à la définition de virus. Ces modèles vont nous permettre de formaliser, au regard de la théorie de la complexité, la problématique de la détection virale. L'étude des aspects liés à la complexité des ensembles viraux est primordiale, dans la mesure où elle permet de mesurer la difficulté de la détection. La plupart des résultats concernent l'indécidabilité et la hiérarchie polynomiale (cf. définition 11).

#### Indécidabilité de l'ensemble viral de Cohen

Fred Cohen a démontré dans [Coh86] que le problème général de la détection virale est indécidable :

**Théorème 1.3.1** *(Indécidabilité de l'ensemble viral, Cohen [Coh86]). Il n'existe pas de machine de Turing $M$ capable de décider en temps fini si une séquence donnée $v$ est un virus pour une machine $M \in \mathcal{M}$.*

**Preuve** : le principe de la démonstration consiste à réduire le problème de décidabilité de l'ensemble viral au problème de l'arrêt. Étant donnée une machine $M'$ et deux séquences $v$ et $v'$, on construit une machine $M$ qui effectue les opérations suivantes :

1. copier $v'$ à partir de $v$,
2. simuler l'exécution de $M'$ sur $v'$,
3. et si le calcul de $v'$ dans $M'$ s'arrête, $v$ est dupliqué.

La machine $M$ est construite de telle sorte que $v$ se reproduit si et seulement si le calcul de $v'$ par la machine $M'$ s'arrête :
- si le calcul de $v'$ par la machine $M'$ s'arrête, alors $v$ est dupliqué,
- si le calcul de $v'$ par la machine $M'$ ne s'arrête pas,

#### Complexité de la détection des virus (au sens de Cohen)

En utilisant le formalisme de F. Cohen, D. Spinellis a prouvé dans [Spi03] que le problème général de la détection d'un virus de taille bornée est NP-complet.

#### Complexité de l'ensemble viral d'Adleman

Un peu plus tard, son directeur de thèse Leonard Adleman a donné des résultats de complexité sur des instances particulières du problème général de la détection virale :

**Théorème 1.3.2** *L'ensemble $V = \{z | \varphi_z$ est un virus (au sens d'Adleman)$\}$ est $\Pi_2$-complet.*

**Théorème 1.3.3** *L'ensemble des virus infectés par un virus $v$, défini par*

$$\{z | \exists z',\ z = v(z')\}$$

*est $\Sigma_1$-complet.*

### Complexité de l'ensemble viral de Zhou et Zuo

Zhou et Zuo ont démontré un résultat similaire avec la définition suivante :

**Théorème 1.3.4** *Un noyau étant fixé, l'ensemble*

$$\{z|\, \varphi_z \text{ est un virus (au sens de Zhou \& Zuo)}\}$$

*est* $\Pi_2$*-complet.*

**Preuve** : Démontrons ce résultat pour les virus non résidents (définition 1.2.5), le principe de la preuve s'appliquant également aux autres types de virus considérés par Zhou et Zuo (virus résident 1.2.6, virus polymorphe avec plusieurs formes 1.2.7, virus polymorphe avec une infinité de formes 1.2.8, virus furtif 1.2.9, etc.).
Un noyau étant fixé, considérons un élément $z \in \{z|\, \varphi_z$ est un virus résident$\}$. On a :

$$\varphi_v \text{ est une fonction récursive totale} \wedge [\forall z, \forall x, y :$$

$$[\mathcal{T}(x,y) \rightarrow \varphi_{\varphi_{v(z)}}(x,y) = D(x,y)] \vee$$

$$[\mathcal{I}(x,y) \rightarrow \varphi_{\varphi_{v(z)}}(x,y) = \varphi_z(x, y[\varphi_v(\underline{S(y)})])] \vee$$

$$[\neg(\mathcal{T}(x,y) \vee \mathcal{I}(x,y)) \rightarrow \varphi_{\varphi_v(z)}(x,y) = \varphi_z(x,y)]]$$

Le prédicat '$\varphi_v$ est une fonction récursive totale' est un prédicat $\Pi_2$, le prédicat '$\varphi_z(x) = \varphi_{z'}(x)$' est un prédicat $\Sigma_1$. On en déduit que le prédicat '$\varphi_z$ est un virus résident' est un prédicat $\Pi_2$.

Considérons un ensemble $\Pi_2$ : $\mathcal{A}$. Il existe donc un prédicat récursif R tel que :

$$a \in \mathcal{A} \Longleftrightarrow \forall b \exists z R(a,b,c).$$

Considérons la fonction récursive suivante :

$$f(z, z', a, \langle x, y\rangle) = \begin{cases} \varphi_z(x, y[\varphi_{z'}(\underline{S(y)})]) & \text{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \text{si } \neg\mathcal{I}(x,y) \wedge \forall b < \langle x,y\rangle \exists c\, R(a,b,c) \\ \uparrow & \text{sinon.} \end{cases}$$

Il existe donc une fonction récursive $\beta(z, z', a)$ telle que : $\varphi_{\beta(z,z',a)}(x,y) = f(z, z', a, \langle x, y\rangle)$. et il existe une fonction récursive $\nu(x)$ telle que $\varphi_{\nu(a)}(z) = \beta(z, \nu(a), a)$. On a donc :

$$\varphi_{\varphi_{\nu(a)}(z)} = \varphi_{\beta(z,\nu(a),a)} = \begin{cases} \varphi_z(x, y[\varphi_{\nu(a)}(\underline{S(y)})]) & \text{si } \mathcal{I}(x,y) \\ \varphi_z(x,y) & \text{si } \neg\mathcal{I}(x,y) \wedge \forall b < \langle x,y\rangle \exists c\, R(a,b,c) \\ \uparrow & \text{sinon.} \end{cases}$$

**Théorème 1.3.5** *L'ensemble* $\{z|\, \varphi_z$ *est un virus*$\}$ *est* $\Sigma_3$*-complet.*

La plupart des ensembles viraux définis par Zhou et Zuo [ZZZ05, ZZ04] sont démontrés $\Sigma_1$-complets, $\Pi_2$-complets ou $\Sigma_3$-complets.

### Complexité de l'ensemble viral de Bonfante, Kaczmarek & Marion

Avant de donner la complexité de l'ensemble viral défini par Bonfante, Kaczmarek & Marion, nous avons besoin du résultat suivant :

**Théorème 1.3.6** *[BKM05] Étant donnée une fonction calculable, l'ensemble des programmes z la calculant est* $\Pi_2$*-complet.*

**Preuve** : soit $f$ une fonction calculable et $E_f$ l'ensemble des programmes $z$ la calculant. Soit $z_0$ un programme calculant $f$. L'ensemble $E_f$ peut être exprimé par une formule $\Pi_2$ :

$$E_f = \{z | \forall x, \exists y : \varphi_z(x) = y \wedge \varphi_{z_0}(x) = y\}$$

Considérons la fonction $g(x, y) = f(y)$ si $\varphi_x(y) \downarrow$ et $g(x, y) = \uparrow$ sinon. Cette fonction est semi-calculable, comme restriction d'une fonction calculable à un domaine semi-calculable. Soit $z_1$ un programme calculant $g$ : $g(x, y) = \varphi_{z_1}(x, y)$. D'après le théorème $s_n^m$, il existe une fonction élémentaire $s$ telle que $\varphi_{z_1}(x, y) = \varphi_{s(z_1,x)}(y)$.
Notons $F = \{z | \varphi_z$ est calculable $\}$. Cet ensemble est $\Pi_2$-complet.

- Si $z \in F$, alors $\varphi_z(y) \downarrow$ donc $g(z, y) = f(y) = \varphi_{z_1}(z, y) = \varphi_{s(z_1,z)}(y)$. Donc $s(z_1, z) \in E_f$.
- Si $z \notin F$, alors $\varphi_z$ n'est pas totale. Or, $Dom_{\lambda y.g(z,y)} \subset Dom_{\varphi_z}$ donc $\lambda y.g(z, y)$ et $\varphi_{s(z_1,z)}$ ne sont pas totales. Puisque $f$ est totale par hypothèse, on a : $f \neq \varphi_{s(z_1,z)}$ donc $s(z_1, z) \notin E_f$.

La fonction $\lambda z.s(z_1, z)$ réduit donc l'ensemble $E_f$ à l'ensemble $F$ qui est $\Pi_2$-complet. Donc $E_f$ est $\Pi_2$-complet. □ Le théorème suivant montre que le problème de la détection virale est $\Pi_2$-complet. Notons $\mathcal{V}_b$ l'ensemble $\mathcal{V}_b = \{v | \varphi_v(z, x) = \varphi_{b(v,z)}(x)\}$.

**Théorème 1.3.7** *$\exists b$ tel que $\mathcal{V}_b$ est $\Pi_2$-complet.*

**Preuve** : Soit $f(y, x)$ une fonction calculable et $E_f$ l'ensemble des programmes $z$ la calculant. Soit $z_0$ un programme calculant $f$. On a donc $\varphi_{z_0}(y, x) = f(y, x)$. D'après le théorème $s_n^m$, il existe une fonction élémentaire $s$ telle $\varphi_{z_0}(y, x) = \varphi_{s(z_0,y)}(x)$. Donc $z_0$ est un virus relativement à la fonction de propagation $s(z_0, y)$. Donc $E_f \subseteq \mathcal{V}_s$.

Réciproquement, si $v \in \mathcal{V}_b$, D'après le théorème 1.3.6, $E_f$ est $\Pi_2$-complet. Donc $\exists b$ tel que $\mathcal{V}_b$ est $\Pi_2$-complet. □

## 1.4 Problèmes ouverts de la virologie informatique

Dans [FHZ06], les auteurs présentent les principaux problèmes de la virologie informatique, tant au niveau de la définition des modèles théoriques de virus et des contre-mesures antivirales, que des problèmes posés par la modélisation de la propagation virale, ou par les domaines de recherches techniques et pratiques.
Nous avons introduit dans les sections 1.2 et 1.3 les principales avancées récentes en virologie informatique théorique, au regard de la théorie de la complexité. On trouvera dans [FHZ06] un exposé complet des problèmes ouverts en informatique théorique. En résumé, nous pouvons observer que même si des résultats établissent sans équivoque la nature fondamentalement difficile en pire cas du problème de détection, des travaux supplémentaires sont sans aucun doute nécessaires pour aboutir à une classification applicable en pratique, de manière à identifier les classes de virus qui pourraient représenter effectivement un défi pour les politiques de protection.
Concernant la classification des modèles eux-même, et leurs relations respectives, nous avons vu dans la section 1.3 que ces modèles sont tous des modèles logiques de second ordre, et sont démontrés largement équivalents.
L'identification de nouvelles classes virales qui pourraient effectivement représenter une menace dans le futur est essentielle.
À titre d'exemple, le modèle viral correspondant aux virus polymorphes ayant une infinité de formes, introduit dans la section 1.2, ne connaît pas d'implémentation dans le monde réel. De manière générale, l'analyse de complexité des virus est faite en pire cas, et on peut se poser la question de la complexité en moyen cas de ces modèles viraux. Le problème sous-jacent, celui de savoir si un tel modèle produit bien un virus polymorphe non trivial lorsqu'il est implémenté dans un programme réel, reste un problème ouvert.
Nous rencontrons un problème équivalent avec le résultat fondamental de Fred Cohen : l'existence d'un virus informatique $v$ tel que l'ensemble de ses programmes infectés $I_v$ est indécidable (ou de manière équivalente le fait que l'ensemble $I_v$ n'est pas récursivement énumérable) ne donne pas d'information sur sa difficulté en moyen cas.

Par ailleurs, le principal inconvénient de l'approche par ensembles viraux est que les relations entre les différentes formes d'une même souche virale poly/métamorphe n'apparaissent pas clairement. L'approche fonctionnelle, par utilisation des fonctions récursives et du théorème du point fixe est sans doute plus à même de capturer une information sur la relation de dépendance entre les différentes générations du code viral, par application d'un moteur de mutation que l'on peut tenter de caractériser dans ce cadre formel. La formalisation d'un virus comme point fixe d'une fonction récursive partielle est à ce titre plus adaptée à la construction effective de nouvelles classes virales et à

l'implémentation de ces modèles. Dans ces conditions, observons que la description fonctionnelle du polymorphisme permet de voir la $i$-ième évolution d'un virus $v$ comme le résultat de l'itération $i$ fois d'une fonction récursive $f$ :

$$f^i(v, z) = \varphi_v(z, x).$$

Pour rendre compte de la menace virale plus concrètement, nous étudions dans cette thèse des formalismes alternatifs ou complémentaires, comme l'utilisation des grammaires formelles au chapitre 2, de la théorie de l'optimisation des programmes au chapitre 3, de la théorie de l'interprétation abstraite au chapitre 4 ou de la théorie des probabilités et statistiques au chapitre 7.

Cependant, l'essentiel de cette thèse se centre sur des domaines identifiés dans [FHZ06] comme étant des domaines de recherche techniques et pratiques de la virologie informatique. Outre le problème de l'évaluation d'un produit antivirus, qui est le thème central de cette thèse, nous pouvons identifier d'autres domaines qui sont en rapport plus ou moins direct avec nos travaux :

- La taxonomie des infections informatiques, dont l'objectif est de parvenir à décrire, nommer et classer ces programmes, et donc à définir une systématique de classement et à mettre en oeuvre des sondes permettant d'extraire une information discriminante de ces programmes.
- La phylogénie des infections informatiques, dont l'objet est d'étudier la formation et l'évolution de ce type de programmes, en prenant en considération le fait que ces programmes peuvent évoluer par le biais de l'itération d'un moteur de mutation ou que les programmes viraux sont rarement développés *from scratch*, mais le plus souvent dérivés de codes déjà connus.

Nous étudions aux chapitres 5 et 6 plusieurs méthodes d'extraction d'informations liées à un programme viral et à la nature de ses interactions avec son environnement système d'exploitation. Nous présentons au chapitre 3 les méthodes liées à la rétro-ingénierie logicielle, parmi lesquelles les méthodes d'analyse différentielle de binaire [DR05] fournissent des outils pratiques de recherche de similarité entre programmes, permettant de déceler d'éventuelles réutilisation de code ou de détecter l'appartenance à une souche virale poly/métamorphe.
Remarquons que l'information que nous prenons en considération dans les processus d'extraction que nous présentons n'est pas uniquement basée sur des séquences d'instructions. L'information collectée par ces sondes comprend des données de toute nature (syntaxique, sémantique, statistique). En outre, elle ne se limite pas au périmètre du programme cible, mais à son environnement d'exécution tout entier (système d'exploitation, matériel).
L'un des principaux problèmes ouverts liés à ce domaine de la recherche antivirale est sans doute la question de la possibilité d'extension de cette approche à une phylogénie fondée sur le modèle des fonctions récursives.

Parmi les domaines identifiés dans [FHZ06], nous pouvons également retenir des problématiques qui sont liées aux contre-mesures antivirales, et qui sont des thématiques abordées implicitement dans cette thèse. Citons notamment :

- Le problème de la vérification de l'intégrité du système, la question principale étant de savoir comment adapter la vérification d'intégrité de manière aussi sécurisée et fiable que possible au regard d'attaques virales.
- Les problématiques associées à la mise en quarantaine et à la virtualisation de systèmes, comme c'est la cas par exemple dans le déploiement des Honeypots. Lorsqu'un virus ou un ver s'installe ou transite en mémoire de tels systèmes, sa charge finale et le comportement de sa réplication peuvent être étudiés sans risque de contamination du système réel. Se posent donc les problèmes fondamentalement techniques de l'isolation et de la fiabilité du processus de récupération d'information.

Nous proposons au chapitre 6 un regard nouveau aux heuristiques et techniques forensiques qui sont actuellement utilisées par les détecteurs de rootkits, dont la plupart sont assimilables à des méthodes de contrôle d'intégrité ou de cohérence de l'environnement d'exploitation. Nous y présentons une méthode statistique de détection des hooks permettant de vérifier l'intégrité du système au regard d'attaques virales bas niveau, fondée sur un processus d'extraction d'information extérieur au système hôte, permettant de rendre difficile le contournement (par simulation des tests statistiques qu'elle implémente) de la fonction de détection (section 6.3.11). Le principe de simulabilité des test statistiques permet également de définir un moyen de contournement systématique des moniteurs de contrôle d'intégrité [Fil07c]. Nous présentons ce concept dans le chapitre 7.
Nous avons globalement centré notre étude sur les méthodes dynamiques d'extraction d'information fondées sur une virtualisation du matériel supportant le système d'exploitation. Nous espérons que ces travaux peuvent apporter des éléments nouveaux pour la conception de Honeypots plus efficaces ou de système de prévention proactifs plus élaborés.

## 1.5 Conclusion

Nous avons présenté dans ce chapitre d'introduction les fondamentaux théoriques de la virologie informatique et positionné nos travaux au sein des axes de recherche identifiés de cette discipline.
Ce premier chapitre donne les principaux modèles s'appliquant à la description formelle des virus, dans le cadre de la théorie de la complexité. L'objectif des deux chapitres suivants est de compléter cette analyse de la menace virale par :

- l'étude des mécanismes cryptographiques susceptibles d'être utilisés par un virus pour résister à l'analyse dynamique. C'est l'occasion d'élargir l'ensemble des problèmes ouverts de la virologie informatique en y ajoutant celui de la dissimulation d'une clé dans un algorithme, dans un contexte cryptographique particulier, appelé contexte (d'attaque) boîte blanche ;
- l'étude des mécanismes non cryptographiques susceptibles d'être utilisés dans le même but par un virus. C'est l'occasion d'élargir l'ensemble des problèmes ouverts de la virologie informatique en y ajoutant celui de la résistance effective d'un mécanisme reposant sur un problème prouvé difficile de l'optimisation de programme dans le contexte boîte blanche.

Chapitre 2

# Cryptovirologie boîte blanche

L'évaluation d'un programme antivirus doit être menée au regard d'une menace virale identifiée. Nous avons vu au chapitre 1 que l'identification de nouvelles classes virales qui pourraient effectivement représenter une menace dans le futur est essentielle. À titre d'exemple, le modèle viral correspondant aux virus polymorphes ayant une infinité de formes ne connaît pas d'implémentation dans le monde réel.
Nous étudions dans ce chapitre des mécanismes cryptographiques spécialement conçus pour permettre une grande diversification de leur implémentation. Ces mécanismes sont adaptés au contexte boîte blanche (l'attaquant est supposé avoir un contrôle total de la plate-forme d'exécution cible) et comme nous allons le démontrer, au contexte viral.
L'utilisation de mécanismes cryptographiques est apparue très tôt dans les programmes viraux, soit pour entraver leur analyse, soit pour mettre en œuvre une charge finale. Ces mécanismes se sont assez vite avérés inadaptés à protéger le code viral face à un analyste maître de la plate-forme d'exécution et disposant d'outils spécialisés. En particulier, le mode de gestion des secrets manipulés par ces mécanismes ne résiste pas à l'analyse dynamique.
La cryptovirologie est née de l'utilisation conjointe de la cryptographie symétrique et asymétrique par les virus, popularisée par la notion de cryptovirus, programmes viraux spécialisés dans l'extorsion [YY96]. La mise en œuvre d'une gestion de clés asymétrique permet de renforcer la sécurité de la charge finale, mais ne permet pas d'entraver l'analyse du code viral.

De nouveaux mécanismes cryptographiques, correspondants à une implémentation particulière de la cryptographie traditionnelle (boîte noire) ont été conçus récemment pour assurer la protection en profondeur d'applications légitimes. Ces mécanismes peuvent être détournés de leur vocation première et utilisés à des fins d'extorsion par un programme viral.
Nous évaluons ces nouvelles primitives cryptographiques et discutons de leur utilisation dans un contexte viral.

## 2.1 Introduction

Nous observons depuis plusieurs années l'utilisation par les auteurs de virus de produits de protection logicielle spécialisés, leur permettant notamment de résister plus efficacement contre la rétro-ingénierie. Le problème de la protection de contenu a suscité une recherche académique importante dans le domaine de la protection logicielle [CMH03, WDHK01, vO03, MAM+05], sous l'hypothèse que l'attaquant et l'utilisateur légitime du logiciel protégé peuvent être une seule et même personne. De nouveaux mécanismes et des suites complètes de protection logicielle permettent aux fournisseurs de contenu d'utiliser les grands moyens pour assurer la sécurité en profondeur de leurs applications [Clo07].
De la même manière que les concepteurs de virus ont utilisé certains outils spécialisés comme les packers [SRT08, Asp08] pour renforcer la protection des virus, on peut conjecturer que d'autres outils plus élaborés, comme les chaînes de compilation spécialisées, dédiées à la protection en profondeur des applications DRM, pourront être détournés de leur utilisation première pour assurer la protection en profondeur des virus.
Parmi les mécanismes mis en œuvre par ces suites de protection logicielle, certains sont de nature cryptographique.

Nous présentons dans ce chapitre une utilisation inédite d'un de ces mécanismes dans un contexte viral. Nous examinons le cas particulier de la conception d'un virus spécialisé dans l'extorsion [Gaz08]. L'utilisation de ce type de technologie est à prévoir et à prendre en compte par la communauté de la recherche anti-virale, afin d'avoir une vision plus complète de la menace virale de demain.

La conception et l'implémentation de logiciels résistants à la rétro-ingénierie est un problème crucial et difficile pour de nombreuses applications, particulièrement lorsqu'il s'agit de protéger les algorithmes propriétaires implémentés par le logiciel et/ou de protéger la fonction de contrôle de droits conditionnant l'accès à tout ou partie de ses fonctionnalités.

Lorsque l'application à protéger ne peut pas asseoir sa sécurité sur l'utilisation d'un composant matériel de sécurité, ou d'un serveur du réseau, nous devons faire l'hypothèse d'un attaquant capable d'exécuter l'application dans un environnement qu'il contrôle parfaitement. Le modèle de l'attaquant correspondant à ce contexte, que nous appellerons WBAC[1] (*White Box Attack Context*) dans la suite de ce chapitre, impose une implémentation logicielle particulière des primitives cryptographiques classiques.

Dans ce contexte, la protection d'un programme repose sur des mécanismes couvrant différents objectifs de sécurité, dont la capacité à contrôler, en plusieurs points de son exécution :

- l'intégrité de son code, de ses données critiques et de son contexte d'exécution ;
- la confidentialité des algorithmes propriétaires ;
- la diversification de ses instances ;
- l'ancrage du logiciel à une plate-forme cible d'exécution personnalisée, etc.

**Contexte viral** Nous pouvons observer que les infections informatiques ne doivent pas seulement être résistantes à la rétro-ingénierie, mais doivent également éviter la détection. Dans le reste de ce chapitre, nous nous intéressons à l'utilisation des mécanismes cryptographiques dans le contexte WBAC avec cette contrainte supplémentaire.

Nous nous intéressons en premier lieu aux contraintes liées à la dissimulation des clés dans le contexte viral, dans le but d'interdire ou d'entraver l'analyse de code viral, ce sans utiliser la cryptographie asymétrique, sans utiliser l'environnement, de manière à assurer une autonomie suffisante (i.e. ne pas être dirigé), tout en échappant à la détection.

Nous allons voir que le problème sous-jacent est ouvert : il correspond au problème de la conception d'un algorithme de chiffrement à clé privée vérifiant la propriété de la boîte noire virtuelle, présentée dans la section 2.2. Ce problème n'a pas de solution algorithmique connue, et il est même démontré qu'il n'existe pas de moyen générique d'y parvenir. Cependant, des résultats positifs, comme celui de l'obfuscation de la fonction point dans le modèle de l'oracle aléatoire laisse penser qu'il y a peut-être un espoir d'y parvenir, en dépit des problèmes liés à l'implémentation de ce modèle.

C'est dans ce cadre théorique que nous étudions les tentatives d'implémentation boîte blanche de l'AES et du DES. Notre principale contribution sur ce sujet sont la définition d'axes d'amélioration des implémentations boîte blanche de l'AES et du DES, et la démonstration de la pertinence de la cryptographie boîte blanche dans le contexte viral. Une partie du contenu de ce chapitre a été publiée dans les actes de la conférence EICAR 2008 [Jos08].

Le reste du chapitre est organisé de la manière suivante : la section 2.2 rappelle les résultats théoriques fondamentaux concernant l'obfuscation, fondés sur la propriété de la boîte noire virtuelle. La section 2.3 présente la problématique liée à l'utilisation de la cryptographie par un virus à des fins d'extorsion et la section 2.4 donne des exemples de l'utilisation (illégitime) de la cryptographie boîte blanche. Nous y présentons également l'expression de besoin d'un mécanisme cryptographique adapté au contexte WBAC. La section 2.5 présente les principes de l'implémentation boîte blanche de deux algorithmes : le DES et l'AES. La section 2.6 discute de la robustesse de ces algorithmes vis-à-vis de la cryptanalyse. La section 2.7 conclut sur l'utilisation qui pourrait être faite de ce type de technologie par les cryptovirus de demain, et sur les contre-mesures et limitations de cette technologie.

[1] Notons que le contexte WBAC correspond au contexte le plus contraignant pour le concepteur de protection logicielle, dans la mesure ou un mécanisme résistant en boîte blanche doit également être résistant aux attaques boîte noire (BBAC, Black Box Attack Context) et boîte grise (GBAC, Gray Box Attack Context). Le contexte BBAC est le plus classique en cryptographie : l'attaquant n'a pas accès aux informations liées à l'implémentation. Le contexte GBAC correspond aux attaques logiques exploitant l'information qui fuit du matériel (consommation de courant, temps d'exécution des instructions ou de certaines opérations du CPU, comme la mise en cache, rayonnement électromagnétique, spectre sonore, etc.). L'attaquant n'a accès qu'à des informations partielles relatives à l'implémentation. Celles-ci sont obtenues par modélisation d'un phénomène physique.

## 2.2 Cadre théorique

Nous nous focalisons dans ce chapitre sur les mécanismes cryptographiques spécialement conçus pour assurer la confidentialité d'une clé secrète au sein d'un algorithme. Une telle transformation (dissimulation d'une clé dans un algorithme de chiffrement, avec ou sans l'aide d'une interaction environnementale) peut être formalisée comme une transformation d'obfuscation. Nous rappelons dans cette section des résultats négatifs et positifs concernant l'obfuscation de code, ainsi que leur impact sur ce problème de gestion de clé.

### 2.2.1 L'obfuscateur idéal

Soit $\Pi$ un ensemble de programmes, et $PPT$ l'ensemble des machines de Turing probabilistes dont le temps de calcul est polynomial relativement à la taille des entrées. Un obfuscateur peut être défini formellement [BGI+01] comme suit :

**Définition 2.2.1** *Soit $\Pi$ un ensemble de programmes, et $PPT$ l'ensemble des machines de Turing probabilistes dont le temps de calcul est polynomial relativement à la taille des entrées. Un algorithme probabiliste $\mathcal{O}$ est un obfuscateur s'il satisfait les propriétés suivantes :*

1. *$\forall P \in \Pi$, $P$ et $\mathcal{O}(P)$ calculent la même fonction.*
2. *$\forall P \in \Pi$, l'accroissement du temps d'exécution et de la taille de $\mathcal{O}(P)$ est au plus polynomial relativement au temps d'exécution et à la taille du programme $P$.*
3. *$\forall A \in PPT$, $\exists S \in PPT$ tel que*

$$\forall P \in \Pi,\ p[A(\mathcal{O}(P) = 1] \backsimeq p[S^P(1^{|P|}) = 1].$$

L'égalité $p[A(\mathcal{O}(P) = 1] = p[S^P(1^{|P|}) = 1]$ est vraie à une fonction négligeable[2] $\mu$ de la taille du programme $|P|$ près. La dernière propriété, appelée propriété de la boîte noire virtuelle, peut donc aussi s'écrire : $\forall A \in PPT$, $\exists S \in PPT$ et $\exists$ une fonction négligeable $\mu$ tel que :

$$\forall P \in \Pi,\ |p[A(\mathcal{O}(P) = 1] - p[S^P(1^{|P|}) = 1]| \leq \mu(|P|).$$

Cette propriété stipule que la version obfusquée $\mathcal{O}(P)$ est parfaitement inexpugniable, dans la mesure où on ne peut pas espérer apprendre plus de la rétro-ingénierie du programme $\mathcal{O}(P)$ que de la simple observation de ses entrées/sorties.
Nous pouvons remarquer que l'action de rétro-ingénierie est formalisée par un calcul de prédicat. On considère donc l'exigence la plus faible sur ce qui peut être calculé à partir de $\mathcal{O}(P)$. L'attaquant est donc en train d'essayer de décider une propriété donnée du programme $P$.
La propriété de la boîte noire virtuelle exprime le fait que la distribution des sorties d'un algorithme probabiliste d'analyse $A$ appliqué au programme obfusqué $\mathcal{O}(P)$ est presque partout égale à la distribution des sorties d'un simulateur $S$ ayant un accès par oracle au programme $P$ (le programme $S$ n'a pas accès à la description du programme $P$, mais pour chaque entrée $x$, il a accès à $P(x)$ en temps polynomial relativement à la taille de $P$. Un accès par oracle au programme $P$ est équivalent à un accès aux seules entrées/sorties du programme $P$). Intuitivement, la propriété de la boîte noire virtuelle stipule simplement que tout ce qui peut être calculé à partir de la version obfusquée $\mathcal{O}(P)$ peut être également calculée via un accès par oracle à $P$.
Un obfuscateur est dit *efficace* s'il s'exécute en temps polynomial relativement à la taille des entrées.

Un tel compilateur générique n'existe pas. La preuve [BGI+01] repose sur la construction d'un programme qui ne peut pas être obfusqué.

Ce résultat d'impossibilité démontre donc qu'un générateur de boîte noire virtuelle - qui pourrait protéger le code de tous les programmes, en l'empêchant de révéler plus d'information que n'en révèle le comportement des entrées/sorties du programme - n'existe pas. Ce résultat d'impossibilité a naturellement des conséquences importantes pour les concepteurs de mécanismes d'obfuscation adaptés au contexte WBAC. Considérons une application pratique de l'obfuscation consistant à transformer un chiffrement symétrique en un chiffrement asymétrique, en

[2] Une fonction $\mu : \mathbb{N} \to \mathbb{N}$ est dite négligeable si pour tout polynôme $P \geq 0$, $\exists N \in \mathbb{N} \forall n \geq N\ \mu(n) \leq 1/P(n)$. Une fonction négligeable est donc une fonction qui croît plus lentement que l'inverse de tout polynôme.

obfusquant un schéma de chiffrement à clé privée. Un schéma de chiffrement à clé privée $(G, E, D)$ (où $G$ désigne l'algorithme de génération de clé, $E$ est l'algorithme de chiffrement et $D$ l'algorithme de déchiffrement) est dit non-obfuscable s'il existe $A \in PPT$ et une fonction négligeable $\mu$ tel que :

$$p_{K \xleftarrow{R} \mathbb{F}_2^k}[A(\widetilde{E}_K) = K] \geq 1 - \mu(k)$$

où $\widetilde{E}_K$ est un circuit quelconque calculant la fonction de chiffrement avec la clé $K$ (et $K \xleftarrow{R} \mathbb{F}_2^k$ désigne une variable aléatoire distribuée uniformément sur $\mathbb{F}_2^k$). Un attaquant est donc capable, à partir de la donnée de n'importe quel circuit calculant la fonction de chiffrement, de retrouver la clé $K$. L'existence de schéma de chiffrement à clé privée non-obfuscable est conditionnée à l'existence de schéma de chiffrement. Ce résultat établit clairement que tous les schémas de chiffrement à clé privée ne se prêtent pas à l'obfuscation. Cependant, remarquons que ce résultat ne prouve pas qu'il n'existe pas de schéma de chiffrement à clé privée tel que l'on puisse donner à l'attaquant un circuit qui calcule l'algorithme de chiffrement sans perte de sécurité. Il prouve en revanche qu'il n'existe pas de méthode générale permettant de transformer tout schéma de chiffrement à clé privée en un système de chiffrement à clé publique en obfusquant l'algorithme de chiffrement.

Le problème de la construction d'un schéma de chiffrement à clé privée vérifiant la propriété de la boîte noire virtuelle (donc résistant dans le contexte WBAC) est donc un problème ouvert, même si le résultat d'impossibilté concernant un moyen générique d'y parvenir peut apparaître décourageant pour le concepteur de produit de sécurité. Comme nous allons le voir dans la section 2.5, l'obfuscation par utilisation d'un réseau de lookup tables *encodées* permet d'obtenir à partir des algorithmes DES et AES des versions plus résistantes en boîte blanche. Le principe général consiste à exprimer l'algorithme de chiffrement comme une séquence (ou un réseau) de *lookup tables*, et d'obfusquer ces tables en *encodant* leurs entrées et sorties. *Une lookup table est un tableau à une dimension. Ce tableau comporte $2^n$ indices et chaque élément du tableau est de taille $m$ bits.*[3] *Le principe d'une lookup table est de remplacer une fonction $g$ prenant en entrée des mots de $n$ bits et retournant des mots de $m$ bits. Cette représentation permet de lire instantanément la sortie de $g$ appliquée à $w \in \{0,1\}^n$ en lisant la valeur se trouvant dans sa lookup table à l'indice $w$.*
Toutes les opérations arithmétiques (y compris par exemple l'addition modulo 2 avec une clé de tour) peuvent être représentées par le parcours d'un réseau de *lookup table*. L'utilisation d'encodages des entrées/sorties rend chaque lookup table localement sûre[4] : il n'est pas possible d'extraire la moindre information, en particulier la clé embarquée.

Les cryptanalyses des implémentations boîte blanche du DES [JBF03, PW05, WMGP07, GMQ07] et de l'AES [BGEC04] établissent cependant que le problème de la construction d'un schéma de chiffrement à clé privée vérifiant la propriété de la boîte noire virtuelle reste entier.

Le modèle idéal d'obfuscateur permettant de transformer tout programme en une boîte noire virtuelle n'est pas implémentable. En particulier, il n'y a pas de transformation générale permettant à partir d'un algorithme de chiffrement et d'une clé, d'obtenir une version obfusquée de cet algorithme que l'on puisse publier sans livrer d'information sur la clé qu'elle contient. Cependant, ce formalisme ne permet pas d'établir qu'il est impossible de noyer une clé dans un algorithme afin de transformer un chiffrement à clé privée en chiffrement à clé publique. Nous nous proposons donc d'appréhender ce problème par la pratique, en évaluant les propositions les plus pertinentes dans ce domaine de recherche.

### 2.2.2 Notes sur des modèles d'obfuscateur moins restrictifs

Plusieurs tentatives ont été faites pour assouplir ce modèle, de manière à obtenir des résultats positifs pour l'obfuscation. Il est possible de modifier la propriété de la boîte noire virtuelle, de manière à la rendre moins inatteignable.

[3] c'est donc une des représentations d'une fonction vectorielle de $\mathbb{F}_2^n \to \mathbb{F}_2^m$

[4] Les lookup tables sont randomisées, de manière à obfusquer leur fonctionnement interne : les entrées/sorties de ces lookup tables sont encodées par des bijections aléatoires. L'utilisation de cet encodage assure une sécurité locale, c'est-à-dire que la lookup table $g \circ T \circ f^{-1}$ encodée par utilisation des bijections $f$ et $g$ ne fournit pas d'information sur la lookup table $T$ d'origine. Étant donnée une lookup table $T'$ quelconque, il existe toujours deux bijections $f'$ et $g'$ telles que $g' \circ T' \circ f'^{-1} = g \circ T \circ f^{-1}$ (prendre par exemple $f' = f \circ T^{-1}$ et $g' = g \circ T'^{-1}$). Remarquons que lorsque la fonction $T$ n'est pas bijective, la sécurité locale n'est plus garantie, et une attaque statistique peut permettre d'extraire de l'information sur la lookup table $T$. Cette sécurité locale est évaluée par une mesure d'ambiguïté, qui exprime la difficulté à laquelle se heurte un attaquant essayant de supprimer ces encodages parasites (voir section 2.6.2 pour une définition de la mesure d'ambiguïté).

Citons notamment la $\tau$-obfuscation [BF07], où l'idée n'est plus de rechercher une obfuscation parfaite, mais plutôt une résistance efficace au moins durant une période $\tau$ aux transformations de dé-obfuscation. Plus précisement, un $\tau$-obfuscateur satisfait la propriété de la boîte noire virtuelle modifiée : $\forall A \in PPT$, $\exists S \in PPT$ tel que :

$$\forall P \in \Pi,\ p[A(\mathcal{O}(P), 1^{\tau \times t(\mathcal{O}(P))}) = 1] \backsimeq p[S^P(1^{|P|}) = 1].$$

Cette propriété établit que tout résultat qui peut être calculé en un temps inférieur à $\tau \times t(\mathcal{O}(P))$ - où $t(\mathcal{O}(P))$ est le temps nécessaire pour obfusquer $P$ - est en fait calculable par le biais d'un oracle au programme $P$.

Il est également possible d'exprimer les propriétés caractéristiques d'un obfuscateur idéal dans un modèle moins contraignant. Le modèle de l'oracle aléatoire a été utilisé [LPS04] pour redéfinir la notion d'obfuscateur et obtenir des résultats positifs d'obfuscation. Il est en effet possible de construire une classe de fonctions qui sont obfusquables dans ce modèle : les fonctions points, c'est-à-dire les fonctions booléennes $1_\alpha : \mathbb{F}_2^k \to \mathbb{F}$ définies par : $1_\alpha(x) = 1$ si $x = \alpha$, 0 sinon. Pour les oracles aléatoires $\mathcal{R} : \mathbb{F}_2^* \to \mathbb{F}_2^{2k}$, l'obfuscateur $\mathcal{O}^{\mathcal{R}}$ associe au programme $1_\alpha$ le programme $\mathcal{O}^{\mathcal{R}}(1_\alpha)$ défini par : $\forall x \in \mathbb{F}_2^k$, $\mathcal{O}^{\mathcal{R}}(1_\alpha)(x) = 1$ si $\mathcal{R} = \mathcal{R}(\alpha)$, 0 sinon. En d'autre terme, dans ce modèle, la méthode la plus classiquement utilisée pour dissimuler un mot de passe (le stockage d'une empreinte numérique $r = \mathcal{R}(\alpha)$) peut être vue comme l'obfuscation d'une fonction point.
De la même manière, le mécanisme de génération environnementale (voir section 2.3.2) conduit à une obfuscation effective de la clé dans le modèle de l'oracle aléatoire.

**Problème de l'implémentation du modèle de l'oracle aléatoire** On espère que tout protocole conçu dans ce modèle idéal restera sûr lorsqu'implémenté en utilisant une fonction facile à évaluer (comme par exemple une fonction de hachage fixée $f(k, .) : \mathbb{F}_2^* \to \mathbb{F}_2^{l(k)}$) à la place de l'oracle aléatoire. Il a été démontré dans [CGH04] que les systèmes dont la sécurité repose sur l'impossibilité de corrélation (*correlation intractability*) de leur oracle aléatoire peuvent être sûrs dans le modèle de l'oracle aléatoire et ne plus l'être dans une implémentation du système utilisant une fonction ou un ensemble de fonction.

**Théorème 2.2.1** *(Implémentation non sûre de l'oracle aléatoire [CGH04]). Il existe des schémas de chiffrement (et de signature) qui sont sûrs dans le modèle de l'oracle aléatoire, mais n'ont pas d'implémentation sûre par des ensembles de fonctions. En outre, chacun de ces schémas possède un attaquant générique qui à partir de la description d'une implémentation de l'oracle, est en mesure de casser le schéma qui utilise cette implémentation.*

Remarquons que ce résultat confirme celui de Barak & al. [BGI+01]. Lorsque l'on cherche à modéliser l'action de rétro-ingénierie, on ne peut pas supposer que la seule chose que peut faire l'attaquant avec la description de l'implémentation de l'oracle est de l'invoquer sur les entrées de son choix : il ne faut pas ignorer que comme c'est courant en théorie de la complexité, tout ou partie du code du programme peut être donné en entrée au programme lui-même, et donc que disposer de la description d'une fonction est beaucoup plus puissant que le fait d'avoir un accès en boîte noire à cette fonction.
Le résultat de Barak & al. est d'ailleurs complémentaire, dans la mesure où il établit qu'une méthode naturelle[5] permettant d'obtenir des ensembles de fonctions adéquats ne permet pas d'obtenir une implémentation sûre quel que soit le protocole sûr dans le modèle de l'oracle aléatoire considéré.

## 2.3 Utilisation de la cryptographie par un virus à des fins d'extorsion

Nous avons rappelé dans la section précédente quelques uns des résultats théoriques fondamentaux concernant l'obfuscation d'un algorithme. Le résultat d'impossibilité de l'obfuscateur idéal stipule qu'il n'existe pas de transformation générale permettant à partir d'un algorithme de chiffrement et d'une clé, d'obtenir une version obfusquée de cet algorithme que l'on puisse publier sans livrer d'information sur la clé qu'elle contient. Nous avons vu cependant qu'il est possible d'exprimer les propriétés caractéristiques d'un obfuscateur idéal dans un modèle moins contraignant, comme celui de l'oracle aléatoire, et d'obtenir des résultats positifs pour certaines classes de fonctions, même si se pose le problème résiduel de l'implémentation sûre de ce modèle.

[5] La méthode consiste à appliquer une transformation (qui modifie le code du programme sans en modifier les fonctionnalités) à un ensemble de fonctions pseudo-aléatoires, c'est à dire à un ensemble de fonctions qui ne peuvent pas être distinguées d'un oracle aléatoire lorsque l'on ne dispose que d'un accès par oracle à ces fonctions.

Ce cadre formel permet de représenter convenablement la sécurité d'un mécanisme cryptographique dans le contexte WBAC. Nous présentons dans cette section trois types différents de mécanismes de gestion des clés adaptés au contexte WBAC et au contexte viral, dans le cas particulier de programmes viraux spécialisés dans l'extorsion.
L'étude des mécanismes viraux pour l'extorsion a été appelée cryptovirology [YY96, YY04]. Après qu'un virus a déclenché sa charge finale, les effets sur le système de la victime peuvent être irréversibles pour celui-ci mais pas pour l'auteur du virus. Celui-ci peut alors extorquer de l'argent à la victime en échange du moyen de restaurer ses données.
Un premier virus de ce type a été observé en 1989 (cheval de Troie AIDS). Il utilisait un simple chiffrement par substitution. Depuis quelques années, une augmentation de ce type d'infections informatiques a cependant été observée et relayée par la presse spécialisée (on parle de phénomène *ransomware* [Gaz08]).

### 2.3.1 Utilisation de la cryptographie symétrique/asymétrique pour l'extorsion

L'utilisation de la cryptographie asymétrique permet au virus d'éviter de transporter une clé de déchiffrement capturable. Les données de la victime sont chiffrées au moyen de la composante publique $K_{pub}$ d'une bi-clé asymétrique $(K_{pub}, K_{priv})$. L'auteur du virus donne la composante privée $K_{priv}$ de la bi-clé en échange d'argent. L'inconvénient de la cryptographie asymétrique est sa lenteur.
Une première solution à ce problème consiste à ne chiffrer que certains fichiers (cheval de Troie *PGPCoder* [Gaz08]).
Une seconde solution consiste à générer aléatoirement une clé $K$ puis à utiliser un algorithme de chiffrement symétrique $E_K$ plus efficace pour chiffrer les données de la victime. Le virus chiffre ensuite la clé $K$ avec la composante publique $K_{pub}$ de la bi-clé $(K_{pub}, K_{priv})$. La victime doit transmettre à l'auteur du virus $K_{pub}[K]$ et la somme d'argent extorquée. Ce dernier peut alors transmettre la clé $K$ à la victime, lui permettant de restaurer ses données sans dévoiler la composante privée de la bi-clé asymétrique.
Nous voyons donc que l'utilisation conjointe de la cryptographie symétrique et asymétrique permet de concevoir des virus[6] cryptographiques spécialisés dans l'extorsion.

### 2.3.2 Gestion de clé par génération environnementale

La cryptographie peut être utilisée pour résoudre d'autre problèmes posés au virus cryptographiques : la gestion de clé et le polymorphisme.
Si la cryptographie asymétrique utilisée seule résout en partie[7] le problème de gestion de clé, ses limitations principales sont la lenteur et le manque de discrétion concernant la détection de la fonction de chiffrement.
L'utilisation conjointe de la cryptographie asymétrique et de la cryptographie symétrique pose un problème résiduel de gestion de clé : celle-ci est générée sur le système cible, puis écrite dans un fichier. Des traces peuvent subsister en mémoire, permettant à une société spécialisée de retrouver la clé $K$. En outre, l'étude du virus en laboratoire permet de développer une procédure de détection des fonctions de génération d'aléa et de chiffrement.

La génération environnementale de clé [RS98, Fil05c, Fil07c] adresse spécifiquement le problème de la gestion des clés et apporte une solution dans le cas de virus dirigés ou semi-dirigés, c'est-à-dire conçus pour ne s'exécuter que sur une plate-forme cible possédant des caractéristiques environnementales (pas nécessairement de la plate-forme) déjà connues du concepteur de virus. La génération environnementale de clé est un mécanisme qui permet de ne pas avoir à stocker la clé dans l'exécutable. Celle-ci est générée, par exemple, par application d'une fonction de hachage à des données d'activation présentes dans l'environnement d'exécution du logiciel. Soit $X$ un entier correspondant à cette observation environnementale, $Y$ la valeur nécessaire pour l'activation (et transportée par le programme), $h$ une fonction de hachage et $R_1$, $R_2$ deux nonces. Alors des constructions possibles, parmi d'autres, sont [RS98] :

on pose $K = X$ où le test est : est-ce que $Y = h(X)$ ? ;
on pose $K = h(X)$ où le test est : est-ce que $Y = h^2(X) = h \circ h(X)$ ? ;
on pose $K = h(X_1, \ldots, X_n)$ où le test est : est-ce que $h(X_n) = Y$ ? ;
on pose $K = h(R_1, X) \oplus R_2$ où le test est : est-ce que $Y = h(X)$ ?

[6] Précisons que les mécanismes cryptographiques utilisés ne concernent que la charge finale des programmes viraux, pas leur éventuelle fonction de réplication.
[7] Seulement pour la charge finale. Pour la partie strictement virale, il faut déchiffrer.

La caractéristique la plus importante de chaque construction est que la connaissance de $Y$ ne fournisse aucune information sur $K$.
Inspirée de ce principe, la *génération environnementale de code* [AdJ06] est un mécanisme qui permet de générer du code dynamiquement, en fonction de données d'activations présentes dans l'environnement d'exécution du logiciel. Lors de la phase de protection du logiciel, un groupe d'instructions $I$ est supprimé. Etant donnée une clé $K$, et une fonction de hachage $h$, une valeur $S$ est recherchée en force brute de manière à satisfaire à l'équation $h(K||S) = I$. Lors de la phase d'exécution du logiciel, la clé $K$ est générée par application d'une fonction de hachage à des données d'activation présentes dans l'environnement d'exécution du logiciel. Le groupe d'instructions $I$ est alors généré par $h(K||S) = I$.
Lors d'une analyse statique de code (ou d'une analyse dynamique dans un environnement ne possédant pas les mêmes propriétés que l'environnement cible), l'analyste connaît $S$ et le domaine de valeurs de $K$. Il ne connaît pas le morceau de code généré. Pour retrouver le morceau de code, l'attaquant doit parcourir l'espace de $K$ et pour chaque valeur, tester le code généré. Selon la nature (sémantique) du code généré, cette attaque par force brute peut être très difficile à mener.
Une analyse critique de ces mécanismes est développée dans [Fil07c]. Remarquons également que lors d'une analyse dynamique du code dans l'environnement cible, l'attaquant peut retrouver la clé $K$ et le code associé.

### 2.3.3 Gestion de clé et diversification par cryptographie boîte blanche

La cryptographie boîte blanche [CEJvO02a, CEJvO02b] cherche une implémentation particulière des algorithmes de chiffrement permettant de sécuriser la gestion des clés. Les algorithmes de chiffrement symétriques boîte blanche visent à assurer la confidentialité de la clé dans le contexte WBAC. Ces implémentations proposent un mécanisme intrinsèque de diversification des instances, permettant de produire de nombreuses versions polymorphes du chiffreur. L'implémentation réduit la portion de code à sa plus simple expression, en banissant toute opération arithmétique classique. Un tel code est beaucoup plus facile à diversifier par utilisation d'un moteur de polymorphisme (mutation engine).
Une implémentation particulière d'un algorithme de chiffrement itératif par bloc permet donc d'obtenir plusieurs propriétés essentielles pour une fonction de chiffrement ou de déchiffrement utlisée par un virus cryptographique : un mode de gestion de clé adapté au contexte WBAC, une asymétrisation d'un algorithme symétrique, une diversification des données de l'algorithme (le code de l'algorithme doit être diversifié par utilisation d'un moteur de mutation. Nous allons voir cependant que le code n'utilise qu'une faible portion du jeu d'instructions du CPU, sans instruction arithmétique, et qu'il se prête donc plus facilement à la diversification par utilisation d'un moteur de mutation) :

- Dissimulation de la clé dans l'algorithme : il est difficile de retrouver la clé à partir de la donnée du code de l'algorithme de chiffrement (ou du code de l'algorithme de déchiffrement) ;
- Asymétrisation de l'algorithme : il est difficile de forger la fonction de chiffrement à partir de la fonction de déchiffrement (et inversement) ;
- Diversification du code et des données de l'algorithme : il est difficile de forger une signature à partir du code de l'algorithme, car celui-ci n'utilise que des instructions non-arithmétiques et donc les instances (code et données) peuvent être très diversifiées.
- Compromis temps d'exécution/espace mémoire : le temps d'exécution, même s'il est plus important que celui de l'algorithme boîte noire (l'espace de stockage des lookup tables est non-négligeable et impose un temps de chargement en mémoire avant exécution), reste bien inférieur à celui d'un algorithme de chiffrement asymétrique.

La première propriété peut être vue comme une instance de la propriété de la boîte noire virtuelle. En principe, la notion de difficulté à retrouver la clé à partir de la donnée du code de l'algorithme de chiffrement signifie que la donnée du code n'apporte pas d'information sur la clé à l'attaquant. En pratique, cela signifie qu'il ne doit pas être plus facile d'attaquer l'implémentation boîte blanche de l'algorithme de chiffrement que d'attaquer ce même algorithme en boîte noire.
La seconde propriété est liée à la première dans la mesure où la connaissance de la clé et de l'algorithme permet de forger la fonction de chiffrement à partir de la fonction de déchiffrement. En pratique, il ne doit pas être possible d'inverser la fonction de déchiffrement sans la clé.
La troisième propriété repose sur l'utilisation d'un moteur de mutation. Nous conjecturons que le parcours d'un

réseau de *lookup table* est plus facile à diversifier qu'un morceau de code comportant des instructions arithmétiques.

Nous présentons à la section 2.5 deux exemples d'implémentation boîte blanche d'algorithmes de chiffrement itératif par bloc (le DES et l'AES) et donnons une mesure de la résistance de ces implémentations dans le contexte WBAC à la section 2.6.

*A priori*, ce type de mécanisme trouve bien sa place à côté des autres primitives cryptographiques de gestion des clés (génération environnementale de clé par utilisation de fonction de hachage, cryptographie asymétrique et/ou symétrique) utilisables par un virus à des fins d'extorsion. Nous illustrons cette position dans la section suivante, par des exemple d'utilisation de la cryptographie boîte blanche dans le contexte viral.

## 2.4 Exemples d'utilisation en contexte viral

Nous présentons dans cette section des exemples d'applications potentielles de la cryptographie boîte blanche dans le contexte viral. Nous supposons que les propriétés caractérisant ce mécanisme cryptographique (voir section 2.3.3) sont bien vérifiées.

### 2.4.1 Contrôle d'intégrité en boîte blanche

Avant de présenter des exemples d'utilisation de la cryptographie boîte blanche par un virus à des fins d'extorsion, il est intéressant de présenter l'utilisation qui en est faite dans la chaîne de compilation spécialisée CSS (*Cloakware Security Suite* [Clo07]). Le programme comporte :

- une première partie implémentant la fonction de hachage $H$ et l'algorithme de déchiffrement boîte blanche $\mathrm{WBD}_K$ ;
- une zone de stockage des hachés $\mathrm{WBE}_K[H(\mathrm{BODY})]$, appelé *Voucher*.

La vérification d'intégrité en boîte blanche correspond au test suivant :

$$H(\mathrm{BODY}) == \mathrm{WBD}_K[\mathrm{WBE}_K[H(\mathrm{BODY})]]?\mathrm{OK} : \mathrm{KO}$$

Puisqu'il est difficile de retrouver la clé $K$ ou de reconstruire la fonction de chiffrement $\mathrm{WBE}_K$ à partir de l'analyse (en contexte WBAC) de la fonction de déchiffrement $\mathrm{WBD}_K$, l'attaquant n'est pas en mesure de substituer de nouveaux hachés aux valeurs stockées dans le *Voucher*, ce qui lui aurait permis d'utiliser sans contrainte une application modifiée.
Remarquons que la fonction de vérification doit être protégée contre l'analyse dynamique.

### 2.4.2 Bombe logique

Un premier exemple élémentaire d'infection informatique utilisant la cryptographie boîte blanche à des fins d'extorsion comporte (en plus de portions de code bénignes et d'une éventuelle condition de déclenchement) une partie $\mathrm{WBE}_K$ implémentant l'algorithme de chiffrement boîte blanche et dont la mission est de chiffrer tout ou partie des données de la victime. La victime n'est pas en mesure de retrouver la clé $K$ utilisée par une analyse en contexte WBAC de la fonction $\mathrm{WBE}_K$.

L'utilisation couplée d'un moteur de polymorphisme MUT et d'un générateur de bijections aléatoires RNG lui permet de créer un très grand nombre de versions de ce programme, pour une unique mise à la clé $K$.

Comme nous l'avons rappelé en introduction, dans le contexte viral, un moteur de mutation, c'est-à-dire un moteur de polymorphisme, doit renforcer à la fois la diversité et la résistance vis-à-vis de la reconnaissance de motif. Le polymorphisme [Qoz99] et le métamorphisme [Fil07b] peuvent être formalisés en terme de productions de grammaire. La difficulté à reconnaître un virus correspond à l'expressivité nécessaire de la machine ou de l'automate qui est capable de reconnaître le langage $L(G)$ généré par la grammaire $G$ en appliquant ses règles de production (voir table 2.1).
Lorsqu'il formalisa le premier les grammaires génératives en 1956 [Cho56], Chomsky proposa la classification suivante :

- les grammaires de type 0 (grammaires sans restriction - unrestricted) produisent les langages récursivement énumérables, c'est-à-dire les langages qui peuvent être reconnus par une machine de Turing (que l'on autorise à boucler sur un mot qui n'est pas du langage). Donc leurs productions simulent les machines de Turing. Par conséquent, le problème de l'arrêt est réductible au problème de décider si $x \in L(G)$ ou pas ;
- les grammaires de type 1 (grammaires contextuelles - context-sentitive) produisent les langages qui peuvent être reconnus par une machine de Turing non déterministe bornée linéairement (machine de Turing non déterministe qui n'utilise de sa mémoire infinie qu'une portion dépendant linéairement de la taille du mot testé) ;
- les grammaires de type 2 (grammaires non contextuelles, hors-contextes ou algébriques - context-free), produisent les langages qui peuvent être reconnus par un automate à pile non déterministe ;
- les grammaires de type 3 (grammaires régulières - regular) produisent les langages qui peuvent être reconnus par un automate à états finis déterministe (langages qui peuvent être décrits par les expressions rationnelles).

| Type de grammaire | Complexité |
|---|---|
| type 0 | indécidable |
| type 1 | NP-complet |
| type 2 | NP-complet |
| type 3 | P |

Tab. 2.1 – Complexité du problème de détection de la production $L(G)$ d'une grammaire $G$, ou de manière équivalente du problème : est-ce que $x$ appartient à $L(G)$ ?

Le type de grammaire générative est donc crucial lorsque l'on conçoit un moteur de polymorphisme [Fil07b]. Ce point sera discuté plus en détail dans la section 2.6.4.

### 2.4.3 Virus polymorphe

Un autre exemple élémentaire de virus (virus polymorphe) utilisant la cryptographie boîte blanche comporte :
- une première partie $\mathrm{WBE}_K$ implémentant l'algorithme de chiffrement boîte blanche ;
- une seconde partie $\mathrm{WBD}_K[\mathrm{RNG}\|\mathrm{MUT}]$ comportant le générateur de bijections aléatoires RNG et le moteur de mutation MUT chiffrés au moyen de l'algorithme de déchiffrement $\mathrm{WBD}_K$.

La première partie du programme viral $\mathrm{WBE}_K$ chiffre tout ou partie des données de la victime, puis déchiffre la seconde partie du corps du virus.
L'exécution du générateur de bijections aléatoires RNG puis du moteur de mutation MUT permettent de générer le couple de fonctions $(\mathrm{WBE}_{K'}, \mathrm{WBD}_{K'})$. La clé $K'$ est éventuellement communiquée à l'auteur du virus (avec les informations sur l'ordinateur cible). L'algorithme $\mathrm{WBD}_{K'}$ est utilisé pour chiffrer $\mathrm{RNG}\|\mathrm{MUT}$ (ou $\mathrm{MUT}(\mathrm{RNG}\|\mathrm{MUT})$). La nouvelle instance du virus est $\mathrm{WBE}_{K'}\|\mathrm{WBD}_{K'}(\mathrm{RNG}\|\mathrm{MUT})$ (ou $\mathrm{WBE}_{K'}\|\mathrm{WBD}_{K'}(\mathrm{MUT}(\mathrm{RNG}\|\mathrm{MUT}))$).

Un exemple d'implémentation boîte blanche de l'algorithme DES est donné dans la section C.1 de l'annexe C. Elle fournit un exemple de fonction $\mathrm{WBE}_K$.

### 2.4.4 Virus métamorphe

Un autre exemple, sans code auto-modifiable (virus métamorphe), comporte :
- une première partie $\mathrm{WBE}_K$ implémentant l'algorithme de chiffrement boîte blanche ;
- une seconde partie $\mathrm{RNG}\|\mathrm{MUT}$ comportant le générateur de bijections aléatoires RNG et le moteur de mutation MUT.

La première partie du programme viral $\mathrm{WBE}_K$ chiffre tout ou partie des données de la victime. L'exécution du générateur de bijections aléatoires RNG puis du moteur de mutation MUT permettent de générer la fonction $\mathrm{WBE}_{K'}$. La clé $K'$ est éventuellement communiquée à l'auteur du virus (avec les informations sur l'ordinateur cible). La nouvelle instance du virus est $\mathrm{WBE}_{K'}\|\mathrm{MUT}(\mathrm{RNG}\|\mathrm{MUT})$.

### 2.4.5 Remarques

#### Diversification du code du mécanisme WBAC

La primitive de chiffrement ou de déchiffrement n'a pas besoin d'être protégée par des mécanismes destinés à entraver l'analyse dynamique, dans la mesure où l'examen pas à pas de l'exécution (examen du contexte à chaque étape de l'exécution) ne fournit pas d'information sur la clé. En revanche, il est nécessaire de diversifier le code de manière à rendre difficile la signature du virus. Les opérations réalisées ne sont pas arithmétiques. Il s'agit simplement du parcours de *lookup tables*. Les *lookup tables* sont diversifiées par construction. Cependant, il est à noter que nous sommes confrontés au problème de la diversification du générateur de bijections aléatoires RNG, du moteur de mutation MUT et des instructions CPU nécessaires pour parcourir le réseau de lookup tables. Ce problème est discuté dans la section 2.6.4.

#### Comparaison avec la méthode hybride symétrique/asymétrique

Les points forts de la cryptographie boîte blanche par rapport à l'utilisation conjointe des schémas asymétrique et symétrique telle que proposée par [YY04] sont que :
- l'on peut se dispenser de la cryptographie asymétrique ;
- le mécanisme de diversification du chiffreur (ou du déchiffreur) est intrinsèque ;
- le code est plus aisé à obfusquer puisqu'il ne comporte pas d'expressions arithmétiques.

#### Comparaison avec la génération environnementale de clé

Le point fort de la cryptographie boîte blanche par rapport à la génération environnementale de clé est que le virus préserve sa liberté. Il n'est pas dirigé vers une plate-forme spécifique et ne dépend pas d'un environnement spécifique.

Comparé aux deux mécanismes mentionnés, l'inconvénient de ce mécanisme est qu'il n'est pas encore assez robuste à la cryptanalyse, comme nous allons le voir dans la section 2.6. Même si les cryptanalyses ne sont pas triviales à mettre en œuvre, les deux propriétés énoncées à la section 2.3.3 (dissimulation de la clé dans l'algorithme et asymétrisation de l'algorithme) ne sont pas vérifiées.

## 2.5 Implémentations boîte blanche du DES et de l'AES

Nous présentons dans cette section les principes de l'implémentation boîte blanche de deux algorithmes bien connus : DES et AES.

### 2.5.1 Implémentation de WB-DES

Nous allons présenter la méthode publiée par Chow & al. dans [CEJvO02a] pour rendre difficile l'extraction de la clé en boîte blanche. Le principe consiste à implémenter une version spécialisée de l'algorithme DES qui embarque la clé $K$, et qui n'est capable d'effectuer qu'une seule des deux opérations chiffrer ou déchiffrer. Cette implémentation est résistante en boîte blanche dans la mesure où il est difficile d'extraire la clé $K$ en observant les opérations effectuées par le programme et qu'il est difficile de forger la fonction de chiffrement à partir de l'implémentation de la fonction de déchiffrement, et vice-versa.

L'idée consiste à exprimer l'algorithme DES comme une séquence (ou un réseau) de *lookup tables*, et d'obfusquer ces tables en *encodant* leurs entrées et sorties.
Toutes les opérations du chiffrement par blocs, comme l'addition modulo 2 avec la clé de tour, sont embarquées dans ces lookup tables. Ces tables sont randomisées, de manière à obfusquer leur fonctionnement. La représentation du DES sous forme de *lookup tables* nécessite de découper les transformations réalisées durant les 16 tours d'une manière différente. La Figure 2.1 illustre ce redécoupage. Chaque tour du DES est découpé en deux couches, l'une qualifiée de non-linéaire contient les S-Box, tandis que la seconde, qualifiée de linéaire rassemble les transformations linéaires telles que l'expansion, l'opération xor et la permutation. Les entrées de cette nouvelle représentation sont désormais des mots binaires de 96 bits. Trois nouvelles variables sont introduites, il s'agit de $X_{r-1}$, $\overline{R_{r-1}}$ et $Y_r$.
- $X_{r-1}$ représente la sortie de l'expansion, il s'agit d'un mot de 48 bits,

- $\overline{R_{r-1}}$ représente les 16 bits de $R_{r-1}$ qui ne sont pas dédoublés par l'expansion (ce sont les deux bits du milieu des entrées des S-Box),
- $Y_r$ représente la concaténation des sorties des S-Box, il s'agit d'un mot binaire de 32 bits.

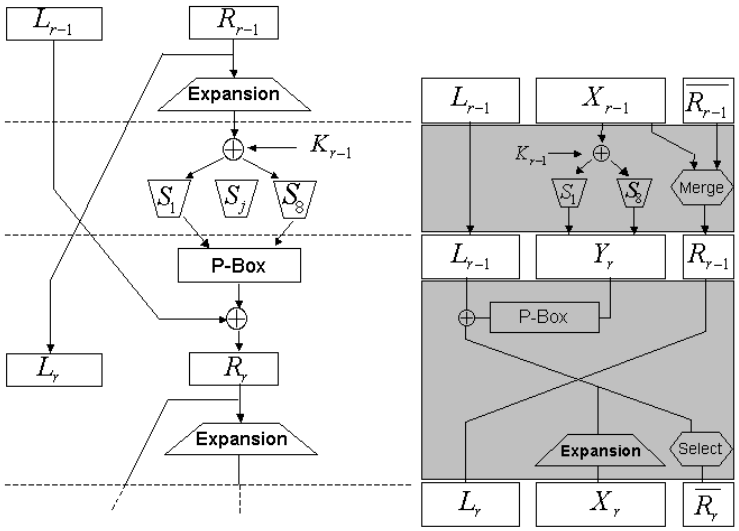



FIG. 2.1 – Un tour du DES et son équivalent en boîte blanche

Des mots des 96 bits transitent donc entre les *lookup tables*. Ces mots ou états internes sont représentés dans la figure 2.1 par la concaténation des blocs $L_{r-1} \in \mathbb{F}_2^{32}$, $X_{r-1} \in \mathbb{F}_2^{48}$ et $\overline{R_{r-1}} \in \mathbb{F}_2^{16}$ : $L_{r-1}||X_{r-1}||\overline{R_{r-1}}$.

L'état interne $L_{r-1}||X_{r-1}||\overline{R_{r-1}}$ du tour $r$ est subdivisé en mots de 8 bits et est donc représenté comme la concaténation de 12 mots binaires : $v_1^r||v_2^r||\ldots||v_{12}^r$. Chaque mot $v_i^r$ représente l'entrée encodée d'une T-Box $T_i^r$ qui peut être de deux sortes :
- soit une T-Box non linéaire, embarquant une S-Box,
- soit une *bypass* T-Box.

Nous pouvons remarquer qu'il est donc possible de défaire le schéma de Feistel de l'algorithme DES et de l'implémenter comme un schéma de substitution/permutation[8], comme c'est le cas de l'AES.

La technique utilisée pour embarquer ces clés consiste à représenter le DES comme un réseau de lookup tables, et d'appliquer des encodages des entrées/sorties de manière à cacher les clés. L'utilisation d'encodages des entrées/sorties rend chaque lookup table localement sûre. L'idée principale de cette méthode d'obfuscation est donc d'être capable de représenter l'algorithme DES tout entier comme une unique lookup table qui est localement sûre, c'est-à-dire de laquelle il n'est pas possible d'extraire de l'information. Malheureusement ce n'est pas possible car la représentation d'une fonction vectorielle booléenne $\mathbb{F}_2^n \to \mathbb{F}_2^n$ nécessite un espace mémoire important (exponentiel relativement à la taille des entrées, c'est-à-dire que le paramètre $n$ fait exploser l'espace mémoire - voir table 2.2). De manière à prendre en considération cette contrainte, des lookup tables de plus petite tailles sont utilisées. Après avoir implémenté l'algorithme DES tout entier par des lookup tables, l'implémentation n'est pas encore sûre. L'étape suivante a pour objectif d'encoder ces lookup tables, de manière à prévenir toute fuite d'information relatives aux clés de tour. La technique consiste à composer chaque T-Box avec des bijections linéaires en entrée et en sortie. Étant données deux bijections aléatoires $f$ et $g$ (compatibles avec $T$), la T-Box est remplacée par :

$$f \circ T \circ g.$$

[8] Plus précisément, chaque tour du DES est séparé en une étape de substitution non linéaire, mettant en jeu les S-Boxes, et une étape linéaire affine.

| $n$ (lookup table $\mathbb{F}_2^n \to \mathbb{F}_2^n$) | espace mémoire $n.2^n/8$ (bytes) |
|---|---|
| 8 | 256 Bytes |
| 16 | 128 KB |
| 24 | 48 MB |
| 32 | 16 GB |

TAB. 2.2 – espace mémoire nécessaire au stockage des lookup tables

Étant données trois lookup tables adjacentes $L_1$, $L_2$ et $L_3$, et $f$ et $g$ les encodages des entrées/sorties appliqués à $L_2$, la table $L_2$ est remplacée par sa version encodée $L_2' = f \circ L_2 \circ g$. Il est donc nécessaire d'encoder les sorties de $L_1$ avec $g^{-1}$ et les entrées de $L_3$ avec $f^{-1}$ de telle sorte que :

$$L_3 \circ f^{-1} \circ L_2' \circ g^{-1} \circ L_1 = L_3 \circ L_2 \circ L_1.$$

Pour que la propriété de sécurité locale soit utile dans notre contexte, il est également nécessaire que l'attaquant ne puisse pas distinguer les T-Boxes non-linéaires (embarquant une S-Box) des T-Boxes *bypass*. Une permutation aléatoire $\pi_r$ sur l'ordre des $T_i^{K_r}$, $i = 1, \ldots, 12$ est donc appliquée.
À ce stade de l'obfuscation, les entrées des lookup tables du premier tour sont encore exposées à une attaque de type *square-like* [CEJvO02b]. Deux encodages externes $F$ et $G$ sont donc intégrés.

En résumé, il est possible d'implémenter l'algorithme DES dans son intégralité sous forme de lookup tables encodées, de telle manière qu'il paraît difficile d'extraire la moindre information d'une des lookup tables, par la seule observation de ses entrées/sorties.

### 2.5.2 implémentation de WB-AES

L'obfuscation de l'AES [CEJvO02b] est semblable à celle du DES. Il s'agit là encore d'embarquer les clés de tours dans le code de l'algorithme, afin de ne pas avoir à stocker la clé en mémoire statique ni à la charger en mémoire dynamique au moment de l'exécution. La technique utilisée pour embarquer de manière sécurisée ces clés est (comme pour le DES) de représenter l'AES sous forme d'un réseau de *lookup tables* et d'appliquer des encodages en entrée et en sortie de ces *lookup tables* de façon à cacher les clés. Rappelons que l'AES commence par une étape initiale *AddRoundKey* et que chaque tour suivant est constitué des quatre étapes : *SubBytes*, *ShiftRows*, *MixColumns* et *AddRoundKey* pour les tours $r = 1, \ldots, 9$ et des trois étapes : *SubBytes*, *ShiftRows* et *AddRoundKey* pour le tour $r = 10$. Pour l'implémentation boîte blanche cette structure est remaniée de telle sorte que le premier *AddRoundKey* fasse partie d'un tour. Plus précisemment, soit $S$ la S-Box réalisant l'opération SubBytes et soit $(k_{i,j}^r)_{(i,j)\in\{0,\ldots,3\}^2}$ la clé du tour $r$, alors nous construisons d'abord les 10 T-boxes suivantes :

$$T_{i,j}^r(x) = \begin{cases} S(x \oplus k_{i,j}^r), & 1 \leq r \leq 9 \\ S(x \oplus k_{i,j}^{10}) \oplus k_{i,j-i}^{11}. & \end{cases}$$

Nous pouvons observer qu'en raison de la linéarité de l'opération *ShiftRows*, il est possible d'intégrer la dernière opération *AddRoundKey* dans une T-Box.
L'étape suivante pour l'obfuscation de l'AES consiste à représenter *Mixcolumns* comme un réseau de *lookup tables*. La T-Box est ensuite remplacée par la composition de la T-Box avec la *lookup table* représentant à la fois les opérations *ShiftRows* et *MixColumns*.
La technique utilisée pour encoder les *lookup tables* est la même que celle qui a été décrite pour l'obfuscation du DES. Elle consiste donc à composer chaque T-Box avec des bijections non-linéaires en entrée et en sortie. De manière à parer une attaque de type *square-like*, il est nécessaire de renforcer la sécurité locale des T-Boxes par l'utilisation de *mixing bijections*, c'est-à-dire de bijections linéaires, de façon à insérer une étape de diffusion. Etant données deux bijections linéaires $m$ et $M$ compatibles avec les opérations précédentes et suivantes, une T-Box est à présent remplacée par :

$$f \circ M \circ MC_i \circ T \circ m \circ g.$$

L'insertion de telles *mixing bijections* $m$ et $M$ impose leur inversion par de nouvelles *lookup tables*. Donc une nouvelle *lookup table* est insérée entre les tours $r$ et $r+1$. Cette dernière annule à la fois les effets de la *mixing bijection*

$M$ du tour $r$ et de la *mixing bijection* $m$ du tour $r+1$. Nous obtenons la composition supplémentaire suivante : $g_{r+1}^{-1} \circ m^{-1} \circ M^{-1} \circ f_r^{-1}$.
À cette étape de l'obfuscation, les entrées des *lookup tables* du premier tour sont encore exposées à une attaque de type *square-like*. Donc deux encodages externes $F$ et $G$ sont intégrés.

En résumé, nous sommes parvenus à implémenter l'intégralité de l'algorithme AES par des *lookup tables*, de telle sorte qu'il paraît difficile d'extraire la moindre information d'une des *lookup tables* par la seule observation de ses entrées/sorties. Pour implémenter complètement l'AES boîte blanche, nous avons besoin de quatre types de *lookup tables* [CEJvO02b] :

type I Encodage externe, c'est-à-dire une fonction $\mathbb{F}_2^8 \to \mathbb{F}_2^{128}$ qui compose deux décodages d'entrées $\mathbb{F}_2^4 \to \mathbb{F}_2^4$, une bijection linéaire $\mathbb{F}_2^8 \to \mathbb{F}_2^{128}$ et 32 encodages des sorties $\mathbb{F}_2^4 \to \mathbb{F}_2^4$ ;

type II $R$-Box, c'est-à-dire une fonction $\mathbb{F}_2^8 \to \mathbb{F}_2^{32}$ qui compose deux décodages d'entrées $\mathbb{F}_2^4 \to \mathbb{F}_2^4$, une *mixing bijection* $m : \mathbb{F}_2^8 \to \mathbb{F}_2^8$, une $T$-Box, l'opération *mixcolums*, une *mixing bijection* $M : \mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ et huit encodages des sorties $\mathbb{F}_2^4 \to \mathbb{F}_2^4$ ;

type III une fonction $\mathbb{F}_2^8 \to \mathbb{F}_2^{32}$ qui compose une *mixing bijection* $M_i^{-1} : \mathbb{F}_2^8 \to \mathbb{F}_2^{32}$, quatre fois une *mixing bijection* $m^{-1} : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ et huit encodages des sorties $\mathbb{F}_2^4 \to \mathbb{F}_2^4$ ;

type IV XOR-Box, c'est-à-dire une fonction $\mathbb{F}_2^8 \to \mathbb{F}_2^4$ qui compose deux décodages d'entrées $\mathbb{F}_2^4 \to \mathbb{F}_2^4$, une *lookup table* XOR et un encodage des sorties $\mathbb{F}_2^4 \to \mathbb{F}_2^4$.

## 2.6 Evaluation

Avant de présenter les cryptanalyses de WB-DES et WB-AES, nous présentons des critères pour évaluer la sécurité d'une primitive cryptographique en contexte boîte noire. Dans le contexte WBAC, on peut envisager d'autres critères, tels que la diversité ou l'ambiguïté, permettant de rendre compte de la qualité cryptographique des composants d'un algorithme de chiffrement en boîte blanche. La diversité et l'ambiguïté sont des mesures permettant de qualifier a priori la robustesse de l'implémentation boîte blanche. Nous appliquons ces critères aux algorithmes WB-DES et WB-AES.
Nous présentons enfin les principales cryptanalyses [WMGP07, BGEC04] dont ont fait l'objet ces deux algorithmes.

### 2.6.1 Critères de sécurité boîte noire

L'analyse en boîte noire du DES a conduit à la définition de critères de sécurité généraux (comme le critère de stricte avalanche, critère de propagation, valeur xor, critère d'indépendance du bit de sortie [WT86, DT91, AT93, MA96]) des boîtes de confusion d'un système de chiffrement itératif vis-à-vis des cryptanalyses linéaires et différentielles. Une haute non-linéarité fournit une résistance vis-à-vis des attaques linéaires. Les fonctions booléennes optimalement non-linéaires sont dites courbes. La non-linéarité d'une S-box $S$ est définie comme la distance minimale de l'ensemble des combinaisons linéaires non triviale des colonnes de $S$ à l'ensemble des fonctions affines. Nous espérons que la table xor ait une distribution constante et uniformément petite. Cette seconde propriété assure une protection vis-à-vis de la cryptanalyse différentielle. Remarquons enfin que le critère de l'ordre SAC maximal est assuré dès lors que des fonctions courbes sont choisies comme colonnes de la S-box. En résumé,

- une haute non-linéarité fournit une résistance aux attaques linéaires ;
- une distribution uniformément petite de la table xor fournit une résistance aux attaques différentielles ;
- les autres propriétés (critères de stricte avalanche et du bit de sortie) assure que la S-box a de bonnes propriétés dynamiques et sont cruciales dans la conception de l'algorithme de chiffrement, en particulier concernant sa propriété de diffusion.

D'autres critères, comme l'équilibre, le degré algébrique et l'ordre d'immunité aux corrélation existent et sont importants pour la sécurité en boîte noire [Can01]. Malheureusement, toutes ces propriétés ne peuvent pas être satisfaites simultanément.

Ces critères sont fondamentaux également pour la conception et l'évaluation dans le contexte boîte blanche pour plusieurs raisons :

- d'une part une implémentation boîte blanche doit également être résistante à la cryptanalyse en boîte noire ;

- d'autre part, la cryptanalyse en boîte blanche utilise des méthodes boîtes noires, avec une granularité plus fine (ici, chaque *lookup table* peut être vue comme une boîte noire, dont on cherche à extraire de l'information ou la clé) ;
- enfin, la conception d'un algorithme de chiffrement spécialement pensé pour être résistant en boîte blanche peut s'appuyer sur ces critères pour prouver sa sécurité.

Une question importante dans notre contexte concerne le problème de la génération aléatoire de S-Boxes et la qualité cryptographique que l'on peut espérer obtenir des S-Boxes générées aléatoirement. Les S-boxes peuvent être générées soit de manière aléatoire, soit de manière déterministe. L'avantage de la première méthode est qu'il n'y a pas de structure mathématique qui puisse être potentiellement exploitée par la cryptanalyse. Les deux méthodes peuvent être utilisées simultanément pour la construction de fonctions courbes, mais les algorithmes déterministes connus sont coûteux et réduisent le niveau de diversité obtenu par randomisation.

Je présente en annexe B une revue plus détaillée des critères de sécurité boîte noire sur lesquels fonder la construction de S-Boxes (section B.1) puis le résultat de mes expérimentations concernant la génération aléatoire de S-Boxes vérifiant ces critères (section B.2). J'y étudie la possibilité de diversifier l'implémentation boîte blanche des algorithmes de chiffrement par blocs de type S/P, à travers la randomisation des S-Boxes. La conclusion de ces travaux préliminaires est qu'il est possible de générer aléatoirement des fonctions hautement non-linéaires en imposant des contraintes simples sur la forme algébrique normale de ces fonctions booléennes et de construire des S-Boxes de qualité cryptographique comparable à celles utilisées dans les implémentations boîte noire du DES et de l'AES au regard des critères de robustesse connus.

### 2.6.2 Critères de diversité et d'ambiguïté boîte blanche

La mesure de diversité consiste à compter le nombre d'implémentations différentes qu'il est possible de générer (incluant la variation des clés embarquées). Cette mesure est importante car elle qualifie la capacité de l'obfuscateur à parer, a priori, les attaques à grande échelle. Les attaques spécifiques à une instance n'ont alors qu'un impact limité.

**Définition 2.6.1** *(Diversité boîte blanche [CEJvO02b]). La diversité boîte blanche est une métrique qui calcule le nombre de constructions ou décompositions distinctes, c'est-à-dire le nombre d'étapes d'encodage possibles.*

À titre d'exemple, la table 2.3 donne les mesures de diversité des quatre types de *lookup tables* qui sont utilisées dans l'implémentation boîte blanche de l'AES[9]. Néanmoins, cette mesure ne rend pas compte de la robustesse d'une

| type | diversity |
|---|---|
| type I | $(16!)^2 \times 20160^{64} \times (16!)^{32}$ |
| type II | $(16!)^2 \times 256 \times 2^{62.2} \times 2^{256} \times (16!)^8$ |
| type III | $(16!)^2 \times 2^{256} \times (16!)^8$ |
| type IV | $(16!)^2 \times 16!$ |

Tab. 2.3 – Diversité des lookup tables de WB-AES [CEJvO02b]

implémentation particulière face à une attaque cherchant à extraire la clé embarquée. Afin de mieux qualifier la robustesse d'une implémentation il est plus intéressant de compter le nombre de constructions, c'est-à-dire le nombre de clés et bijections aléatoires, aboutissant à une même lookup table. Plus ce nombre est grand, plus l'obfuscateur introduit de l'ambiguïté. La mesure de l'ambiguïté permet de rendre compte de l'espace des possibilités auquel l'attaquant fait face pour trouver les bonnes combinaisons clé / bijection utilisées lors de la génération de l'instance boîte blanche qu'il détient.

[9] Pour le lecteur souhaitant vérifier les calculs, rappelons qu'il y a $2^n!$ bijections $\mathbb{F}_2^n \to \mathbb{F}_2^n$, et parmi elles $2^n \prod_{i=0}^{n-1}(2^n - 2^i)$ bijections affine (et $\prod_{i=0}^{n-1}(2^n - 2^i)$ bijections linéaires).

**Définition 2.6.2** *(Ambiguïté boîte blanche [CEJvO02b]). L'ambiguïté boîte blanche est une métrique qui donne une estimation du nombre de constructions qui produisent exactement une certaine table (d'un type donné). Elle se définit comme le rapport de la diversité d'une table et du nombre de tables distinctes (de ce type).*

À titre d'exemple, la table 2.4 donne une approximation des mesures d'ambiguïté des quatre types de *lookup tables* qui sont utilisées dans l'implémentation boîte blanche de l'AES.

| type | ambiguity |
|---|---|
| type I | $(16!)^2 \times 20160^{32}$ |
| type II | |
| type III | $(16!)^2 \times 15!$ si les deux blocs de la matrice sont de rang nul |
| | $(16!)^2 \times 20160^2$ si les deux blocs de la matrice sont de rang maximal |
| type IV | $16! \times 16$ |

Tab. 2.4 – Ambiguïté des *lookup tables* de WB-AES [CEJvO02b]

### 2.6.3 Cryptanalyses de deux implémentations boîte blanche

#### Cryptanalyse de WB_DES

Rappelons que dans la version boîte blanche du DES, des mots des 96 bits transitent entre les *lookup tables*. Ces mots ou états internes sont représentés dans la figure 2.1 par la concaténation des blocs $L_{r-1} \in \mathbb{F}_2^{32}$, $X_{r-1} \in \mathbb{F}_2^{48}$ et $\overline{R_{r-1}} \in \mathbb{F}_2^{16}$ : $L_{r-1}||X_{r-1}||\overline{R_{r-1}}$.

L'état interne $L_{r-1}||X_{r-1}||\overline{R_{r-1}}$ du tour $r$ est subdivisé en mots de 8 bits et est donc représenté comme la concaténation de 12 mots binaires : $v_1^r||v_2^r||\ldots||v_{12}^r$. Chaque mot $v_i^r$ représente l'entrée encodée d'une T-Box $T_i^r$ qui, pour rappel, peut être de deux sortes : soit une T-Box non linéaire (embarquant une S-Box), soit une *bypass* T-Box.
L'attaque [WMGP07] travaille directement sur les vecteurs $v_i^r$ en leur additionnant des différences notée $\Delta v$ (ou de manière équivalente en substituant à $v_i$ une valeur $v_i'$. En effet, $v_i \oplus v_i' = \Delta v \Leftrightarrow v_i \oplus \Delta v = v_i \oplus v_i \oplus v_i' = v_i'$). Les vecteurs $v_i^r$ étant des représentations encodées des véritables entrées $f_i^r(v_i^r)$ des T-Boxes d'une part, et les encodages $f_i^r$ utilisés n'étant pas linéaires d'autre part, il n'est pas possible, a priori, de déduire de $\Delta v$ la différence effectivement appliquée à l'entrée $f_i^r(v_i^r)$. De ce fait l'attaque observe les propagations de ces différences sur les T-Boxes des tours suivants ($r+2$, $r+3$ et $r+4$).

La cryptanalyse de l'implémentation boîte blanche du DES [WMGP07] repose donc sur une propriété remarquable de la fonction de tour : les bits d'entrée n'affectent qu'une certaine quantité des bits de sortie de la fonction de tour : en particulier, un octet de l'état du tour $r$ n'affecte pas tous les octets de l'état du tour $r+1$. En analysant la propagation d'une différence $\Delta v = v \oplus v'$ sur l'entrée d'une boîte $T$ encodée (c'est-à-dire $g \circ T \circ f$) sur plusieurs tours, il est possible d'obtenir une information sur le comportement interne de cette différence (c'est-à-dire sur $f(\Delta v)$). Une fois identifiés un ensemble de différences : $\{\Delta v \,|\, f(\Delta v)$ correspond à un ou deux inversions de bit en entrée de $T\}$, il est possible de retrouver la clé embarquée dans l'implémentation boîte blanche.

L'attaque différentielle publiée dans [WMGP07] se déroule en trois étapes :

1. en premier lieu, distinguer les T-Boxes non-linéaires parmi les 12 T-Box ;
2. ensuite, découvrir en partie la permutation aléatoire $\pi_r$ appliquée aux T-Box non-linéaires, en utilisant une implémentation standard (boîte noire) de l'algorithme DES ;
3. enfin, extraire la clé $K_r$.

Nous présentons de manière plus détaillée les jalons de cette attaque dans la section C.2 de l'annexe C.

En résumé, cette attaque exploite la faiblesse de la fonction de tour du DES. De manière à parer une telle attaque, l'ultime recours pour améliorer cette conception semble être la randomisation des S-Boxes. Si une telle modification

de l'algorithme DES est clairement inacceptable (principalement parce que les S-Boxes vérifient des propriétés de sécurité particulières, comme cela a été établi dans la section 2.6.1), cela pourrait être une piste intéressante dans le contexte viral, même si la génération de telles fonctions vectorielles booléennes n'est pas une tâche triviale. En effet, un attaquant aurait à reconstruire les S-Boxes pour chaque nouvelle version de l'algorithme.
Examinons à présent la résistance en boîte blanche d'un algorithme de chiffrement plus puissant, l'AES, dont la fonction de tour est réputée immune à une telle attaque différentielle.

### Cryptanalyse de WB_AES

Comme le montre la description de l'AES obfusqué (section 2.5.2), un tour est désormais composé de deux *lookup tables*. La première réalise les opérations *AddRoundKey*, *ShiftRows* et *MixColumns* (le dernier tour est légèrement différent), tandis que la seconde, disposée entre les tours $r$ et $r+1$, inverse les bijections linéaires insérées à la fois

- avant l'encodage de sortie de la *lookup table* implémentant le tour $r$, ainsi que celle insérée
- après l'encodage d'entrée de la *lookup table* implémentant le tour $r+1$.

Une *lookup table* étant une représentation particulière d'une fonction vectorielle booléenne, il est tout à fait possible de composer les *lookup tables* entre elles lorsqu'elles sont compatibles. C'est ce que fait la cryptanalyse présentée dans [BGEC04] en s'intéressant aux quatre bijections qui, à quatre octets de l'état courant, font correspondre quatre octets de l'état suivant. Ces quatres bijections sont facilement identifiables en raison du réseau de XOR-boxes qui permet de calculer la sortie de l'opération MixColumns. Chacune de ces bijections est dénotée dans la suite par $R_j^r$, $j = 0, \ldots, 3$, $r = 1, \ldots, 9$ (c.f. Figure 2.2). Le coeur de $R_j^r$ est la concaténation de quatre T-boxes suivie de la multiplication par $MC$. Ce noyau est "protégé" en entrée et en sortie par des encodages (8 au total : 4 en entrée, 4 en sortie). Pour rappel, ces derniers sont constitués :

- en entrée, de la concaténation de deux encodages de quatre bits, suivie d'une bijection linéaire de huit bits,
- en sortie, d'une bijection linéaire de huit bits, suivie de la concaténation de deux encodages de quatre bits.

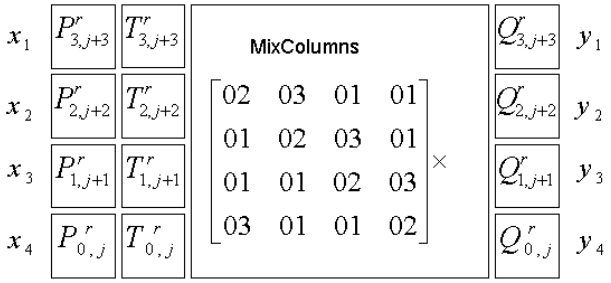



FIG. 2.2 – $R_j^r$, $j = 0, \ldots, 3$, $r = 1, \ldots, 9$

La cryptanalyse s'attache à retrouver les parasites protégeant les R-boxes. Pour cela, les auteurs [BGEC04] procèdent en plusieurs étapes :

1. Dans un premier temps, il s'agira de linéariser les P-box et Q-box, en recouvrant leur partie non linéaire.
2. Le fait de pouvoir ôter la partie non linéaire des P-box et Q-box, transforme donc ces dernières en des bijections affines inconnues. La seconde étape consiste donc à retrouver la composante linéaire ainsi que le vecteur de translation.
3. La découverte des bijections linéaires permet dans le même temps de récupérer les sous clés intégrées dans les S-boxes, mais dans un ordre inconnu. La dernière étape consiste donc à exploiter les contraintes pesant sur ces octets pour les classer dans le bon ordre. Ces contraintes sont issues de l'algorithme de cadencement.

Nous présentons de manière plus détaillée les jalons de cette attaque dans la section C.3 de l'annexe C.

En résumé, il semble difficile de dissimuler la structure algébrique de l'AES en utilisant uniquement des bijections non linéaires encodées. Pour parer cette attaque, une idée est d'introduire une opération de diffusion linéaire

imédiatemment après le premier encodage (notons ce nouveau réseau de *lookup tables* $32 \times 8$ $D_1^{-1}$), une opération $D_2$ (résau de *lookup tables* $8 \times 32$) étant introduite immédiatement après l'étape de substitution. On peut espérer que le bruit introduit par ces permutations aléatoires rendent difficile la seconde étape de l'attaque (se situant après linéarisation des encodages).

### 2.6.4 Polymorphisme

Nous présentons ici brièvement les avantages et inconvénients liés à l'utilisation de lookup tables. Rappelons que le polymorphisme peut être formalisé comme production d'une grammaire générative [Fil07b]. Plus la grammaire est irrégulière, plus il est difficile de dériver un automate à partir de la grammaire qui soit capable de détecter la fonction de chiffrement.

D'un point de vue théorique [ZZ04], le noyau d'un virus polymorphe est constitué d'une condition de déclenchement de l'infection $I(d, p)$, une fonction charge finale $D(d, p)$, la condition de déclenchement de la charge finale correspondante $T(d, p)$ et une fonction de sélection $S(p)$ des programmes cibles à infecter. La dernière fonction est en charge de la mutation du code.
Les virus métamorphes diffèrent des virus polymorphes car si les formes polymorphes d'un virus partagent le même noyau, ce n'est pas le cas des formes métamorphes. Par l'utilisation de grammaires formelles génératives [Fil07b], il est possible de donner une définition plus pratique de la notion de métamorphisme : Soit $G = (N, T, S, R)$ une grammaire formelle, où $N$ est un ensemble de symboles non terminaux, $T$ est un alphabet de symboles terminaux, $S \in N$ est le symbole start et $R$ est un système de production (ou système de ré-écriture ou semi-Thue) sur $(N \cup T)^*$, c'est-à-dire un ensemble de règles produisant le langage $L(G)$. Nous définissons $G' = (N', T', G, R')$ où l'alphabet de symboles terminaux $T'$ est un ensemble de grammaires formelles, et $R'$ est un ensemble de règles de production sur $(N' \cup T')^*$.

**Définition 2.6.3** *(Virus métamorphe [Fil07b]). Un virus métamorphe est un virus dont le moteur de mutation est décrit par une grammaire dont les mots sont eux mêmes un ensemble de productions relativement à une grammaire. Un virus métamorphe est donc décrit par $G'$ et chacune de ses formes mutées est un mot de $L(L(G'))$.*

Donc d'une forme métamorphe à une autre, le noyau viral est modifié : le virus mute et change en même temps les règles de mutation.
Concernant la complexité de la détection des techniques de mutation, plusieurs résultats théoriques ont déjà été établis :

- la détection des virus polymorphes de taille bornée est un problème NP-complet [Spi03] ;
- l'ensemble des virus polymorphes ayant un nombre infini de formes est un ensemble $\Sigma_3$-complet [ZZZ05] ;
- une technique de mutation de code embarquant le problème du mot - qui est connu pour être indécidable relativement à un système semi-Thue - conduit à des virus métamorphes dont la détection est indécidable [Fil07b]. Les règles de production du moteur PBMOT changent de mutation en mutation et ce dernier est spécialement conçu pour embarquer le problème du mot (relativement à un système semi-Thue).

Le cadre formel étant posé, la principale question qui se pose dans notre contexte est de prouver que l'implémentation boîte blanche est adaptée pour entraver la détection antivirale basée sur des séquences, en utilisant un moteur de mutation. Parce que les données de l'implémentation sont diversifiées par utilisation de bijections aléatoires, seul le traitement du code doit annuler autant que possible tout élément fixe qui pourrait constituer un motif de détection potentiel (syntaxique ou structurel, comme le graphe CFG par exemple). Intuitivement, puisque l'ensemble des instructions est très petit et correspond uniquement aux instructions nécessaires au parcours d'une table, les règles de production peuvent associer de telles instructions à n'importe quel morceau de code. Des recherches supplémentaires sont nécessaires pour valider cette hypothèse. En outre, plusieurs comportements peuvent représenter un invariant utilisable par un antivirus, comme le parcours linéaire de *lookup tables*.

## 2.7 Conclusion

Nous avons présenté dans ce chapitre une utilisation inédite de la cryptographie boîte blanche dans le contexte viral. Les mécanismes cryptographiques WBAC autorisent :

- une gestion originale des clés de chiffrement, en embarquant les clés dans l'implémentation par une évaluation partielle vis-à-vis de la clé. Ce mode de gestion des clés définit une alternative originale à la génération environnementale des clés de chiffrement (où la clé n'est pas embarqué dans le corps du programme viral mais générée dynamiquement à partir d'informations de déclenchement présentes dans l'environnement du virus) ou à l'utilisation d'une infrastructure de gestion de clé asymétrique (où seule la composante publique de la bi-clé est stockée dans le corps du virus). Ce mécanisme propose un compromis entre chiffrement symétrique et asymétrique, par asymétrisation de l'implémentation d'un algorithme de chiffrement par blocs. ;
- une diversification importante des implémentations, par intégration de bijections aléatoires (utilisées pour encoder les entrées/sorties des *lookup tables* ou insérer une étape de diffusion supplémentaire par le biais des *mixing bijections*). En plus d'assurer une sécurité locale, cette randomisation des implémentations permet de générer de nombreuses évaluations partielles de l'algorithme de chiffrement, pour une même mise à la clé. En outre, l'implémentation de l'algorithme de chiffrement ne comportant plus d'opérations arithmétiques (elle ne comporte plus que les opérations permettant au flot de données de transiter par un réseau de *lookup tables*), il est plus aisé de générer des instances polymorphes de l'algorithme par l'utilisation d'un moteur de mutation (le sous-ensemble du jeu d'instructions du CPU utilisé par une telle implémentation est réduit à des instructions de manipulations élémentaires de la mémoire. Ces opérations sont donc facilement diversifiables).
  Dans le contexte viral, ces propriétés sont les bienvenues. En particulier, la transformation d'obfuscation correspondante pourrait être appliquée à la cryptographie asymétrique ou aux fonctions de hachage, de manière à accroître le niveau de polymorphisme des fonctions de chiffrement et de génération environnementale des clés ;
- une résistance notable à la cryptanalyse en contexte WBAC : même si les mécanismes proposés aujourd'hui ne sont pas encore aussi résistants en boîte blanche qu'en boîte noire, les premiers résultats sont encourageants. Les algorithmes de chiffrement obtenus sont étonnement difficiles à casser. S'ils n'offrent pas encore un niveau de sécurité suffisant pour des applications critiques de gestion numérique des droits, leur utilisation potentielle dans un contexte viral pose d'ores et déjà un problème que la recherche anti-virale doit prendre en considération. Si les chaînes de compilation spécialisées dans la protection logicielle ne sont pas encore distribuées largement dans le commerce, cet état de fait pourrait changer dans les années à venir. De la même manière que les applications virales ont utilisé les packers spécialisés pour augmenter leur résistance face à la rétro-ingénierie, on peut imaginer qu'elles utilisent des solutions plus complètes et intégrant des mécanismes cryptographiques boîte blanche, leur permettant d'assurer une protection en profondeur.

La résistance en boîte blanche des mécanismes cryptographiques reste limitée en comparaison de la résistance en boîte noire des algorithmes de chiffrements itératifs par blocs à patir desquels ils ont été conçus.

Nous avons conjecturé qu'un code viral comportant uniquement les instructions CPU requises pour parcourir des tables, ces dernières étant complètement recalculées à chaque réplication du code viral, est plus facile à remodeler qu'un code contenant des données statiques immuables et riche en instruction arithmétique. Nous pouvons observer que pour être en mesure d'échapper à la détection d'un programme antiviral, une attention toute particulière doit être portée au moteur de mutation. Il est également nécessaire de renforcer la robustesse de la primitive cryptographique boîte blanche par d'autres mécanismes de sécurité. En particulier, d'autres mécanismes de protection logicielle doivent empêcher un attaquant d'extraire le code implémentant la fonction de chiffrement, qui pourrait être utilisée comme une clé. Dans la suite logicielle CSS, ces hypothèses de sécurité sont couvertes par l'utilisation d'une chaîne de compilation spécialisée, rendant possible l'intégration de protections supplémentaires à chaque étape de la compilation, de manière à assurer une protection en profondeur de l'application.

Il serait intéressant de spécifier un schéma de chiffrement symétrique spécialement pensé pour être résistant en boîte blanche, tout en assurant une résistance suffisante en boîte noire. Il est possible qu'un schéma itéré ne soit pas un bon choix de conception.
Nous pensons étudier les possibilités offertes par les schéma de chiffrement par flot. Nous souhaitons également étudier les possibilités offertes par les grammaires génératrices pour diversifier le code d'une implémentation boîte blanche.

Plusieurs tâches additionnelles pourraient être effectuées pour donner une vue d'ensemble plus complète de cette technologie :

- examiner la robustesse de l'implémentation boîte blanche de l'algorithme AES avec des tailles de clés égales à 192 et 256, renforcée par les bijections linéaires supplémentaires décrites dans la section 2.6.3 et une éventuelle

randomisation des boîtes de substitution ;
- étudier l'implémentation d'un moteur de mutation fondée sur l'utilisation des grammaires formelles, spécialement conçu pour exploiter la structure de l'implémentation boîte blanche.

Concernant la randomisation des S-boxes, je présente en annexe B un algorithme de génération aléatoire partant de propriétés de la représentation sous forme ANF (*Algebraïc Normal Form*) d'une fonction booléenne (une fonction booléenne peut être représentée de manière équivalente comme une table de vérité, un spectre de Fourier ou un polynôme multivarié appelé ANF). L'idée principale d'un tel algorithme de génération aléatoire est d'imposer des contraintes aux propriétés algébriques des monômes multivariés d'une fonction booléenne, de manière à garantir une haute non linéarité (comparable aux résultats obtenus par utilisation des méthodes déterministes connues).

Concernant le re-design de parties des algorithmes DES et AES pour atteindre le critère de diversité, des travaux ont déjà été effectués dans ce domaine de recherche, principalement dans les contextes BBAC et GBAC :
- dans le cas de l'AES, on peut citer notamment les *dual ciphers* [BB02], permettant de diversifier les implémentations de l'AES tout en maintenant un niveau de sécurité équivalent ;,
- dans le cas du DES, on peut citer la proposition de design FDE (*Fast Data Encryption* [SHG04]), qui a une structure très comparable au DES et vient avec un algorithme pour construire des S-boxes robustes.

Une démarche plus ambitieuse serait de reprendre du début la conception d'un algorithme de chiffrement spécialement conçu pour être résistant dans un contexte boîte blanche, c'est-à-dire ne correspondant pas par exemple à l'implémentation boîte blanche d'un algorithme de chiffrement itératif par blocs connu pour sa résistance en boîte noire - le paradigme de la fonction de tour itérée n'est peut-être pas une solution dans le contexte WBAC, dans la mesure où le cryptanalyste peut travailler sur un nombre arbitrairement réduit de tours.

À défaut de proposer un nouveau schéma de conception d'algorithme de chiffrement adapté au contexte WBAC, ce chapitre vise à apporter une première contribution dans l'élaboration d'un schéma de chiffrement symétrique fondé sur la génération aléatoire de S-Boxes et adapté au contexte viral. L'hypothèse de S-Boxes fixes et connues de l'attaquant est aujourd'hui une hypotèse trop forte dans le contexte WBAC. Je pense que l'utilisation de la cryptographie boîte blanche dans ces conditions (randomisation des boîtes de substitution) représente une technologie susceptible d'être utilisée par un virus pour résister à l'analyse dynamique.

Nous avons cependant observé qu'utilisés seuls, les mécanismes cryptographiques adaptés au contexte WBAC ne sont pas suffisants pour protéger un programme viral contre la rétro-ingénierie. Ces derniers permettent d'augmenter la diversification de leur implémentation, de lier un programme viral à un environnement d'exécution cible ou à un serveur du réseau ; mais ces mécanismes doivent eux-même être protégés. C'est l'utilisation conjointe de mécanismes cryptographiques et non-cryptographiques et leur agencement savant qui va permettre d'augmenter le niveau de résistance du programme protégé face à la rétro-ingénierie.
Nous complétons au chapitre suivant notre panorama de la menace virale par l'étude des mécanismes non cryptographiques susceptibles d'être utilisés par un virus pour asseoir la robustesse des mécanismes cryptographiques qu'il implémente et parer la rétro-ingénierie.

Chapitre 3

# Protection logicielle par embarquement de problèmes difficiles

Ce chapitre introduit les aspects techniques et théoriques liés à la protection et à la rétro-ingénierie logicielle, au regard de la théorie de la complexité et de l'optimisation des programmes. Il complète le panorama de la menace virale établi au chapitre 2 par l'étude et l'évaluation des mécanismes non cryptographiques d'un produit de protection logicielle représentatif de l'état de l'art dans ce domaine. L'objectif de ce chapitre est tout d'abord de présenter quelques unes des briques fondamentales (non cryptographiques[1]) constituant un système de protection logicielle et une manière de les agencer, à travers l'étude d'une solution phare du marché civil. Nous présentons ensuite un état de l'art des méthodes de rétro-ingénierie, puis une démarche, des critères et des outils pour évaluer la robustesse d'une solution de protection logicielle.
Nous adoptons donc le point de vue de l'attaquant, de manière à compléter l'analyse de la menace indispensable à l'énoncé de la problématique de sécurité (spécification du besoin de sécurité) liée à l'évaluation d'un produit antivirus.

## 3.1 Introduction

La motivation première de ce chapitre est de complèter le panorama de la menace virale établi au chapitre 2 par l'étude et l'évaluation des mécanismes non cryptographiques d'un produit de protection logicielle représentatif de l'état de l'art dans ce domaine. Si les concepteurs de programmes malicieux utilisent volontier les packers d'exécutables, ils n'utilisent pas encore d'outils plus évolués, comme les chaînes de compilation spécialisées, dont la distribution reste encore très discrétionnaire. Nous avons choisi d'analyser la menace potentielle que représenterait l'utilisation de ce type d'outils spécialisés par les auteurs de virus.
Ce chapitre a également vocation à introduire les concepts clés liés à la protection et à la rétro-ingénierie logicielle, tant au niveau technique qu'au niveau de la théorie de la complexité et de l'optimisation des programmes.
Nous avons choisi dans ce chapitre d'adopter le point de vue de l'attaquant et de définir des critères, une démarche et de proposer des outils pour évaluer la résistance d'un virus à la rétro-ingénierie. Ce point de vue est utile à la fois pour spécifier le besoin de sécurité d'un produit antivirus au regard de la menace virale que représenterai l'utilisation d'outils de protection plus évolués ; et pour estimer l'efficacité des mesures de sécurité implémentées par un produit antivirus au regard de cet ensemble d'exigences. Nous reviendrons sur ce point au chapitre 8.
La conception et l'implémentation de logiciels résistants à la rétro-ingénierie est un problème crucial pour de nombreuses applications, particulièrement lorsqu'il s'agit de protéger les algorithmes propriétaires implémentés par le logiciel et/ou de protéger la fonction de contrôle de droits conditionnant l'accès à tout ou partie de ses fonctionnalités.
Lorsque l'application à protéger ne peut pas asseoir sa sécurité sur l'utilisation d'un composant matériel de sécurité, ou d'un serveur du réseau, nous devons faire l'hypothèse d'un attaquant capable d'exécuter l'application dans un

[1] Nous avons traité dans le chapitre 2 la problématique de la conception cryptographique, dans un contexte boîte blanche, à travers la présentation de modes de gestion des clés et de primitives de chiffrement adaptés au contexte viral.

environnement qu'il contrôle parfaitement. Nous avons vu au chapitre 2 que le modèle de l'attaquant correspondant à ce contexte, que nous avons appelé WBAC (*White Box Attack Context*), impose une implémentation logicielle particulière des primitives cryptographiques classiques.
Dans ce contexte, la protection d'un programme repose sur des mécanismes couvrant différents objectifs de sécurité, dont la capacité à contrôler, en plusieurs points de son exécution, l'intégrité de son code, de ses données critiques et de son contexte d'exécution ; la capacité à garantir la confidentialité des algorithmes propriétaires, la diversification de ses instances, l'ancrage du logiciel à une plate-forme cible d'exécution personnalisée, etc.
Le marché civil offre plusieurs systèmes de protection logicielle *sur étagère*, permettant en principe de couvrir ces objectifs de sécurité. Ces systèmes permettent l'application de mécanismes de protection élémentaires au niveau du code source [Clo07] ou au niveau du code exécutable [SRT08, Asp08]. Dans le premier cas, il s'agit de chaînes de compilation spécialisées (autorisant l'application des protections et des transformations d'obfuscation au niveau du code source) ; dans le second cas, il s'agit d'un loader de protection (qui ne va déchiffrer tout ou partie de l'exécutable qu'après avoir contrôlé les droits conditionnant l'accès à tout ou partie des fonctionnalités de l'exécutable protégé).
Nous voyons donc que nos objectifs de sécurité peuvent théoriquement être couverts par l'utilisation conjointe d'une *chaîne de compilation spécialisée* et d'un *loader de protection*.
Cependant, l'utilisation et l'évaluation de ces solutions de protection logicielle n'est pas aisée. Dans quelle mesure la protection mise en oeuvre est-elle effectivement résistante à la rétro-ingénierie ?

Remarquons qu'il y a une différence notable entre la rétro-ingénierie manuelle, humaine, correspondant à l'ensemble des actions mises en oeuvre par un analyste de sécurité sur le programme protégé pour en comprendre le fonctionnement ; et son automatisation par un outil. Dans le contexte de la détection virale, la conception d'un moteur de détection capable de reconnaître tous les virus appartenant à une même souche virale métamorphe suppose que l'on est capable d'inverser automatiquement les transformations d'obfuscation implémentées par le moteur de mutation virale. Nous devons en outre supposer que le temps et l'espace requis par l'analyse est raisonnable. En théorie, le problème de la détection statique des virus métamorphes est prouvé **NP**-complet [BM08] dès lors que le moteur de mutation embarque un des problèmes difficiles de la théorie de l'optimisation. Nous voyons donc que le problème de la détection des virus métamorphes et celui de la conception d'un outil automatique de désobfuscation sont deux formulations du même problème.

À l'issu de ce chapitre et du chapitre 2, nous disposons d'une vision assez précise des caractéristiques, en terme d'architecture, des solutions de protection logicielle du marché civil et des mécanismes fondamentaux qu'elles implémentent. Nous avons en particulier une meilleure compréhension des problèmes théoriques et pratiques rencontrés lors d'une tentative de lever la protection d'une application, ainsi que des principales méthodes d'analyse pouvant être mises en oeuvre par un attaquant. Nous proposons une démarche et des outils permettant de mener à bien une analyse de robustesse des mécanismes, sur la base de critères théoriques et empiriques.
Une telle approche peut être utilisée par un évaluateur, pour mesurer le niveau de confiance qu'il peut porter à une solution ; par le concepteur de solution, afin d'en améliorer l'efficacité ; ou par l'utilisateur, afin d'intégrer une ou plusieurs solutions et éventuellement développer des protections supplémentaires.
Dans le contexte de l'évaluation d'un produit antivirus, nous prenons mesure de la difficulté que peut représenter la rétro-ingénierie de code viral et la conception d'un moteur de détection virale adapté à cette menace virale potentielle.

Une partie du contenu de ce chapitre a été publiée dans les actes de la conférence C&ESAR 2007 [DJ07], ainsi que dans le journal MISC en 2005 [Jos05].

Le reste de ce chapitre est organisé de la manière suivante : nous présentons l'architecture générale d'une solution de protection logicielle du marché civil représentative de l'état de l'art dans ce domaine et quelques un des mécanismes sur lesquels elle s'appuie dans la section 3.2. Nous présentons les méthodes d'attaque dans la section 3.3. Nous présentons enfin une démarche, des critères et des outils pour évaluer la robustesse d'un système de protection logicielle dans la section 3.4.

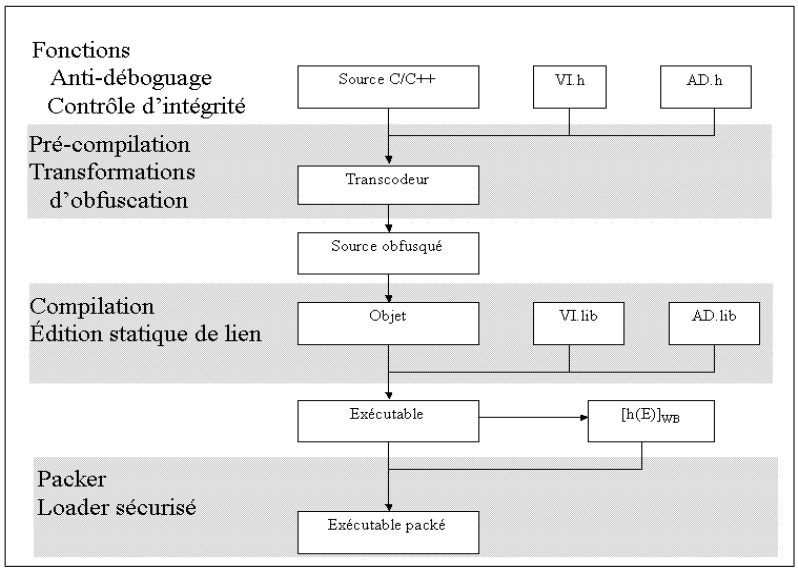



FIG. 3.1 – Synoptique de la chaîne de compilation CSS

## 3.2 Solution et mécanismes de protection logicielle

### 3.2.1 Architecture et limites des solutions

Le marché civil offre différents types de solutions de protection purement logicielle[2] :

- les logiciels de protection d'exécutable, comme Armadillo [SRT08] ou AsProtect [Asp08] sont aussi appelés loaders de protection ou packers (le loader de protection a pour fonction de déchiffrer tout ou partie de l'exécutable après avoir contrôlé les droits conditionnant l'accès à tout ou partie des fonctionnalités de l'exécutable protégé) ;
- les chaînes de compilation spécialisées (Cloakware CSS, Hitachi, IBM EMMS[3], Intel) dédiées à l'application de transformations d'obfuscation et de diversification du code.

Ces dernières paraissent les plus pertinentes, dans la mesure où elles autorisent l'intégration des mécanismes de protection à différentes étapes de la compilation du logiciel, assurant ainsi une protection en profondeur. On trouve dans cette catégorie de produits la solution CSS (Cloakware Security Suite [Clo07]), qui est à plusieurs égards représentative de l'état de l'art dans ce domaine, notamment en raison de la richesse des mécanismes qu'elle met en oeuvre. Elle propose en particulier des primitives cryptographiques adaptées au contexte WBAC, décrites au chapitre 2. La figure 3.1 représente le synoptique de fonctionnement de cette chaîne de compilation, qui se compose pour l'essentiel d'un pré-compilateur, d'un packer et de bibliothèques spécialisées.
Le développeur dispose de deux bibliothèques pour protéger son application : la première regroupe les fonctions de contrôle d'intégrité, et la seconde regroupe les fonctions anti-déboguage. L'appel à ces fonctions est à la discrétion du développeur.
Un pré-compilateur, nommé *transcodeur*, applique plusieurs transformations d'obfuscation se fondant principalement sur le codage/décodage dynamique des variables et sur des transformations d'obfuscation intra-procédurales. Ce *transcodeur* implémente également un mécanisme de diversification des transformations, paramétrable par le développeur via une mise à la clé.
Le code source obfusqué est ensuite compilé. L'édition de liens entre l'objet et les bibliothèques du produit est une édition statique de liens. Une ou plusieurs empreintes numériques portant sur tout ou une partie de l'exécutable résultant sont calculées. Ces empreintes sont ensuite chiffrées par utilisation d'un algorithme de chiffrement boîte blanche, le résultat étant concaténé avec l'exécutable. L'exécutable est ensuite packé (encapsulé) au moyen d'une application appelée *Secure Loader*. Cette ultime protection implémente effectivement une partie des fonctionnalités

[2] Certains systèmes de protection des logiciels se fondent sur l'utilisation d'un composant matériel de sécurité
[3] Electronic Media Management System

d'un loader d'exécutable. Elle est chargée de sécuriser le chargement en mémoire de l'application protégée. Elle procède en particulier à son déchiffrement en boîte blanche.
Nous présentons dans la section suivante quelques uns des mécanismes implémentés dans les solutions de protection. Nous étudions donc plus en détail les rouages internes de ces solutions de protection.

### 3.2.2 Mécanismes de protection

Nous avons vu dans la section précédente que la robustesse des fonctions de sécurité proposées par une solution de protection logicielle repose sur l'agencement savant de mécanismes élémentaires de protection logicielle. Nous avons présenté au chapitre 2 des mécanismes représentatifs des approches suivantes :

- la protection en confidentialité des clés et en intégrité des hachés par utilisation de la cryptographie boîte blanche (section 2.3.3) ;
- la protection en confidentialité des clés et l'ancrage logiciel, par génération environnementale de clé et de code (section 2.3.2).

Nous présentons dans cette section une sélection de mécanismes de protection représentatifs des approches suivantes :

- la protection contre l'analyse statique et dynamique, par application de transformations d'obfuscation et diversification du code (section 3.2.2) ;
- la protection contre l'instrumentation du contexte d'exécution, par contrôle de l'intégrité de l'environnement d'exécution (section 3.2.2) ;
- la protection contre l'analyse statique et dynamique par utilisation de code auto-modifiable (section 3.2.2).

#### Techniques d'obfuscation de code résistantes à l'analyse statique et dynamique

Les techniques d'obfuscation de code visent à augmenter la complexité du programme pour ralentir la compréhension de celui-ci par un attaquant. Nous proposerons une définition plus formelle de cette notion au paragraphe 3.4.1. Il existe une grande similarité entre ces techniques, destinées à dégrader l'intelligibilité d'un programme, et les méthodes de diversification de code, visant à générer des versions polymorphes d'un programme, c'est à dire des programmes possédant les mêmes fonctionnalités que le programme d'origine, mais dont le code est sensiblement différent.
L'analyse de ces protections se fonde également sur des méthodes similaires, visant à défaire les transformations d'obfuscation ou à normaliser le programme protégé, par application de transformations successives sur une représentation intermédiaire du code [CKJ+05, BKM08], de manière à converger vers une forme *canonique* du programme, proche du programme d'origine.
Les techniques d'obfuscation doivent être résistantes à l'analyse statique, et en particulier à toutes les transformations d'optimisation visant à réduire la taille du programme [ACD+06] tout en conservant sa sémantique. Les transformations implémentées par un moteur de polymorphisme doivent de plus être résistante aux méthodes d'analyse différentielle de binaire [DR05, SAB07a, SAB07b], lorsque l'attaquant dispose de plusieurs instances polymorphes du programme protégé.

Parmi les techniques usuelles d'obfuscation, on trouve notamment : l'utilisation de *junk code*, la destructuration du découpage fonctionnel du code, la sur-utilisation d'instructions de saut à des adresses calculées lors de l'exécution et l'utilisation de prédicats opaques.
Un générateur de *junk code* transforme le code initial en un code final équivalent, en ajoutant des instructions supplémentaires dans le code initial. Ces instructions supplémentaires forment le *junk code*. Elles ne doivent pas changer la sémantique du programme. Le *junk code* peut être de différentes natures : il est soit mort (il n'est jamais exécuté), soit vivant. Le *junk code* vivant peut être composé d'instructions neutres (par exemple `add eax, ebx` suivi de `sub eax, ebx`) ou d'instructions utilisant des variables mortes (par exemple `add eax, ebx` sachant que le registre `eax` n'est pas utilisé).
Un code non obfusqué possède une certaine structure : un découpage en procédures et fonctions. Cette structure est très importante pour la compréhension du programme. La déstructuration du découpage fonctionnel du code vise à masquer la structure du code d'origine. L'ensemble des fonctions et des appels à ces fonctions est caché en le remplaçant par un autre ensemble de fonctions fabriquées aléatoirement. Les appels à ces nouvelles fonctions sont mis à jour de manière à ne pas modifier la sémantique du programme. Parmi les transformations qu'il est possible d'appliquer pour masquer la structure du programme, on trouve notamment :

- l'*inlining* (remplacement de l'appel à une fonction par le corps de la fonction),

- la fusion de fonctions (transformation permettant d'obtenir une fonction à partir de plusieurs) et
- la distribution de fonctions (transformation inverse permettant d'éclater une fonction en plusieurs).

La sur-utilisation d'instructions de saut à des adresses variables vise à entraver l'analyse statique du code. En effet, pour chaque `jmp eax` l'attaquant devra trouver la valeur contenue dans le registre `eax` lors de l'exécution du saut. Un prédicat opaque est une séquence d'instructions ajoutées au code initial. Il n'en modifie pas la sémantique et permet de lier artificiellement des sections de code indépendantes. Un prédicat opaque est un prédicat dont la valeur est connue lors de la compilation, mais qui est difficile à établir après compilation, par analyse du programme. Un prédicat opaque peut être soit toujours vrai, soit toujours faux, soit indéterminé. Afin de ne pas être facilement supprimé, un prédicat opaque est basé sur une propriété que l'on appellera son fondement. Ce fondement peut être par exemple une comparaison de pointeurs de l'application ou de pointeurs ajoutés, une comparaison de variables de l'application (sémantique) ou une propriété mathématique, dont la valeur de vérité est difficile à évaluer par une analyse du flot de données.

Ces techniques élémentaires d'obfuscation sont le plus souvent utilisées conjointement par les chaînes de compilation spécialisées. En particulier, le transcodeur de la solution CSS permet d'appliquer des transformations d'obfuscation mettant en oeuvre la plupart de ces techniques.

#### Contrôle de l'intégrité de l'environnement d'exécution

Ces techniques ont pour objectif de détecter les modifications apportées à l'environnement système d'exécution du logiciel, par rapport à l'environnement cible.
Un premier jeu de techniques vise à détecter les modifications effectuées par l'attaquant au niveau des API du système, afin de pouvoir récupérer de l'information sur un programme. Ces techniques d'instrumentation des API (*API Hooking*) peuvent être détectées en contrôlant l'intégrité des fonctions d'API critiques utilisées par le logiciel pour implémenter ses fonctions de sécurité.
Un second jeu de techniques vise à détecter l'exécution dans un mode correspondant à celui d'une analyse dynamique. Une analyse dynamique peut être menée au moyen d'un débogueur ou au moyen d'un émulateur.
Une analyse dynamique au moyen d'un débogueur peut être détectée par contrôle d'intégrité du code de l'exécutable ou par examen des modifications induites par le déboguage sur le contenu des registres du CPU.
Une analyse dynamique au moyen d'un émulateur est plus difficile à détecter. En particulier, si l'émulateur développé par l'attaquant est de bonne qualité, le contrôle d'intégrité du code ou l'examen des changements d'états du CPU virtuel ne seront d'aucune utilité. L'idée est alors d'appliquer des stimuli complexes au matériel émulé, et de contrôler la valeur de retour.

#### Code auto-modifiable

On parle de code auto-modifiable pour désigner un programme qui modifie dynamiquement sa section de code en mémoire. Les packers d'exécutable (par exemple le Secure Loader de la solution CSS) implémentent ce type de mécanisme pour supprimer ou déplacer une partie de l'information nécessaire à l'exécution d'un programme ou pour ne déchiffrer à chaque instant que certaines portions du programme protégé en mémoire. Ce mécanisme permet d'entraver à la fois l'analyse statique et la reconstruction de l'exécutable à partir de son image mémoire, lors d'une analyse dynamique.

Un packer est un programme qui prend en entrée un exécutable, le chiffre ou le compresse, puis forge un nouvel exécutable composé d'une routine de déchiffrement/décompression, que nous appellerons fonction de restauration et d'un ou plusieurs blocs de données. La fonction de restauration implémente une partie des fonctionnalités du *loader* d'exécutable du système d'exploitation. Au moment de l'exécution, l'exécutable d'origine est dynamiquement reconstruit en mémoire puis exécuté.

Rappelons que le chargement d'un exécutable en mémoire par le système d'exploitation est effectué par un module appelé loader. Au vu des renseignements consignés dans l'entête du programme, le loader effectue le chargement en mémoire du code et des données du programme et une résolution des liens indispensables à son exécution, i.e. la localisation et le chargement en mémoire des bibliothèques partagées utilisées par le programme puis la résolution des symboles en adresses. Une fois ce travail fait sur le code chargé en mémoire, le loader rend la main au code du

programme au niveau de son point d'entrée, c'est à dire de la première instruction de celui-ci.

Un packer (ou loader de protection) implémente une partie du code de chargement, tâche habituellement effectuée par le chargeur du système hôte. Son rôle est d'assurer le déchiffrement et la décompression en mémoire des sections de l'exécutable et de charger les adresses des fonctions importées par l'application, avant de rendre la main à l'application protégée.

Les protections qui se contentent d'envelopper le programme et de l'extraire en une fois au début de l'exécution n'apportent qu'une protection très limitée contre la rétro-ingénierie dès qu'on a la possibilité d'exécuter le programme. En effet, il suffit alors d'exécuter le programme supposé protégé sous débogueur jusqu'à ce que l'enveloppe soit complètement enlevée et de sauvegarder le résultat : un programme non protégé.
Pour éviter cela, il est nécessaire que l'application protégée soit plus intimement associée à son enveloppe protectrice : plus précisément, il faut que celle-ci soit toujours active ou, à défaut, que l'application puisse y faire explicitement appel lors de certaines opérations jugées plus sensibles par le développeur de l'application.

Afin de rendre difficile l'analyse, les packers implémentent des mécanismes de détection afin de ralentir la progression dans le code de la protection par l'utilisation d'un débogueur. Ces techniques visent à détecter la présence d'un débogueur en mémoire ou l'exécution pas à pas du code de la protection. Parmi les méthodes de détection de l'utilisation d'un débogueur, on touve notamment :

- La mesure des temps d'exécution : le principe de cette technique de protection est de quantifier le temps d'exécution normal de certains blocs d'instructions et de le comparer aux durées relevées. Si le temps d'exécution constaté est supérieur au temps d'exécution normal, le programme est considéré comme étant tracé. Les horloges envisageables pour ces mesures sont : l'horloge de la machine, accessible par un appel système ; un compteur spécifique du processeur associé à l'horloge du processeur.
  À titre d'exemple, le programme suivant met en oeuvre la mesure des temps d'exécution par utilisation de l'instruction `rdtsc` :

```
      message1 db ":o)",0
      message2 db ":o[",0

      rdtsc                    ; Première mesure du temps
      push eax

      .....                    ; Code de l'application initiale
      .....

      rdtsc                    ; Deuxième mesure du temps
      sub eax, dword ptr [esp] ; Comparaison du delta avec la valeur seuil DELAI_MAXI
      cmp eax, DELAI_MAXI
      jb no_traced
      invoke MessageBoxA, 0, offset message2, offset message2, MB_ICONERROR
      db 0FFh, 0FFh

no_traced:
      .....                    ; Code de l'application initiale protégée contre
      .....                    ; l'analyse dynamique
```

  Le programme suivant met en oeuvre la mesure des temps d'exécution par utilisation de la fonction GetTickCount :

```
      .....
      call GetTickCount        ; Première mesure du temps
      push eax
      .....                    ; Code de l'application initiale
```

```
        .....

        call GetTickCount        ; Deuxième mesure du temps
        sub eax, dword ptr [esp] ; Comparaison du delta avec la valeur seuil DELAI_MAXI
        cmp eax, DELAI_MAXI
        jb no_traced

        invoke MessageBoxA, 0, offset message2, offset message2, MB_ICONERROR
        db 0FFh, 0FFh            ; Code levant une exception

no_traced:
        .....   Code de l'application initiale protégée contre
        .....   l'analyse dynamique
```

- Le contrôle d'intégrité du code. La manière classique de suspendre l'exécution du processus en cours d'analyse consiste à remplacer l'instruction ou on souhaite s'arrêter par une instruction qui provoque un déroutement vers le système d'exploitation qui donne le contrôle au débogueur. Les processeurs ont souvent une instruction ad hoc d'un seul octet (taille minimale d'une instruction machine. Sur les Intel x86, c'est « INT 3 » de code CC hexadécimal.) Calculer une somme de contrôle (ou un checksum) sur la zone du code du programme suffit pour détecter des modifications qui peuvent être imputés à des points d'arrêt. Cette méthode n'échoue qu'en cas de collisions dans l'algorithme de la somme de contrôle. Ce contrôle doit typiquement être fait en permanence lors de l'exécution.
- La mise à zéro des registres de déboguage. Cette technique consiste à mettre à zéro les registres de déboguage pour effacer les points d'arrêt matériels que l'attaquant peut poser au moyen d'un débogueur. Cette opération est effectuée par le gestionnaire d'exception de la protection, qui peut lorsqu'il prend la main, accéder en écriture au contexte de l'application protégée. Cette structure contient l'état de tous les registres à l'instant de l'exception :

```
CONTEXT STRUCT
        ContextFlags  DWORD      ?
        iDr0          DWORD      ?                        ; context+4
        iDr1          DWORD      ?                        ; context+8
        iDr2          DWORD      ?                        ; context+0Ch
        iDr3          DWORD      ?                        ; context+10h
        iDr6          DWORD      ?
        iDr7          DWORD      ?                        ; context+18h
        FloatSave     FLOATING_SAVE_AREA <>
        regGs         DWORD      ?
        regFs         DWORD      ?
        regEs         DWORD      ?
        regDs         DWORD      ?
        regEdi        DWORD      ?
        regEsi        DWORD      ?
        regEbx        DWORD      ?
        regEdx        DWORD      ?
        regEcx        DWORD      ?
        regEax        DWORD      ?
        regEbp        DWORD      ?
        regEip        DWORD      ?                        ; context+0B8h
        regCs         DWORD      ?
        regFlag       DWORD      ?
        regEsp        DWORD      ?
        regSs         DWORD      ?
        ExtendedRegisters db MAXIMUM_SUPPORTED_EXTENSION dup(?)
```

```
CONTEXT ENDS
```

Le programme d'exemple suivant illustre de fonctionnement de ce mécanisme de protection :

```
Code de l'application protégée

      push offset SEH1 ;  code d'installation de SEH1
      push fs:[0]
      mov fs:[0], esp
-------------------------------------------------------------------
      ; code protégé par SEH1
      ; lève une exception qui ne doit pas être interceptée
      ; par le débogueur (c'est donc SEH1 qui la traite).
      ; il est impossible de positionner un BPM dans la suite du code.
-------------------------------------------------------------------
      pop fs:[0]       ;       code de désinstallation de SEH1
      add esp, 4h
      ret

SEH1:
-------------------------------------------------------------------
      ; code de SEH1
      ...
      ; Mise à zéro des registres de déboguage DRx (structure CONTEXT) dans SEH1
      mov     eax, [esp+0Ch]             ; récupération de la structure CONTEXT
      add     dword ptr [eax+0B8h], 2
      mov     dword ptr [eax+4], 0
      mov     dword ptr [eax+8], 0
      mov     dword ptr [eax+0Ch], 0
      mov     dword ptr [eax+10h], 0
      mov     dword ptr [eax+18h], 155h;
      ...
      mov eax, 1             ; si eax=1, aller au gestionnaire suivant
      ret                    ; si eax==0, recharger le contexte et continuer l'exécution
```

- Auto-déboguage. Puisqu'un processus ne peut être contrôlé efficacement que par un seul débogueur, un moyen efficace de rendre difficile l'exécution sous débogueur est de faire en sorte que le programme protégé se fasse passer vis à vis du système d'exploitation pour un débogueur se contrôlant lui-même. Il peut ainsi détecter la présence d'un débogueur externe et s'il est présent compliquer l'analyse en entrant en compétition pour les ressources et évènements globaux comme les registres de support du processeur ou les déroutements liés aux points d'arrêts. Cette technique, combinée avec un autre mécanisme de protection de type enveloppe permet à celle-ci d'être toujours active et donc plus efficace. En particulier, lorsque l'une des couches de protection consiste à chiffrer le code machine, cette technique d'auto-déboguage permet de déchiffrer le programme par petit bloc et de re-chiffrer le code après exécution afin d'empêcher la construction d'une image complète de l'espace d'adressage.
- Détournement des interruptions. Cette technique consiste à détourner les interruptions correspondant aux points d'arrêt logiciels et matériels, afin de détecter et de perturber le fonctionnement d'un traceur ou d'un débogueur. Les interruptions, une fois détournées, pointent vers du code plus ou moins offensif. Ce type de protection se fonde sur la possibilité d'accéder en écriture à la table des interruptions. Les systèmes d'exploitation actuels interdisent cette possibilité en espace utilisateur. Il est alors nécessaire de passer par un pilote pour accéder à l'espace noyau et effectuer des modifications visant à détourner les interruptions.

Lors de l'analyse d'un programme packé, il est nécessaire de projeter l'image mémoire de l'application sur le système de fichier. Un packer implémente des mécanismes pour entraver cette étape de l'analyse. Ils permettent soit

d'empêcher la projection d'une image mémoire soit de faire en sorte que l'image mémoire soit inexploitable. Parmi ces méthodes de protection, on trouve notamment :

- La redirection des appels aux fonctions importées par l'application. Cette technique de protection consiste à détourner les appels aux fonctions importées par l'application. Au lieu d'appeler une fonction directement, l'application protégée appelle d'abord une fonction de protection qui se chargera de l'appel à la fonction importée.
  L'appel d'une fonction dont le code se situe à l'adresse `af` :

```
call dword ptr [a1] ; appel à une fonction externe que l'on veut masquer
-------------------------------------------------------------------
a1    af            ; table des imports
-------------------------------------------------------------------
af    push ebp      ; code de la fonction f (externe)
      mov ebp, esp
```

  devient, après redirection simple des appels aux fonctions importées par l'application :

```
call dword ptr [a1] ; appel à une fonction externe que l'on veut masquer
-------------------------------------------------------------------
a1    a2            ; table des imports
-------------------------------------------------------------------
a2    push af       ; fonction de redirection (chiffrée)
      ret
-------------------------------------------------------------------
af    push ebp      ; code de la fonction f
      mov ebp, esp
```

  Cette redirection permet de rendre la projection d'une image mémoire plus difficile à exploiter si et seulement s'il est possible d'empêcher la copie de l'enveloppe de protection en même temps que le reste de l'espace d'adressage. Dans le cas contraire, suivre la chaîne d'appel suffit à déterminer la routine effectivement appelée et il suffit pour normaliser le programme de remplacer l'appel initial par l'adresse trouvé en extrémité de chaîne d'appel. Cette normalisation est similaire à l'optimisation standard d'un compilateur éliminant les sauts vers des sauts.
- La redirection avec émulation partielle des imports. Afin de rendre moins directe la reconstruction de la table des imports, les protections logicielles implémentent le détournement de la table des imports en exécutant les premiers octets de la fonction importée dans sa propre routine. La protection implémente ce type d'émulation par copie des premiers octets du corps des fonctions importées. En effet, le début d'une routine est standard, il contient des instructions construisant un cadre de pile pour l'accès aux paramètres et variables locales, et cette construction utilise souvent des instructions spécifiques à cet usage. On appelle cette séquence d'instructions caractéristiques le prologue de procédure. Dans une émulation partielle, on trouve un prologue à une adresse qui n'est pas atteinte via une instruction d'appel de procédure mais qui contient un appel vers un fragment de code sans prologue de procédure. Remarquons que dans certaines protections, le fragment de code dupliqué est plus long que le simple prologue de procédure.
  Après redirection avec émulation partielle des imports, le programme d'exemple précédent devient :

```
call dword ptr [a1]
-------------------------------------------------------------------
a1    a2                ; table des imports
-------------------------------------------------------------------
a2    push ebp          ; fonction d'émulation partielle :
      mov ebp, esp      ; les x premiers octets de f sont stockés
      ...               ; dans la fonction d'émulation partielle
      jmp af+x
-------------------------------------------------------------------
af    push ebp ;        ; code de la fonction f
      mov ebp, esp
```

```
      ...
af+x
```

– Redirection avec émulation totale des imports. Dans le cas très particulier des fonctions sans paramètre ni effet de bord et à valeur constante dans un processus donné, une protection similaire à la protection précédente consiste à remplacer les appels à ces fonctions par des références à une variable. Cette variable est initialisée par un appel camouflé dans l'enveloppe de protection. Après redirection avec émulation totale des imports, le programme d'exemple précédent devient :

```
      call f                  ; code inclus dans le loader de protection
      mov [ebx-0ah], eax      ; sauvegarde du résultat de l'exécution de la fonction f
-------------------------------------------------------------------
      ...
-------------------------------------------------------------------
      push 0                  ; fonction d'émulation
      call g                  ; appel factice à la fonction g
      push dword ptr [a1]     ; autre leurre
      pop eax
      mov eax, [ebx-0ah]
      ret
```

– Altération de la section code. Cette méthode de protection consiste à modifier la section code de l'exécutable avant de la chiffrer. Cette modification rend l'application protégée dépendante de la protection. A aucun moment, les instructions originales ne sont remises dans l'application. Elles ont été tout simplement détruites, pour être ensuite émulées par la protection. Il est alors nécessaire, après projection d'une image mémoire du processus, de réparer la section code du programme. Cette altération peut se faire par le biais du gestionnaire d'exception de la protection : la protection remplace des blocs d'instructions par une exception. Son gestionnaire d'exception se charge d'émuler ces instructions. Lors d'un dump, le code de la protection n'est plus présent, et ne peut donc plus gérer les exceptions générées par l'application. Celle-ci devient inutilisable. Cette altération peut affecter les appels aux fonctions d'API, en redirigeant ces instructions vers la protection. Cette altération peut affecter le point d'entrée d'origine du programme. Les instructions du point d'entrée du programme sont placées dans le code de la protection et protégées par chiffrement. Les instructions correspondant au point d'entrée original sont exécutées par la protection puis effacées, afin de rendre toute projection mémoire non opérationnelle. Ces instructions sont d'abord recopiées dans le code de la protection, puis mises à zéro au niveau du programme, lors de la protection de celui-ci. À chaque exécution, les instructions protégées sont effacées avant l'appel du reste du programme.
– Virtualisation du code. Cette technique de protection peut être vue comme la conjonction du mécanisme d'altération de la section code, avec un moteur de diversification ou d'obfuscation de code, fondé sur un CPU logiciel. Plusieurs logiciels de protection d'exécutable proposent d'ores et déjà ce mécanisme (VMProtect, StarForce, Thémidia, SecuROM). Le principe de cette protection est le suivant : le développeur insère des marques dans son code, pour délimiter une portion critique de code ; après compilation, le logiciel de protection repère la zone à protéger dans le binaire, la désassemble puis traduit les instructions dans le jeu d'instruction d'un processeur logiciel (bytecode) ; le processeur logiciel, ajouté à l'exécutable au sein du loader de protection, se charge d'exécuter le bloc de bytecode.
À titre d'exemple, l'utilitaire de virtualisation de code (Code Virtualizer) inclus dans la solution Thémidia, remplace la zone de code à protéger par un appel à la routine d'initialisation de la machine virtuelle :

```
.text:00401111                db 0E9h ; jmp     loc_41A152
.text:00401112                db  3Ch ;
.text:00401113                db  90h ;
.text:00401114                db    1 ;
.text:00401115                db    0 ;
```

La routine d'initialisation de la machine virtuelle commence par pousser sur la pile l'adresse du bytecode

implémentant l'algorithme protégé :

```
.v_lizer:0041A152 loc_41A152:
.v_lizer:0041A152              push    offset dword_419E4D
.v_lizer:0041A157              jmp     loc_4152EC
```

La routine d'initialisation effectue ensuite un calcul de delta offset avant de débuter la boucle responsable du chargement, du décodage puis de l'exécution des instructions virtuelles.

```
.v_lizer:004152EC loc_4152EC:
.v_lizer:004152EC
.v_lizer:004152EC                 pusha
.v_lizer:004152ED                 pushf
.v_lizer:004152EE                 cld
.v_lizer:004152EF                 call    $+5
.v_lizer:004152F4
.v_lizer:004152F4 loc_4152F4:
.v_lizer:004152F4                 pop     edi
.v_lizer:004152F5                 sub     edi, offset loc_4152F4
.v_lizer:004152FB                 mov     eax, edi
.v_lizer:004152FD                 add     edi, offset dword_415000
.v_lizer:00415303                 cmp     eax, [edi+2Ch]
.v_lizer:00415306                 jnz     short loc_41530A
.v_lizer:00415308                 jmp     short loc_415340

...

.v_lizer:00415340 loc_415340:
.v_lizer:00415340
.v_lizer:00415340                 mov     esi, [esp+24h]
.v_lizer:00415344                 mov     ebx, esi
.v_lizer:00415346                 add     esi, eax
.v_lizer:00415348                 mov     ecx, 1
.v_lizer:0041534D
.v_lizer:0041534D loc_41534D:
.v_lizer:0041534D                 xor     eax, eax
.v_lizer:0041534F                 lock cmpxchg [edi+30h], ecx
.v_lizer:00415354                 jnz     short loc_41534D
.v_lizer:00415356
.v_lizer:00415356 loc_415356:
.v_lizer:00415356
.v_lizer:00415356                 lodsb
```

Ici, l'instruction `lodsb` charge le premier octet de bytecode (le registre `esi` pointe vers le bytecode, `ecx` vaut 1).
Le processeur logiciel possède une structure généralement facilement identifiable, quelle que soit l'architecture implémentée : un contexte virtuel ; un tableau de gestionnaires d'instructions virtuelles ; des primitives de décodage des instructions virtuelles ; une boucle responsable du chargement, du décodage puis de l'exécution de l'instruction virtuelle courante, qui se traduit notamment par un changement du contexte virtuel du processeur logiciel.
Deux niveaux de diversification peuvent intervenir :

1. le processeur virtuel est potentiellement différent à chaque instance de la protection. Sa complexité peut varier, dans la mesure où l'architecture abstraite simulée peut être plus ou moins complexe et éloignée de l'architecture du processeur hôte. Le contexte virtuel et le nombre d'instructions virtuelles peuvent

également varier. En particulier, le jeu d'instruction virtuel peut être à la fois très riche et très différent du jeu d'instruction du CPU hôte.

2. Les gestionnaires responsables du traitement des instructions peuvent avoir une implantation très différentes dans le jeu d'instruction du CPU hôte, par utilisation de transformations d'obfuscation.

C'est très probablement ce second niveau de diversification qui est le plus primordial. L'utilisation d'un processeur logiciel apparaît comme une méthode astucieuse pour l'implantation de transformations d'obfuscation en aval de la chaîne de compilation, par réécriture de code binaire.

- Dissimulation du processus. Cette technique de protection est destinée à empêcher la projection d'une image mémoire du processus en dissimulant son exécution aux yeux de l'outil de projection utilisé. Les techniques de furtivité mises en oeuvre consistent le plus souvent à modifier les DLL de MS-Windows et en particulier, les fonctions du système d'énumération des processus ou installer un driver pour obtenir les privilèges du mode noyau.
- Protection du processus. Cette technique de protection est destinée à empêcher la projection d'une image mémoire du processus en interdisant la lecture à tout autre processus que lui-même. Comme la technique précédente, on change les DLL du système mais les fonctions ciblées sont celles utilisées par les débogueurs et outils de projection.
- Altération de l'entête. Cette technique de protection est destinée à empêcher la projection d'une image mémoire du processus en indiquant une valeur erronée pour la taille de l'image mémoire dans l'entête de l'exécutable. Il s'agit de rendre le fichier acceptable par le système d'exploitation mais pas par les outils d'analyse.

Nous reviendrons sur ces mécanismes et leurs robustesse au chapitre 5. Nous avons présenté dans cette section une solution de protection logicielle représentative de l'état de l'art dans ce domaine. Nous avons également examiné quelques-uns des rouages fondamentaux de ce type de solution.

## 3.3 Introduction aux techniques et méthodes de rétro-ingénierie

Nous avons vu à la section précédente une sélection de mécanismes de protection représentatifs des approches suivantes :

- la protection contre l'analyse statique et dynamique, par application de transformations d'obfuscation et diversification du code (section 3.2.2) ;
- la protection contre l'instrumentation du contexte d'exécution, par contrôle de l'intégrité de l'environnement d'exécution (section 3.2.2) ;
- la protection contre l'analyse statique et dynamique par utilisation de code auto-modifiable (section 3.2.2).

Nous présentons dans cette section les principales techniques et méthodes pouvant être mises en oeuvre par un attaquant pour analyser un exécutable protégé et lever la protection :

- les méthodes d'analyse statique, dynamique et hybride statique/dynamique (section 3.3.1) ;
- les techniques d'instrumentation du contexte d'exécution (section 3.3.2) ;
- les méthodes d'unpacking d'exécutable et d'analyse de code auto-modifiable (section 3.3.3).

### 3.3.1 Méthodes d'analyse statique, dynamique, hybride

#### Analyse statique

L'analyse statique d'un exécutable vise à en obtenir le maximum d'information. Cette analyse comporte plusieurs étapes, chacune pouvant être naturellement entravée, compte tenu de la structure du jeu d'instruction CPU (non alignement des instructions) ou des limites intrinsèques de l'analyse statique (séparation du code et des données). Nous présentons maintenant les différentes étapes mises en oeuvre (et les difficultés rencontrées en pratique) par un outil d'analyse statique pour désassembler et forger le graphe PCG d'un programme.

**Analyse du flot de contrôle** L'analyse du flot de contrôle est une étape cruciale de l'analyse d'un programme. En outre, de nombreuse techniques d'analyse s'appuie sur le résultat de cette analyse. L'analyse du flot de contrôle comporte généralement deux étapes :

- une étape inter-procédurale qui vise à construire le graphe des appels entre procédures ;
- une étape intra-procédurale qui vise à identifier les blocs de base d'une procédure.

Globalement, l'objectif est de construire un graphe de graphes. Le premier graphe, appelé graphe PCG (*Program Control Graph*), ICFG (*Inter-Procedural Control Graph*) ou simplement graphe des appels, comporte autant de noeuds que de procédures. Les noeuds du graphe PCG sont appelés graphes CFG (*Control Flow Graph*).

**Définition 3.3.1** *(Graphe CFG). Représentation sous forme d'un graphe orienté $G = (N, E)$ des blocs de base d'une procédure. On ajoute parfois deux types de blocs de code : un (ou plusieurs) bloc(s) d'entrée dans la fonction ; un ou plusieurs blocs de sortie de la fonction, que l'on prolonge vers un bloc terminal unique.*

Les noeuds d'un graphe CFG sont les blocs de base d'une procédure, les arcs représentant les relations de branchement entre ces blocs d'instructions :

**Définition 3.3.2** *(Bloc de base). On définit un bloc de base comme une séquence maximale d'instruction dont la première est l'unique point d'entrée et la dernière est l'unique point de sortie.*

*La première instruction d'un bloc de base peut être :*
- *le point d'entrée d'une procédure ;*
- *la cible d'un branchement ;*
- *l'instruction qui suit immédiatement une instruction de branchement ou une instruction de retour.*

*Ces instructions sont appelées des leaders.*

À première vue, les blocs de base d'un graphe CFG peuvent être identifiés par un algorithme très simple fondé sur la sélection des *leaders* : pour déterminer les blocs de base constituants une procédure, on commence par identifier les leader puis, pour chaque *leader*, on inclut dans le bloc de base correspondant toutes les instructions du leader courant jusqu'au leader suivant (ou jusqu'à la fin de la procédure). Nous allons voir cependant qu'en pratique, le problème de l'extraction du graphe de flot d'un programme impose des approximations à l'outil d'analyse statique. En fait, le problème général de l'extraction du graphe de flot d'un programme n'est pas calculable. En effet, ce calcul est réductible au problème de l'arrêt [BKM08].

La première étape consiste à décoder les instructions, afin de localiser les instructions de branchement du programme. Les instructions de branchement inter-procédurales sont utilisées pour construire le graphe PCG (*Program Control Graph*) du programme, tandis que les instructions de branchement intra-procédurales sont utilisées pour forger les graphes CFG (*Control Flow Graph*) des procédures. Cette analyse peut être séquentielle ou récursive (les désassembleurs utilisent parfois conjointement les deux approches). Aucune de ces deux approches n'aboutit à un désassemblage et à une représentation exacte du programme : la première approche aboutit généralement à une sur-approximation conservative du graphe PCG d'origine, tandis que la seconde aboutit à une sous-approximation du graphe PCG d'origine.
Des heuristiques sont donc employées pour localiser le point d'entrée des procédures. Ensuite, pour chaque procédure, une analyse récursive peut être conjuguée à une analyse séquentielle afin d'identifier les instructions de branchement intra-procédurales permettant de construire le graphe CFG. Enfin, le graphe CFG obtenu est élagué par application d'heuristiques appropriées.

Identification des procédures. Il s'agit d'une analyse inter-procédurale, dont l'objectif est de diviser le binaire en procédures qui peuvent ensuite être analysées indépendamment. Cette première étape est importante en pratique, pour des raisons de performance. Une première idée peut être de localiser l'adresse de destination des instructions `call`. Ceci suppose cependant que l'on est capable de localiser les instructions `call` au sein du programme, ce qui n'est pas toujours le cas.
En outre, même si c'était le cas, un obfuscateur peut rediriger tous les appels vers une unique fonction de branchement qui a pour mission de transférer le contrôle vers la bonne fonction cible. Cette technique modifie tous les appels de telle sorte qu'ils pointent vers une même adresse. Dans ces condition, toute information pertinente pour identifier les procédures est supprimée.
Pour résoudre ce problème, les désassembleur statique peuvent utiliser de la recherche de motif ou des heuristiques pour localiser le point d'entrée des procédures (instructions du prologue qui correspondent à la

sauvegarde sur la pile du pointeur de base et à l'augmentation de la taille de la pile en décrémentant le pointeur de pile). Cette approche est cependant sérieusement entravée par les transformations d'obfuscation du point d'entrée (*Entry Point Obscuring*).

Extraction du graphe CFG initial d'une procédure par analyse séquentielle. Une analyse séquentielle des instructions (*linear sweep*) correspond au décodage des instructions l'une après l'autre, dans l'ordre de leur apparition dans la section de code de l'exécutable ou de la procédure. Cette approche est source d'erreurs dès lors que des données sont embarquées dans le code. En outre, une analyse du flot de donnée est indispensable pour calculer les valeurs des adresses utilisées par les sauts indirects.
Elle conduit à la construction d'un graphe CFG correspondant à une sur-approximation du graphe, car elle décode toutes les instructions possible entre le début et la fin d'une procédure, à la recherche d'instructions de branchement intra-procédurales candidates. Il est nécessaire de considérer exhaustivement toutes les adresses possibles en raison de la taille variable des instructions du CPU (qui varie entre 1 et 15 pour le CPU intel x86), qui ne sont pas alignées en mémoire. Par conséquent, seule une partie des instructions décodées est valide. Une analyse séquentielle des instructions permet d'identifier un sur-ensemble des instructions de sauts intra-procédurales. Les noeuds invalides correspondent à des octets arguments d'instructions valides qui sont interprétées à tort comme des instructions de transfert de contrôle[4].

Extraction du graphe CFG initial d'une procédure par analyse récursive. Lors d'une analyse récursive des instructions (*recursive traversal*), le processus d'analyse suit le flot d'exécution. Cette approche échoue par exemple dès lors qu'une instruction de saut à une adresse variable est rencontrée, l'adresse de destination ne pouvant pas être déterminée statiquement.
Elle conduit à la construction d'un graphe CFG correspondant à une sous-approximation du graphe, dans la mesure où seule une partie du graphe peut être atteinte. Cette situation est exacerbée lors de l'analyse de code obfusqué, lorsque les appels inter-procéduraux et les sauts sont redirigés vers une fonction de branchement utilisant des sauts indirects.
Une analyse récursive des instructions, appliquée à partir de chaque adresse de saut, permet de forger le graphe CFG initial.

L'étape suivante consiste souvent à supprimer les noeuds inutiles du graphe CFG ainsi construit, en résolvant les incohérences par l'application d'heuristiques appropriées.

Dès lors que l'on est parvenu à desassembler correctement le programme et à en extraire le graphe PCG, l'objectif est d'effectuer des transformations sur ce graphe, de manière à en obtenir une représentation plus compréhensible. Nous souhaitons en particulier défaire les transformations d'obfuscation appliquées au programme.

**Réécriture du graphe PCG** L'introduction de sauts inutiles permet de rendre l'analyse statique difficile et de diversifier le code d'un programme, en réordonnant les instructions au sein du programme tout en maintenant l'ordre d'origine de l'exécution. Ces sauts inutiles peuvent être supprimés par analyse statique dès lors que l'on est capable de construire le graphe PCG exacte d'un programme. La figure 3.2 montre les deux seuls cas où un saut ne peut pas être supprimé du graphe PCG. Nous pouvons en effet observer que tout noeud possédant au moins un prédécesseur immédiat qui effectue un saut inconditionnel a nécessairement un arc entrant qui correspond

- soit à un saut conditionnel ;
- soit à un une instruction de branchement autre qu'un saut.

Dans tout les autres cas, nous pouvons considérer que le saut est inutile et le marquer comme tel. L'instruction de saut peut être supprimée et les *basics blocs* correspondants fusionnés.

L'algorithme 1 [CJK+07] donne la transformation à appliquer sur le graphe du flot de contrôle $G = (N, E)$ d'un programme pour en supprimer les sauts inconditionnels inutiles.

[4] Le jeu d'instructions du CPU Intel compte 26 instructions sur 1 octet qui codent pour une instruction de contrôle, parmi 219 instructions codées sur un octet. Dès lors, la probabilité qu'un octet argument aléatoire code pour une instruction de branchement est non négligeable (12% pour le CPU Intel).

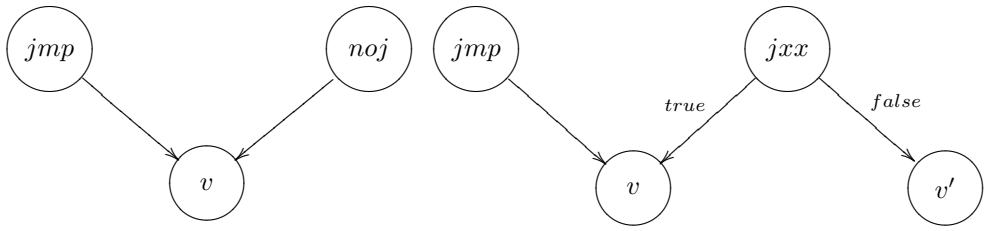



FIG. 3.2 – Invariants d'un saut

**Entrée** : un graphe $G = (N, E)$
**Sortie** : le graphe $G' = (N', E')$
$N' \leftarrow N$
$E' \leftarrow E$
**Répéter**
  $I \leftarrow \emptyset$
  **Pour** $i$ **de** $0$ **à** $|N| - 1$ **faire**
    **Si** $(P(n_i))$ **Alors**
      $I \leftarrow I \cup \{n_i\}$
    **Fin Si**
  **Fin Pour**
  **Pour** $i$ **de** $0$ **à** $|I| - 1$ **faire**
    $j \leftarrow \texttt{PredJump}(n_i)$
    $N' \leftarrow N' - \{j\}$
    $E' \leftarrow E' - \{n \in N' \mid n = (j, x) \vee n = (x, j)\}$
    $E' \leftarrow E' \cup (\texttt{Pred}(j) \times \{n_i\})$
  **Fin Pour**
**jusqu'à ce que** $(I = \emptyset)$
**Retourner** $G' = (N', E')$

Algorithme 1: Réordonner les instructions d'un programme [CJK+07]

Une implémentation de cette méthode de désobfuscation est décrite dans [CJK+07]. Elle repose sur l'utilisation des heuristiques implémentées dans IDA Pro pour implémenter un oracle de flot de contrôle, dont le rôle est d'établir la liste des successeurs d'un noeud pour toutes les exécutions possibles. Cette méthode de réécriture de graphe PCG est également utilisée, parmi d'autres, dans le cadre d'une approche dite *structurelle* [DR05] ou *morphologique* de la détection virale [BMM06, BKM08]. Cette approche repose sur la comparaison de deux graphes PCG, et fournit une solution approchée au problème d'isomorphisme de graphe PCG, soit sur la base d'heuristiques appropriées [DR05][5], soit sur la réduction du problème à celui de la recherche d'un sous-graphe [BMM06, BKM08].

Nous présentons dans la section 3.3.1 un algorithme dynamique, fondé sur ce même invariant caractérisant un noeud comme destination d'un saut inconditionnel ne pouvant pas être supprimé.

S'il est parfois possible d'appliquer ces transformations de désobfuscation sur le graphe PCG, une analyse du flot de donnée est cependant souvent indispensable pour réécrire le graphe sous une forme plus simple [MADD05]. Lorsque l'analyse a pour objectif de revenir à une représentation haut niveau du programme (décompilation vers

[5] Le module `BinDiff` [SAB07a] s'intègre au désassembleur intéractif IDA Pro et permet d'automatiser la recherche de similarité entre deux programmes (analyse différentielle de binaires). Son application à l'analyse des variantes `Bagle/W` et `Bagle/X` du virus Bagle est décrite dans [SAB07b].

un langage haut niveau), l'analyse du flot de donnée joue également un rôle primordiale. Elle s'applique alors à une représentation intermédiare du code assembleur et a pour objectif de transformer cette représentation intermédiaire en une représentation intermédiaire de plus haut niveau [Cif96].

Nous introduisons à présent les principaux concepts relatifs à l'analyse du flot de données.

**Analyse du flot de données** L'objectif de l'analyse du flot de données est de fournir une information globale sur la manière dont une procédure ou une partie plus importante d'un programme manipule ses données. Nous avons vu qu'un programme peut être représenté par un graphe de flot. L'analyse du flot de données caractérise le flot de données le long des noeuds du graphe de flot.

**Définition 3.3.3** *Un problème d'analyse du flot de données peut être caractérisé par la direction du flot d'information.*

- *Lorsque le flot d'information est dans la direction du flot d'exécution, on parle d'analyse forward.*
- *Lorsque le flot d'information est dans la direction opposée à celle du flot d'exécution, on parle d'analyse backward.*
- *Lorsque le flot d'information est dans les deux directions, on parle d'analyse bi-directionnelle.*

Puisqu'un graphe de flot peut représenter un nombre potentiellement infini de chemins d'exécution, en raison de la présence de boucles éventuelles, il est nécessaire de faire une approximation conservative de l'information du flot de données.

Nous présentons dans les deux paragraphes suivants deux problèmes classiques de l'analyse du flot de données.

**Analyse du flot de données : portée des définitions**

**Définition 3.3.4** *(Problème de la portée des définitions - reaching definition). Une définition est un assignement d'une valeur à une variable. Une définition de variable atteint un certain point dans une procédure s'il existe un chemin d'exécution de la définition à ce point tel que la variable puisse avoir à ce point la valeur assignée par la définition.*
*Le but de l'analyse de portée des définition consiste à déterminer quelles définitions particulières de chaque variable peuvent atteindre, par un chemin donné du flot de contrôle, un point particulier de la procédure.*

Ce problème peut être résolu par l'utilisation de vecteurs de bits. L'algorithme fonctionne de la manière suivante : un vecteur de bits est associé à chaque noeud du graphe CFG, chaque bit représentant la présence ou l'absence d'une information au niveau du noeud. L'exécution d'une instruction du programme peut modifier la valeur du vecteur de bits en supprimant certains bits ou en ajoutant de nouveaux bits. L'algorithme part du noeuds d'entrée et propage le vecteur de bit initial à travers le graphe CFG jusqu'à ce que la valeur associée à chaque vecteur de bit se stabilise. L'ensemble des valeurs finales des vecteurs de bits associés aux noeuds du graphe CFG fournit une solution au problème.

La propagation du vecteur de bits à travers le graphe CFG peut être modélisé par le système d'équations récursives suivant :

$$\begin{cases} in(i) & = & \bigcap_{(j,i)\in E} out(j) \\ out(i) & = & gen(i) \cup (in(i) - kill(i)) \end{cases}$$

où $in(i)$ est l'ensemble des définition en entrée du noeud $i$, $out(i)$ est l'ensemble des définition en sortie du noeud $i$, $gen(i)$ est l'ensemble des définitions générées par le noeud $i$ et $kill(i)$ est l'ensemble des définitions supprimées par le noeud $i$.

Si tous les problèmes de l'analyse du flot de données ne peuvent pas être résolus par l'utilisation d'un simple vecteur de bits, de nombreux problème de l'analyse du flot de données peuvent cependant être résolus par une approche équationelle. Ces problèmes sont alors formulés en utilisant un système d'équation de flot plus général :

$$\begin{cases} in(i) & = & \bigcap_{(j,i)\in E} out(j) \\ out(i) & = & F_i(in(i)) \end{cases}$$

où $F_i$ est la fonction de transfert qui modifie les valeurs du flot de données lorsqu'il se propage à travers le noeuds $i$ et $\bigsqcap$ est l'opérateur de combinaison des valeurs de flots venants des différents prédécesseurs.

Dans l'approche équationnelle, on recherche un point fixe. La valeur du flot de donnée associée à un noeud du flot de contrôle ne dépend que de la valeur initiale du flot de donnée au noeud de départ $s$. La valeur du flot au noeud $i$ est la composée de $f_{s\to i}(I_0)$ pour tous les chemins $s \to i$, où $I_0$ désigne la valeur initiale du flot et $f_{s\to i}$ représente le résumé de la fonction de transfert sur le chemin $s \to i$.

L'étude des fonctions de transfert et de leur cloture transitive a conduit à une seconde approche pour résoudre les problèmes d'analyse de flot de données, fondée sur l'utilisation d'un arbre de contrôle. Ces méthodes sont appelées, pour des raisons historiques, méthodes d'élimination. Elles se fondent sur la théorie des treillis des fontions de flot. Le modèle correspondant de l'analyse du flot de données est dit monotone. L'implémentation de ces méthodes d'analyse se fait généralement en trois étape : analyse structurelle, réduction et propagation.

- L'analyse structurelle du flot de contrôle est une généralisation de l'analyse des intervalles. Ces deux methodes d'analyse du flot de contrôle permettent de construire un arbre de contrôle.
- La réduction correspond au calcul du résumé des fonctions de transfert sur chaque intervalle.
- La propagation correspond au calcul effectif des valeurs de flot.

Le modèle d'analyse du flot de données monotone fournit une structure algébrique permettant de coder la plupart des problèmes d'analyse du flot des données. Il fournit un cadre suffisament large pour appréhender des problèmes de l'analyse du flot de données plus complexe que le modèle du vecteur de bits n'est pas en mesure de traiter correctement, en raison notamment de la faible expressivité des opérations de combinaison qu'il autorise (l'union et l'intersection).

Nous décrivons en détail le modèle d'analyse du flot de données monotone au chapitre 4. Le problème de propagation de la constante, présenté au paragraphe suivant, est un exemple de problème que l'on ne peut pas résoudre efficacement par la seule utilisation d'un vecteur de bits. Nous donnons un exemple d'analyse du flot de donnée par propagation de la constante dans le cadre formel de l'interprétation abstraite au chapitre 4.

Nous pouvons dans le cadre des modèles précédents, définir deux types de solution à un problème d'analyse du flot de données d'un programme :

**Définition 3.3.5** *(Meet Over all Path). On définit une solution MOP d'un problème d'analyse du flot de donnée* $\Pi$ *par :*

$$\sigma(i) = \begin{cases} I_0 & \textit{si } i = s \\ \bigsqcap\limits_{p\in\{s\to i\}} f_p(I_0) & \textit{sinon.} \end{cases}$$

**Définition 3.3.6** *(Maximal FixPoint). On définit une solution MFP d'un problème d'analyse du flot de donnée* $\Pi$ *par :*

$$\sigma(i) = \begin{cases} I_0 & \textit{si } i = s \\ \bigsqcap\limits_{(j,i)\in E} f_{(j,i)}(\sigma(j)) & \textit{sinon.} \end{cases}$$

La solution MOP apparaît comme la solution idéale puisqu'elle considère tous les chemins possibles du programme. Malheureusmeent, il n'existe pas d'algorithme général pour calculer la solution MOP. Intuitivement, ce résultat d'impossibilité vient du fait qu'il y a potentiellement une infinité de chemins à prendre en compte lorsqu'il y a une boucle dans le programme qui contribue au calcul du flot de données à chaque itération de la boucle.
La solution MFP est moins précise, puisqu'elle ne considère que les effets des noeuds voisins sur le flot de donnée, mais est généralement calculable. Lorsque l'on impose aux fonctions du treillis d'être distributive, la solution MFP est aussi la solution MOP. De nombreux problèmes de l'analyse du flot de données sont distributifs. Le problème de propagation de la constante, présenté au paragraphe suivant, ne l'est pas.

**Analyse du flot de données : propagation de la constante**

**Définition 3.3.7** *(Problème de propagation de la constante). L'analyse de la propagation de la constante cherche à déterminer si sur tous les chemins qui vont de l'assignement d'une variable $x$ à une constante $c$, $x \leftarrow c$, jusqu'à l'utilisation effective de la valeur de la variable $x$, la valeur de la variable $x$ reste constante et égale à $c$. Le cas échéant, la variable peut être remplacée par la constante $c$.*

Le problème correspondant peut être formulé de deux manière :

**May Constants** : la question est de déterminer si la variable $x$ est de valeur constante au niveau de l'instruction $s$ durant certaines exécutions du programme.

**Must Constants** : la question est de déterminer si la variable $x$ est de valeur constante au niveau de l'instruction $s$ pour toutes les exécutions du programme.

Ces problèmes sont d'autant plus complexe lorsque le langage permet des assignations par le biais d'instructions évoluées. Nous pouvons distinguer cinq cas [MOR01] :

cas 1 : l'assignation de la variable $x$ n'utilise pas d'opérateurs ($x \leftarrow c$) ;
cas 2 : l'assignation est une opération linéaire $x \leftarrow ay + b$ ;
cas 3 : l'assignation n'utilise que les opérateurs $\{+, -\}$ ;
cas 4 : l'assignation n'utilise que les opérateurs $\{+, -, *\}$ ;
cas 5 : l'assignation peut utiliser les opérations $\{+, -, *, \texttt{div}, \texttt{mod}\}$.

Le tableau 3.1 présente la complexité des problèmes **May Constants** et **Must Constants** dans le cas où le flot de contrôle ne comporte pas de boucle [MOR01]. Nous pouvons constater que le problème **May Constants** est toujours au moins aussi difficile que le problème **Must Constants**.

Le tableau 3.2 présente la complexité des problèmes **May Constants** et **Must Constants** dans le cas où le

| | **Must Constants** | **May Constants** valeur unique | **May Constants** valeurs multiples |
|---|---|---|---|
| Cas 1 | **P** | **P** | **NP**-complet |
| Cas 2 | **P** | **P** | **NP**-complet |
| Cas 3 | **P** | **NP**-complet | **NP**-complet |
| Cas 4 | **co-NP** complet | **NP**-complet | **NP**-complet |
| Cas 5 | **co-NP** complet | **NP**-complet | **NP**-complet |

Tab. 3.1 – Complexité des problèmes **May Constants** et **Must Constants** dans le cas où le flot de contrôle ne comporte pas de boucle [MOR01]

flot de contrôle ne comporte pas de boucle [MOR01]. Ces problèmes sont d'autant plus complexes selon que le langage autorise les boucles ou pas. Dans ce contexte, il est donc le plus souvent nécessaire de mener une analyse

| | **Must Constants** | **May Constants** valeur unique | **May Constants** valeurs multiples |
|---|---|---|---|
| Cas 1 | **P** | **P** | **PSPACE**-complet |
| Cas 2 | **P** | **NP**-difficile | indécidable |
| Cas 3 | **P** | indécidable | indécidable |
| Cas 4 | **NP**-difficile | indécidable | indécidable |
| Cas 5 | indécidable | indécidable | indécidable |

Tab. 3.2 – Complexité des problèmes **May Constants** et **Must Constants** dans le cas où le flot de contrôle peut comporter des boucles [MOR01]

approximative du flot de données. Deux concepts influent notablement sur la difficulté d'une technique d'analyse du flot de données :

– La sensibilité au flot de contrôle : une technique d'analyse est dite sensible au flot de contrôle si elle prend en compte l'ordre des noeuds dans le graphe du flot de contrôle. Dans le cas contraire, l'analyse est dite insensible

au flot de contrôle. Les analyses insensibles au flot de contrôle sont généralement moins coûteuses et se prêtent mieux à une approche compositionnelle, car elles ne dépendent pas de l'ordre des noeuds. En revanche, elles sont moins précises.
- La sensibilité au contexte : une technique d'analyse est dite sensible au contexte d'exécution si elle fait la distinction entre les contextes de la procédures d'appel, et considère donc les effets de différents contextes. Dans le cas contraire, l'analyse est dite insensible au contexte d'exécution.
  Le problème de la détermination des chemins d'exécution valides est un exemple d'analyse sensible au contexte, où le contexte est la pile d'exécution. Les techniques d'analyses sensibles au contexte d'exécution sont plus précises, mais aussi souvent plus coûteuses que les techniques d'analyse qui y sont insensible.

Remarquons que ces deux notions de sensibilité sont orthogonales : il y a des techniques d'analyse représentatives des quatres combinaisons de ces concepts.

**Analyse inter-procédurale du flot de données** L'analyse inter-procédurale du flot de données est plus complexe que l'analyse intra-procédurale, dans la mesure où elle ne se réduit pas à une combinaison élémentaire des analyse du flot de données des procédures constituant le programme. Ainsi, nous pouvons être tenté d'étendre naturellement la définition de solution MOP sur l'ensemble des chemins inter-procéduraux. Malheureusement, l'existence de chemins irréalisables a pour conséquence d'induire une trop grande imprécision de la solution MOP ainsi étendue au cas inter-procédural. Pour contourner cette difficulté, on définit de nouveaux types de solutions à un problème d'analyse du flot de données d'un programme :

**Définition 3.3.8** *(Meet over all Valid Path). On définit une solution MVP d'un problème d'analyse du flot de données* $\Pi$ *de la même manière qu'une solution MOP, en ne considérant que les chemins d'exécution valides.*

**Définition 3.3.9** *(Inter-procedural Maximal FixPoint). On définit une solution IMFP d'un problème d'analyse du flot de données* $\Pi$ *de la même manière qu'une solution IMFP, en ne considérant que les chemins d'exécution valides.*

Puisque la solution MVP est une extension de la solution MOP, le problème correspondant est donc également indécidable. En revanche, la solution IMFP est approximative, mais généralement calculable.

Outre le problème des chemins d'exécutions non valides, de nombreuses difficultés peuvent rendre l'analyse inter-procédurale du flot de données notablement plus difficile que l'analyse intra-procédurale, comme par exemple les alias causés par le passage des paramètres.

**Analyse des alias** L'analyse des alias est cruciale lorsque l'on tente d'optimiser du code par application de certaines méthodes d'analyse du flot de données. Si l'analyse des alias est incorrecte, on risque de produire un code incorrect.

**Définition 3.3.10** *(Alias). Un alias apparait à un certain point de l'exécution d'un programme lorsque au moins deux noms existent pour désigner une même zone mémoire. Par exemple, en C, l'instruction* `p=&v` *crée un alias entre* `*p` *et* `v`.

L'analyse *liveness* ou analyse des variables vivantes conditionne l'application de nombreuses transformations d'optimisation. Nous allons voir qu'elle requiert une analyse des alias.

**Définition 3.3.11** *(Problème des variables vivantes). Soit $P$ un programme et $s$ une étiquette localisant une instruction dans le programme $P$. Une variable $x$ est vivante au niveau de l'instruction $s$ si et seulement si pour une exécution du programme $P$, la variable $x$ est utilisée, après que l'instruction $s$ ait été exécutée, sans être redéfinie. Le problème d'analyse statique sous-jacent consiste à déterminer l'ensemble des couples $(x, s)$ tels que la variable $x$ est vivante au niveau de l'instruction $s$. Une variable vivante est donc une variable dont on ne doit pas perdre la valeur car elle sera utilisée plus tard.*

Considérons le problème qui consiste à trouver les variable vivante du programme suivant :

$s_1$ : `v=1 ;`
$s_2$ : `p=&v ;`
$s_3$ : `w=2 ;`
$s_4$ : `printf("%d", *p) ;`

La variable `v` est vivante au niveau de l'instruction $s_3$ car `*p` et `v` désignent la même zone mémoire lorsque l'instruction $s_4$ est exécutée. De nombreux problèmes de l'analyse statique sont donc influencés par les problèmes d'alias. Tout problème comportant un problème d'alias sous-jacent est au moins aussi difficile que le problème d'alias mis en jeux.

La détermination des alias peut être intra-procédurale ou inter-procédurale. Elle peut être sensible au flot d'exécution ou insensible au flot d'exécution.
L'information sur un alias est sensible au flot d'exécution si elle est dépend du flot de contrôle. Cette information est insensible au flot d'exécution si elle est indépendante du flot de contrôle. L'information sur un alias peut indiquer ce qui peut se produire sur un chemin du graphe de flot (**May Alias**). Ce sera le cas si la procédure comprend des chemins qui assignent l'adresse d'une variable $y$ à une variable $x$ et d'autres chemins qui assignent l'adresse de la variable $z$ à une variable $x$. L'information sur un alias peut indiquer ce qui se produit sur tous les chemins du graphe du graphe de flot (**Must Alias**). Ce sera le cas par exemple si tous les chemins d'une procédure assignent l'adresse d'une variable $x$ à une variable $y$ et n'assignent que cette valeur à la variable $y$.

La caractérisation formelle des problèmes d'alias dépend fortement de la nature de l'information recherchée (inter-procédurale/intra-procédurale, sensible/insensible au flot d'exécution, may/must alias). En effet, nous avons les cas suivants :

- **May Alias** insensible au flot : la relation d'alias considérée est une relation binaire réflexive, symétrique et intransitive.
- **Must Alias** insensible au flot : la relation d'alias considérée est une relation binaire réflexive, symétrique et transitive.
- **May Alias** sensible au flot : la relation d'alias considérée est une fonction qui à une instruction $s$ et à une variable $x$ associe un ensemble de zones de stockage abstraites, $Alias(s, x)$. La relation $Alias(s, x) = L$ signifie alors qu'au niveau de l'instruction $s$, la variable $x$ peut faire référence à n'importe quelle zone de stockage de l'ensemble $L$. Pour un point fixé $s_0$ du programme, la relation $Alias(s_0, x) \cap Alias(s_0, y) \neq \emptyset$ est une relation réflexive, symétrique et intransitive. Pour une variable fixée $x_0$ du programme, la relation $Alias(s, x_0) \cap Alias(t, x_0) \neq \emptyset$ est une relation réflexive, symétrique et intransitive.
- **Must Alias** sensible au flot : la relation d'alias considérée est une fonction qui à une instruction $s$ et à une variable $x$ associe une zone de stockage abstraite, $Alias(s, x)$. La relation $Alias(s, x) = l$ signifie alors qu'au niveau de l'instruction $s$, la variable $x$ doit faire référence à la zone de stockage $l$. Pour un point fixé $s_0$ du programme, la relation $Alias(s_0, x) = Alias(s_0, y)$ est une relation réflexive, symétrique et symétrique. Pour une variable fixée $x_0$ du programme, la relation $Alias(s, x_0) = Alias(t, x_0)$ est une relation réflexive, symétrique et symétrique.

Nous voyons donc que la caractérisation d'un alias comme relation dépends fortement des hypothèses faites. Cette distinction conduit à des informations différentes sur les alias. Cette distinction est également importante car elle va conditionner la complexité du problème considéré. Un problème insensible au flot d'exécution peut de manière générale être résolu en résolvant des sous-problèmes et en combinant leurs solutions pour fournir une solution au problème entier. Au contraire, un problème sensible au flot d'exécution impose que nous soyons en mesure de suivre les chemins du graphe de flot pour calculer une solution.

Le problème de l'analyse d'alias est plus difficile dans le cas de l'analyse inter-procédurale que dans celui de l'analyse intra-procédurale. Si l'on se limite au cas de l'analyse intra-procédurale, le problème de l'analyse d'alias est plus difficile si la détermination des alias est dépendante du flot d'exécution que si elle y est insensible. Même si l'on se limite au cas d'une analyse intra-procédurale et insensible au flot d'exécution des alias, le problème **Must Alias** est plus difficile que le problème **May Alias**.

Dans sa thèse [Lan92], Landi rappelle que la complexité du problème des alias varie en fonction du type des mécanismes à l'origine de la création d'alias et de la manière dont ils sont combinés. Il traite principalement de l'analyse intra-procédurale des alias. Le problème des alias est selon les cas dans **P**, **NP**-difficile ou **co-NP**-difficile. Lorsqu'il y a au moins deux niveaux d'indirection, indépendament du mécanisme utilisé pour créer ces alias, le problème de détermination des alias est **NP**-difficile. Cependant, avec un seul niveau d'indirection, alors le problème

d'aliasing est dans **P** [ZR94]. Il montre également que le problème des alias en présence de certains mécanismes est **PSPACE**-complet. Par conséquent, avec un langage de programmation comme le C, autorisant l'usage de pointeurs généraux, le problème de la détermination des alias est considéré comme **NP**-difficile.
Si le langage autorise de multiples niveaux de pointeurs, le problème d'alias intra-procédural et insensible au flot d'exécution **May Alias** est **NP**-difficile [Hor97].
Si le langage autorise de multiples niveaux de pointeurs, le problème d'alias intra-procédural et sensible au flot d'exécution **May Alias** est **NP**-difficile. Sous ces hypothèses, le problème **Must Alias** est **co-NP**-difficile. Si le langage considéré autorise également l'allocation dynamique de mémoire, le problème **May Alias** devient indécidable (il reste semi-décidable), le problème **Must Alias** devient incalculable (il n'est même pas semi-décidable).
Le tableau 3.3 présente un résumé de ces résultats dans le cas de l'analyse intra-procédurale et sensible au flot d'exécution des alias.

| | **May Alias** | **Must Alias** |
|---|---|---|
| pointeurs simples | **P** | **P** |
| pointeurs multiples | **NP**-difficile | **co-NP** difficile |
| pointeurs avec au moins 4 niveaux d'indirection | **PSPACE**-complet | **PSPACE**-complet |
| allocation dynamique | indécidable | incalculable |

Tab. 3.3 – Complexité des problèmes **May Alias** et **Must Alias** dans le cas dans le cas de l'analyse intra-procédurale et sensible au flot d'exécution [Lan92, Hor97]

**Limites de l'analyse statique** Nous avons vu dans cette section les principales étapes de l'analyse statique d'un programme et les principales difficultés rencontrées :

- au niveau du désassemblage : il est possible de modifier un binaire de telle sorte que l'analyse (le décodage) des instructions devienne difficile. Une première approche exploite le fait que le jeu d'instructions du CPU contienne des instructions de longueur variable qui peuvent s'exécuter à n'importe quelle adresse mémoire. En ajoutant des octets à des zones qui ne peuvent pas être atteintes durant l'exécution, les désassembleur peuvent être amenés à interpréter de manière incorrecte de larges parts d'un exécutable [LD03].
- la localisation du point d'entrée des procédures est entravée par l'application de méthodes élémentaires d'obfuscation du point d'entrée ou par l'utilisation de tables de pointeurs de fonctions [OSSM02, OSSM03].
- l'extraction du graphe CFG d'une procédure est également entravée par l'utilisation de tables de redirection des adresses de saut et des sauts à des adresses variables. Dès lors, une analyse du flot de données doit être menée pour améliorer la précision du graphe CFG. Dans ces conditions, l'embarquement de problèmes difficiles de l'analyse statique du flot de données pose problème. La difficulté augmente lorsque l'analyse doit être interprocédurale, et en fonction de sa sensibilité au flot de contrôle et au contexte.

Cependant, la limitation la plus importante de l'analyse statique réside probablement dans l'utilisation intempestive de sauts à des adresses variables.
La figure 3.3 illustre le problème de l'utilisation de sauts à des adresses variables, à travers l'utilisation des moteurs d'analyse statique d'OllyDbg [Yus07] (plugin OllyGraph [Ope07]) et d'IDA Pro [Dat07].

- Dans le cas d'OllyGraph, l'instruction `call eax` n'est pas considérée comme une instruction de branchement. L'analyse s'arrête au *basic block* contenant l'instruction de saut.
- Dans le cas d'IDA Pro, la granularité des *basic blocks* prend en compte l'instruction de saut mais occulte la partie du programme invoquée par le saut.

Ce dernier problème est résolu par l'analyse dynamique.

### Analyse dynamique

Nous avons vu au paragraphe précédent que l'analyse statique souffre de certaines limitations, dont celle d'aboutir à une sur-approximation (conservative) du graphe PCG et l'incapacité à résoudre certains problèmes liés à l'utilisation de variables dynamiques. L'analyse dynamique résouds naturellement le second problème. La figure 3.4, page 63

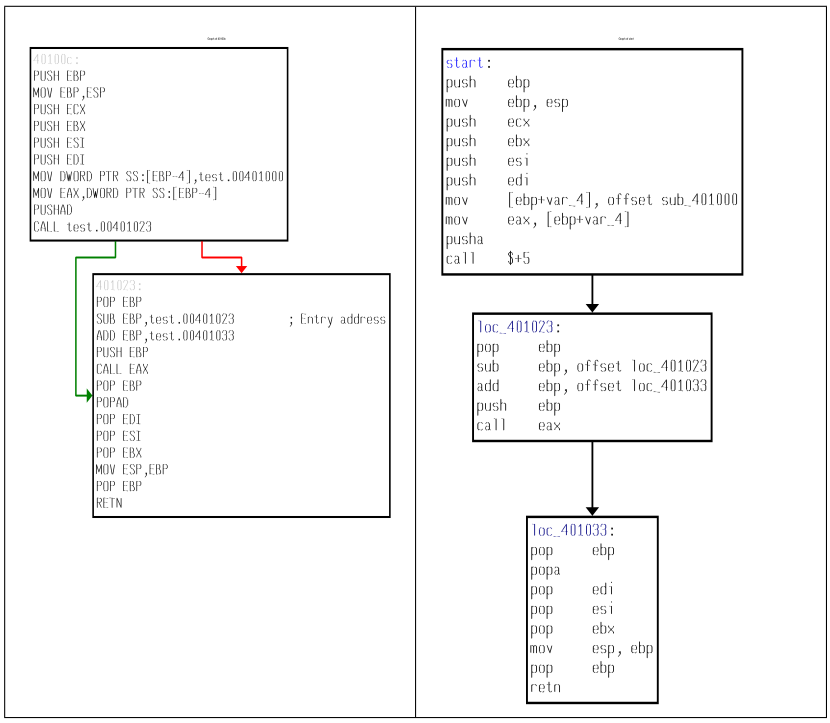



FIG. 3.3 – Graphes VCG de `call eax` sous Ollygraph et IDA Pro

illustre la résolution du problème de l'utilisation de sauts à des adresses variables, par utilisation de l'outil d'analyse dynamique VxStripper. Une analyse dynamique peut donc permettre d'analyser correctement le code en présence

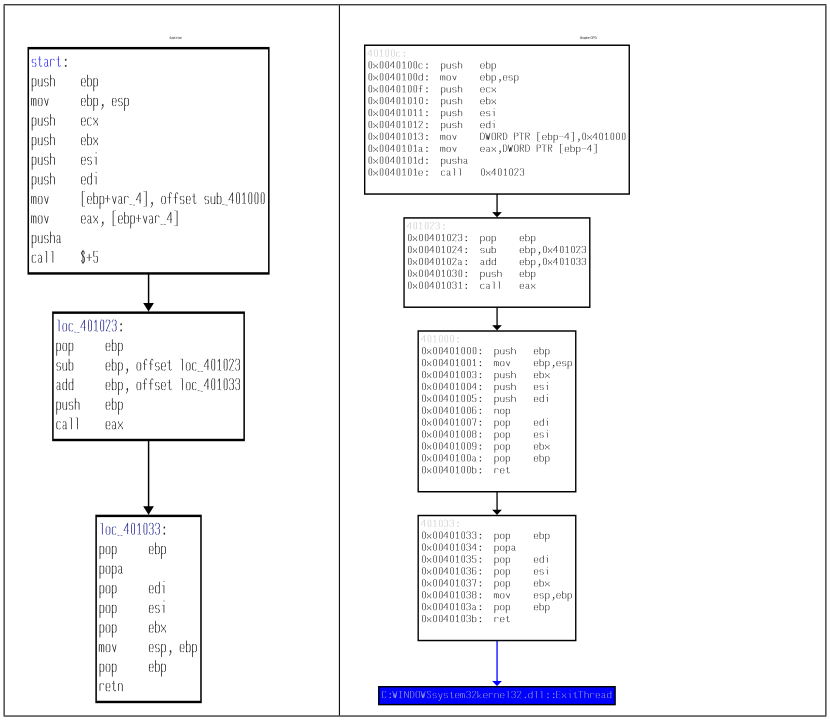



FIG. 3.4 – Graphe VCG de `call eax` sous IDA Pro et VxStripper

de sauts à des variables.
L'algorithme suivant est implémenté par `VxStripper` pour construire le graphe PCG d'un programme binaire.

1. L'algorithme d'analyse comporte une phase d'initialisation, lors de laquelle les nombres de noeuds et d'arêtes sont initialisés à 0.
2. Si le bloc de base courant n'a pas encore été rencontré, un nouveau noeud est créé et ajouté à la liste. Si le noeud courant a un père, celui-ci est recherché dans la liste et si la dernière instruction du noeud père est une instruction de branchement (saut, appel de procédure, répetition de chaîne, retour de procédure), l'arête liant le père au fils est créée. Dans le cas où la dernière instruction du père est une instruction de branchement vers une fonction d'API, un noeud supplémentaire est créé pour représenter cet appel et lié au noeud père par une arête.
3. L'algorithme d'analyse comporte une phase de terminaison, lors de laquelle un noeud correspondant à la dernière fonction d'API invoquée est créé. Ce noeud est lié au noeud courant par une arête.

Cependant, la limitation majeure de l'analyse purement dynamique provient du fait que seule une partie du graphe PCG est susceptible d'être découverte (lors de l'exploration d'un ou plusieurs chemins d'exécution), aboutissant donc à une sous-approximation du graphe PCG. Pour augmenter la taille du graphe PCG, et obtenir ainsi une représentation suffisamment complète pour permettre la mise en oeuvre d'une attaque efficace, l'idée consiste à forcer l'exploration de plusieurs chemins d'exécution [MKK07]. Cette exploration peut être :

1. systématique, c'est-à-dire que l'on localise tous les branchements conditionnels par lesquels le flot d'exécution passe et que l'on inverse la condition de manière à explorer une nouvelle branche ;
2. gouvernée par une analyse du flot de données, en traçant dynamiquement certaines données utilisées par le programme comme entrée et en identifiant les endroits où cette entrée est utilisée pour prendre une décision qui va modifier le flot de contrôle du programme.

Prenons l'exemple d'un programme qui contrôle la présence d'un fichier sur le système avant d'exécuter certaines portions de son code. Dans le premier cas (exploration systématique des chemins d'exécution), le principe consiste, pour chaque branchement conditionnel visité, à effectuer une sauvegarde de l'image mémoire du processus et de l'état du CPU. Après exploration de la branche courante du graphe des appels, et immédiatement avant terminaison du processus, l'image mémoire et l'état du CPU sauvegardés sont restaurés. La valeur de retour est modifiée de manière à déclencher l'exploration d'une nouvelle branche du graphe.

Le principal inconvénient de cette méthode est que le nombre d'états visités augmente très vite et que notre exploration systématique risque de découvrir des chemins qui ne peuvent pas être exécutés, ou qui ne présentent pas d'intérêt pour l'analyste. Son intérêt est qu'il permet d'obtenir un graphe PCG qui contient le graphe exacte, et un désassemblage fiable des *basics blocs* le constituant.

Dans le second cas (l'exploration des chemins d'exécution est gouvernée par une analyse du flot de données), le principe consiste à instrumenter l'appel système qui contrôle la présence d'un fichier et à marquer la donnée retournée par le programme. Lorsque cette donnée est utilisée ensuite lors d'un branchement conditionnel, une sauvegarde de l'image mémoire du processus et de l'état du CPU est effectuée. Après exploration de la branche courante du graphe des appels, et immédiatement avant terminaison du processus, l'image mémoire et l'état du CPU sauvegardés sont restaurés. La valeur de retour est modifiée de manière à déclencher l'exploration d'une nouvelle branche du graphe.

La principale limitation de cette méthode est qu'elle requiert la capacité à analyser le flot de données de manière précise, ce qui n'est pas toujours possible. Dès lors, le graphe PCG résultant de l'analyse est potentiellement incomplet. Il s'agit donc d'une méthode approximative. Son principal atout est qu'elle permet à l'analyste d'extraire sélectivement l'information dont il a besoin concernant les données manipulées par le programme : la durée et la complétude de l'analyse sont paramétrables, dans la mesure où l'on peut choisir le nombre de fonctions d'API qui seront instrumentées.

Une application élémentaire de cette méthode [MKK07] consiste à analyser le flot de données de la manière suivante (*memory tainting*) :

- La zone mémoire $M_1$ qui recoit la valeur de retour d'une fonction d'API instrumentée est marquée/étiquetée (étiquette $L_1$).
- Lorsqu'une opération utilise la valeur de $M_1$, les zones mémoires $M_i$ contenant le résultat de l'opération sont marquées au moyen de l'étiquette $L_1$ : à la zone mémoire $M_i$ est associée l'étiquette $L_1$ et une représentation du calcul effectué permettant de rendre compte de la propagation des données teintées, par exemple $L_i = L_1 \mathcal{R} X$, où $\mathcal{R}$ est une opération arithmétique ou logique et $X$ une valeur ou une étiquette $L_j$.
- Une résolution du système de contraintes obtenu permet de déterminer comment les zones mémoires $M_i$ doivent être modifiées de manière cohérente avant exploration d'un nouveau chemin d'exécution.

La résolution du système de contraintes sera possible si toutes les opérations teintées peuvent être représentées par des contraintes linéaires. Ce ne sera pas le cas par exemple, si des opérations booléennes non linéaires sont effectuées ou si la valeur de $M_1$ sert d'index dans une table. Si une dépendance non-linéaire existe entre deux étiquettes $L_i$ et $L_j$, le système de contraintes ne pourra pas être résolu et le système ne pourra pas être replacé dans un état parfaitement cohérent avant exploration d'un nouveau chemin d'exécution.

Lorsque le système de contraintes n'est pas linéaire, l'utilisation d'un algorithme comme SAT permet d'obtenir de meilleur résultats qu'une analyse purement linéaire.

Nous percevons donc les même limites en analyse statique et dynamique lorsqu'un branchement conditionnel est déclenché par une valeur qui est liée aux autres valeurs par une dépendance fortement non linéaire. Cette limitation est particulièrement gênante dans le cas de programmes qui contrôlent l'intégrité de leur environnement d'exécution par utilisation de fonctions de hachage à sens unique.

Cependant, nous voyons que l'analyse conjointe du flot de donnée (certaines données) et du flot de contrôle (branchements conditionnels) permet d'obtenir un graphe CFG plus complet à la demande de l'analyste. L'exploration de nouveaux chemins du graphe PCG, lorsqu'elle est gouvernée par l'analyse du fux des données d'interaction avec l'environnement, permet de retrouver facilement les conditions de déclenchement environnementales, lorsqu'elles ne sont pas protégées contre l'analyse linéaire des contraintes.

L'analyse dynamique permet de lever facilement certaines transformations d'obfuscation, lorsqu'elles ne sont pas suffisament robustes (comme par exemple l'utilisation de sauts inconditionnels ou de sauts conditionnés par un prédicat opaque déterminé pour réordonner les instructions d'un programme, ou l'utilisation de *junk code* mort).

**Réordonner les instructions d'un programme** Nous avons vu dans la section 3.3.1 une méthode d'analyse statique pour réordonner les instructions d'un programme en supprimant les sauts inconditionnels inutiles intentionnellement ajoutés (principalement pour diversifier le code d'un programme ou modifier l'ordre des instructions). Cette méthode repose sur les nombreuses heuristiques implémentées par IDAPro et fait l'hypothèse que le programme cible a pu être désassemblé correctement par analyse statique.
Le même principe peut s'appliquer à l'analyse dynamique d'un programme, sans être soumis aux mêmes restrictions. La figure 3.5 présente l'adaptation de cette méthode statique à l'analyse dynamique. La partie gauche de la figure montre le programme obfusqué. Dans la partie centrale, les noeuds violant l'invariant sont indiqués en rouge. La partie droite montre le graphe de flot du programme après suppression des sauts inconditionnels superflus.

Remarquons que l'analyse dynamique permet de supprimer naturellement le *junk code* mort et de *casser* les prédicats opaques déterminés.

**Suppression des prédicats opaques** Nous avons vu à la section 3.2.2 qu'un prédicat opaque est un prédicat dont la valeur est connue lors de la compilation, mais qui est difficile à établir après compilation, par analyse du programme. Nous considérons dans cette section les prédicats opaques déterminés et dont le fondement est une propriété mathématique, comme $\forall x,\ x|f(x)$ ou $\forall x,\ f(x) \subseteq P$.

L'analyse purement dynamique d'un programme permet de supprimer naturellement ces prédicats opaques. Le principe consiste à exécuter le programme avec différentes entrées et à observer le chemin parcouru après chaque instruction de branchement conditionnel. Une instuction de branchement conditionnel sera classifiée comme condi-

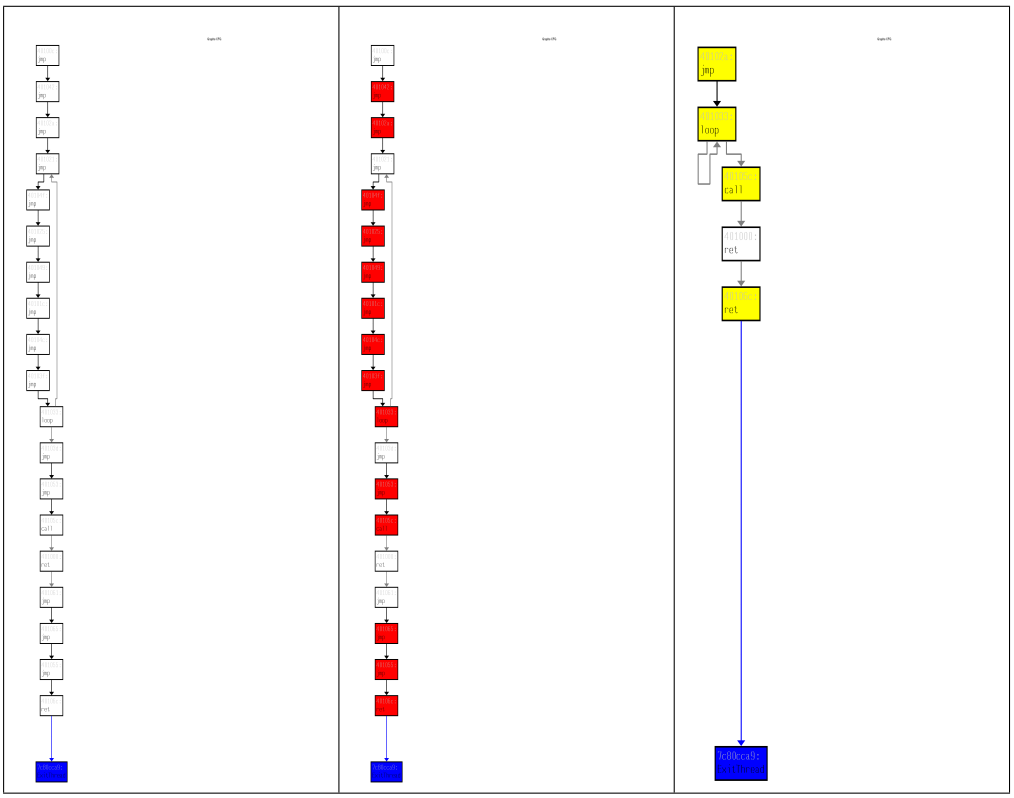



FIG. 3.5 – Suppression des sauts inutiles sous `VxStripper`

tionnée par un prédicat opaque toujours vrai si durant toutes les exécutions du programme, le chemin vrai est toujours emprunté. Une analyse purement dynamique permet donc de détecter tous les prédicats opaques exécutés, mais en raison de l'espace limité des entrées soumises, elle peut classifier un prédicat comme opaque alors que ce n'est pas le cas.

Un attaquant peut tenter d'améliorer le taux de détection en forgant des données de test permettant d'augmenter le pourcentage CDC (Condition/Decision Coverage) de sauts conditionnels qui sont exécutés comme vrai au moins une fois et faux au moins une fois. Malheureusement, des données expérimentales établissent que ce pourcentage est au plus de 58% pour des programmes complexes.

L'analyse purement dynamique (en force brute) permet de supprimer tous les prédicats opaques déterminés, mais ne permet pas de mettre en oeuvre une procédure de détection précise.

Une méthode plus efficace consiste à identifier les sauts conditionnels, puis à exécuter le *basic block* correspondant sur l'espace des entrées possibles de ses instructions. De manière plus précise, pour chaque saut inconditionnel $j$, on considère l'instruction $j-1$ le précédent au sein du *basic block*. Cette instruction utilise des registres dont on fait varier la valeur avant d'exécuter les instructions $j-1$ et $j$. Si pour toutes les exécutions, l'un des chemin vrai ou faux est systématiquement emprunté, le saut inconditionnel est classifié comme conditionné par un prédicat opaque. Dans le cas contraire, on itère cette procédure avec l'instruction $j-2$, jusqu'à ce qu'un prédicat opaque soit détecté ou que la fin du basic bloc soit atteinte. Dans ce dernier cas, le prédicat est classifié comme n'étant pas opaque.

Cette méthode de détection des prédicats opaques est très précise. Elle est cependant très coûteuse. En effet, Si $n$ est le nombre d'instructions du prédicat opaque, $r$ le nombre de registres qu'il utilise et $w$ la taille des registres, le nombre d'itération nécessaire en pire cas de l'algorithme est de l'ordre de $n^2 2^{wr}$. Remarquons également que nous faisons l'hypothèse implicite que les instructions du prédicat opaque sont toutes regoupées, et au sein d'un même

*basic block*.

Il est cependant possible d'améliorer notablement l'efficacité de cet algorithme, en le spécialisant pour un type donné de prédicat opaque. Cela signifie que dès lors que l'on connait la bibliothèque de prédicats opaques utilisés par une solution de protection logicielle, on est en mesure de les détecter en un temps raisonnable, et ce indépendament de la manière dont ils sont implémentés.

Le principe consiste à utiliser les résultats de l'interprétation abstraite [DPMDG06] pour modéliser les attaquants et les classifier relativement à leur niveau de précision. Un attaquant peut être modélisé par un domaine abstrait $A$. Si $n$ est le nombre d'instructions du prédicat opaque, $r$ le nombre de registres qu'il utilise et $d$ est le nombre de valeurs abstraites, le nombre d'itération nécessaire en pire cas de l'algorithme basé sur l'interprétation abstraite est de l'ordre de $n^2 d^r$.

Cette méthode est limitée dans le cas où deux prédicats opaques correspondant à deux attaquants spécialisés (modélisés par des domaines abstraits différents) sont dépendants. Prenons le cas d'un prédicat opaque $P_1$ toujours vrai, dont le chemin faux conduit à une instruction quelconque du prédicat opaque $P_2$. Le predicat opaque $P_2$ est toujours vrai, son chemin faux conduit à une instruction quelconque du prédicat opaque $P_1$. L'attaquant modélisé par le domaine abstrait $A_1$ devrait être en mesure de détecter le prédicat opaque $P_1$. Mais l'arc de $P_2$ vers $P_1$ protège $P_1$. Nous voyons donc que des prédicats opaques difficiles à casser peuvent protéger des prédicats moins robustes, vis-à-vis d'attaquants spécialisés.

#### Analyse hybride statique-dynamique

Les transformations d'obfuscation sont conçues pour entraver l'analyse statique. Elle visent le plus souvent à complexifier le graphe des chemins d'exécutions.
Les méthodes d'analyse statique (propagation de la constante, par exemple) propagent généralement l'information sur un sur-ensemble des chemins d'exécution effectivement parcourus lors d'une exécution du programme. Ces méthodes d'analyse sont, pour cette raison, dite conservatives.
Les méthodes d'analyse dynamique (*tracing*, *profiling*) ne peuvent généralement pas prendre en compte toutes les valeurs d'entrées ou toutes les interactions environnementales possibles d'un programme, et ne peuvent donc observer qu'un sous-ensemble des chemins d'exécution possibles. L'information est obtenue dynamiquement en exécutant le programme et en observant les chemins empruntés lors de l'exécution. Parce qu'il n'est pas possible d'observer le programme pour toutes les entrées et interactions possibles, ces méthodes d'analyse sont dites approximatives.
Ces deux approches, duales par nature, peuvent être utilisées conjointement. Les méthodes d'analyse hybrides [MADD05] conjuguent une analyse dynamique, permettant de reconstruire une partie du graphe, avec une analyse statique, permettant d'ajouter des noeuds au graphe. Il est possible également de conjuguer d'abord une analyse statique, permettant de construire une sur-approximation du graphe, avec une analyse dynamique permettant d'élaguer le graphe, c'est-à-dire de supprimer les noeuds ou les chemins qui ne sont jamais parcourus lors d'une exécution du programme.

Nous illustrons à présent cette méthode d'analyse par le cas pratique de la suppression des protections *control flow flattening*, qui se traduisent par un aplatissement du graphe CFG.

**Suppression des protections control flow flattening** [UDM05, MVD06b]. La figure 3.6 présente le graphe CFG d'une fonction effectuant le calcul $F(4) = 4!$. Ce programme est obfusqué par application de la transformation d'aplatissement (*flattening*) du flot de contrôle [WDHK01]. Cette transformation est implémentée dans la solution CSS. La figure 3.7 présente une application basique de cette transformation et la figure 3.8 présente cette même transformation avec introduction de *junk code* et utilisation de pointeurs.

Cette transformation introduit des chemins d'exécution factices pour contrecarrer l'analyse statique. Les informations parasites vont se propager le long de ces chemins lors de l'analyse. L'information du flot de donnée factice se mêle à l'information du flot de donnée d'origine et a pour effet d'introduire de l'imprécision dans le résultat de l'analyse du programme.

Par exemple, lorsque plusieurs arcs partent d'un même *basic block* $S$, une analyse *backward* du flot de donnée (cf. définition 3.3.3), comme par exemple une analyse *liveness* (cf. définition 3.3.11), est affectée à la sortie de ce bloc $S$. Lorsque plusieurs arcs convergent vers un même *basic block* $S$, une analyse *forward* (cf. définition 3.3.3), comme par exemple l'analyse *reaching definition* (cf. definition 3.3.4) est affectée à l'entrée de ce bloc $S$.

Une première méthode pour adresser ce problème consiste à séparer les flot de données, en dédoublant le *basic block* $S$ vers lequel converge les arcs. Cette transformation a pour effet de diviser le flot de données et d'améliorer l'analyse *forward* du flot de donnée (en entrée du bloc $S'$). Remarquons qu'elle a aussi pour effet d'augmenter le nombre d'arcs et de noeuds du graphe CFG. Une analyse par propagation de la constante (cf. définition 3.3.7) permet ensuite de supprimer la plupart des arcs et noeuds factices. Le résultat peut être ensuite affiné par une analyse *liveness* et par une suppression des noeuds inutiles.

Une analyse statiques des chemins d'exécutions acycliques possibles permet de supprimer les chemins d'exécutions irréalisables. Soit $\pi$ un chemin d'exécution acyclique, notons $\bar{x}$ l'ensemble des variables vivantes à l'entrée du chemin $\pi$. Cette analyse se fonde sur la construction d'une contrainte $C_\pi$ telle que l'expression boolénne $\exists \bar{x}\, C_\pi$ n'est pas satisfaisable si et seulement si le chemin $\pi$ n'est jamais exécuté, lorsque l'on considère toutes les exécutions possibles du programme. Notons $i_k$ les instructions du programme, $x_0$ la valeur d'une variable $x$ au début du chemin $\pi$ et $x_k$ sa valeur imédiatement après exécution de l'instruction $i_k$. Nous notons $\perp$ une valeur inconnue de la variable $x$. La contrainte $C_\pi$ est construite comme conjonction des contraintes $C_k$ au niveau de chaque instruction $i_k$ du chemin $\pi$ de la manière suivante :

- si l'instruction $i_k$ est une assignation $x \leftarrow y$, alors si $i_j$ est l'instruction du chemin $\pi$ correspondant à la définition la plus récente de la variable $y$ ($j = 0$ s'il n'y a pas de définition de $y$ avant l'instruction $i_k$ dans le chemin $\pi$), on définit la contrainte $C_k$ comme égale à $x_k = y_j$.
- si l'instruction $i_k$ est une assignation du type $x \leftarrow y \odot z$, alors si $i_j$ est l'instruction du chemin $\pi$ correspondant à la définition la plus récente de la variable $y$ et si $i_l$ est l'instruction du chemin $\pi$ correspondant à la définition la plus récente de la variable $z$, on définit la contrainte $C_k$ comme égale à $x_k = f_\odot(y_j, z_l)$, où $f_\odot$ exprime la sémantique de l'opération $\odot$. Si la sémantique de l'opération $\odot$ est inconnue, ou si soit $y = \perp$, soit $z = \perp$, alors on définit la contrainte $C_k$ comme égale à $x = \perp$.
- si l'instruction $i_k$ est une instruction de branchement du type "si $E$, alors goto $L$", où $E$ est une expression booléenne, alors la contrainte $C_k$ vaut $E$ si la branche prise est dans $\pi$, et vaut $\neg E$ dans le cas contraire.

Une fois la contrainte $C_\pi$ construite (comme conjonction des contraintes $C_k$), on utilise un algorithme de résolution de contrainte pour statuer de la satisfaisablilité de l'expression booléenne $\exists \bar{x}\, C_\pi$.

Ces deux méthodes d'analyse statique peuvent être utilisées conjointement, et donnent de bon résultats lorsqu'il d'agit de défaire la transformation d'aplatissement dans sa forme basique. Ces méthodes purement statiques échouent cependant lorsqu'il s'agit de défaire cette même transformation avec introduction de *junk code* et utilisation de pointeurs.

L'idée est d'utiliser ces méthodes d'analyse statique pour améliorer le résultat de l'analyse dynamique. Sachant quels sont les arcs du flot de contrôle qui sont empruntés lors de l'exécution, il est possible de les marquer et de propager l'information du flot de donnée uniquement le long des arcs marqués. Cette approche permet de réduire l'imprécision résultant de la propagation d'information le long de chemins qui ne sont pas exécutables.
Il est possible ensuite d'effectuer une analyse par propagation de la constante (cf. définition 3.3.7) le long des chemins marqués. Lorsqu'un branchement conditionnel est rencontré lors de l'analyse par propagation de la constante où une seule des deux branches est empruntée lors de l'exécution, et que la branche de sortie ne peut pas être uniquement déterminée par propagation de la constante, l'arc correspondant est ajouté et marqué.
Cette approche permet d'élaguer l'information du flot de donnée propagée dans les blocs $S$. Cette dernière méthode a l'inconvénient d'être approximative. Notons $e(\mathcal{O}(P)) = \#\{e \mid e \in \mathcal{O}(P)\}$ le nombre d'arc du programme obfusqué. Deux types d'erreurs peuvent se produire :

- le programme désobfusqué $\mathcal{D}(\mathcal{O}(P))$ peut contenir des arcs qui n'existent pas dans le programme $P$ d'origine. Notons

$$\Delta_{over} = \{e \mid e \in \mathcal{D}(\mathcal{O}(P)) \text{ et } e \notin P\}$$

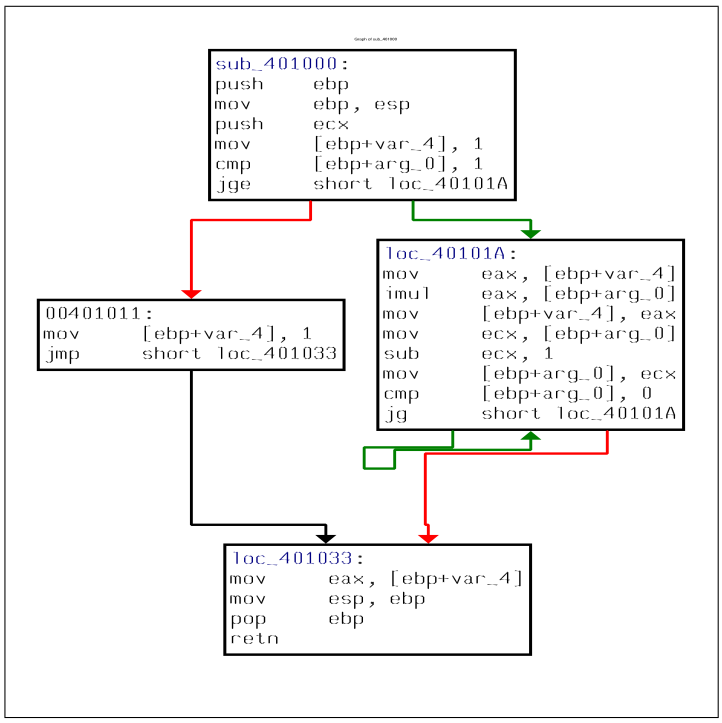



FIG. 3.6 – Arbre CFG d'un programme $F$ calculant 4 !

et

$$p_{over} = \frac{\Delta_{over}}{e(\mathcal{O}(P))}.$$

– le programme désobfusqué peut ne plus contenir des arcs qui existaient dans le programme $P$ d'origine. Notons

$$\Delta_{under} = \{e \mid e \notin \mathcal{D}(\mathcal{O}(P)) \text{ et } e \in P\}$$

et

$$p_{under} = \frac{\Delta_{under}}{e(\mathcal{O}(P))}.$$

L'erreur $p_{under}$ est souvent négligeable en pratique. En revanche, l'erreur $p_{over}$ peut être importante, et donc la désobfuscation insuffisante, lorsque l'on mesure la capacité du désobfuscateur à supprimer les arcs factices ajoutés par l'obfuscateur.

**Analyse de code protégé par virtualisation [GG09].** Nous avons présenté à la section 3.2.2 un mécanisme de protection fondé sur l'utilisation d'un processeur logiciel, appelé virtualisation de code. Ce mécanisme permet notamment l'application de transformations d'obfuscation sur une portion de code, après compilation, par réécriture de code binaire. Ce mécanisme peut en outre être vu comme un mécanisme d'altération de la section code de l'exécutable, dans la mesure où le code d'origine est traduit dans le jeu d'instruction du CPU logiciel de la protection avant d'être supprimé à son emplacement d'origine. Ces instructions sont émulées par la protection, par utilisation du processeur logiciel.
La résistance des transformations d'obfuscation aux transformations d'optimisation conditionne clairement la résistance du mécanisme de virtualisation de code à l'analyse hybride statique-dynamique.

Une méthode d'analyse hybride statique-dynamique a été appliquée récemment à ce mécanisme de protection [GG09]. Elle se fonde sur des extensions apportées à l'outil Metasm [Gui07], permettant :

– de réécrire le graphe de flot, en substituant un bloc d'instruction à un ou plusieurs blocs ;

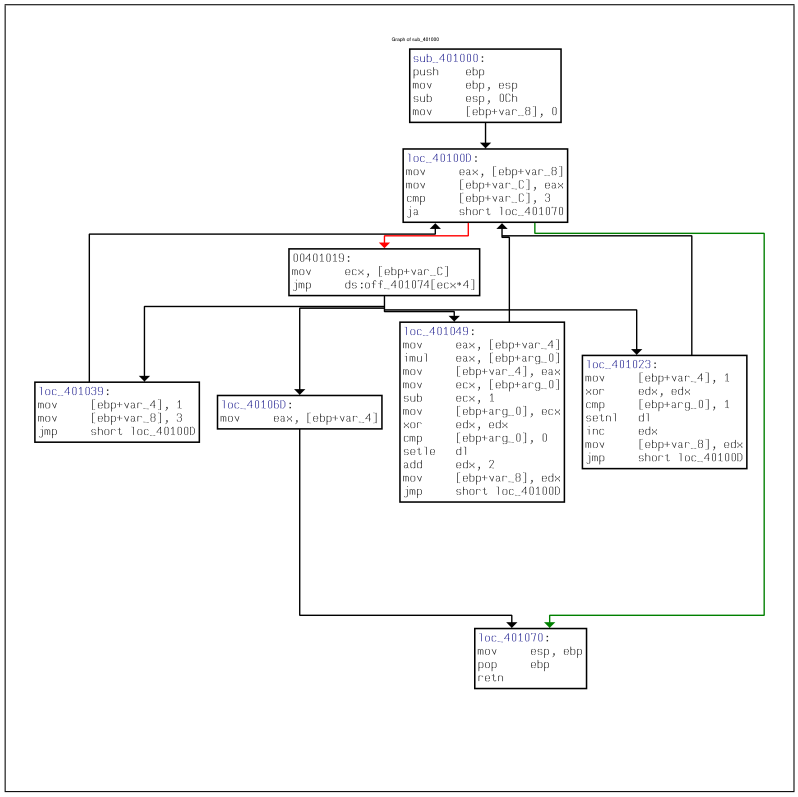



FIG. 3.7 – control flow flattening du programme $F$

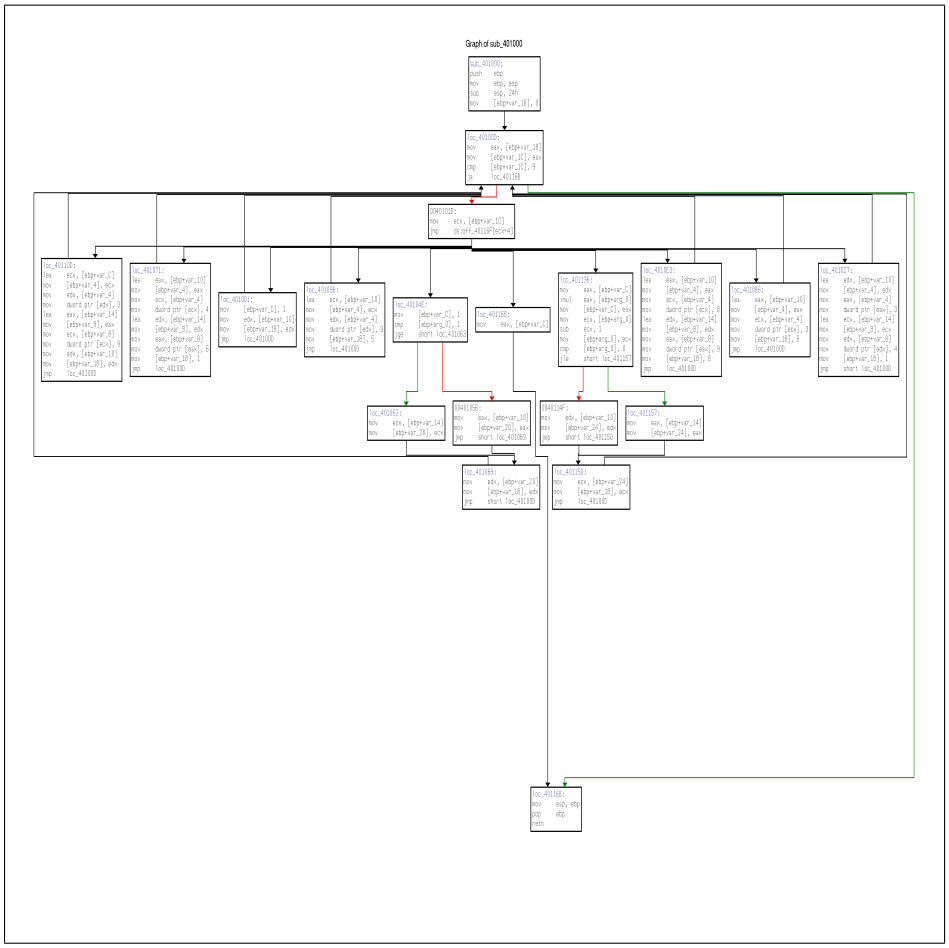



FIG. 3.8 – control flow flattening avec *junk code* et prédicats opaques

– de récupérer une information sur les effets d'une portion de code sur les registres du CPU et la mémoire.

L'analyse de la protection commence par l'application de transformations d'optimisation (réduction du flot de contrôle, analyse du flot de données, nettoyage de la pile, etc.) au code des gestionnaires de code virtuel. Le module correspondant réécrit progressivement le code, par application de règles de réécriture à une représentation des instructions désassemblées (objets de type `Instruction` de Metasm). Les différentes méthodes d'optimisation sont appliquées de manière itérative.
Les auteurs ont appliqué cette méthode à la solution de protection logicielle VMProtect. L'application des transformations d'optimisation permet de normaliser complètement le code des gestionnaires d'instructions virtuelles, simplifiant considérablement la suite de l'analyse de la protection.
A l'issue de l'application d'une méthode d'optimisation, une représentation intermédiaire du code est extraite (méthode `code_binding`). Cette représentation donne les valeurs de sortie en fonction des valeurs d'entrée de la portion de code considérée.
La représentation extraite représente l'interpréteur du bytecode utilisé par la machine virtuelle.

L'objectif de l'étape suivante est, disposant d'un interpréteur du bytecode écrit dans un langage intermédiaire, de compiler le bytecode vers le langage assembleur x86, reconnu par le processeur hôte.
Cette compilation est effectuée en appliquant une méthode d'exécution symbolique : l'exécution de chaque gestionnaire de code virtuel (opcode de la machine virtuelle), sous sa forme intermédiaire, permet la mise à jour d'un contexte symbolique proposant une vue partielle du processeur hôte.
Cette phase de spécialisation d'un gestionnaire de code virtuel permet la traduction d'une instruction virtuelle en une instructions exprimées en assembleur x86. A l'issue de cette étape, nous avons procédé à la spécialisation de l'interpréteur.

Le code assembleur obtenu est compilé puis optimisé. Pour appliquer les transformations d'optimisation, le CPU logiciel de Metasm est étendu (manuellement) aux registres virtuels (mapping des registres natifs). Cette extension permet d'abstraire les registres de la machine virtuelle dans le désassembleur de Metasm. Après avoir injecté les registres virtuels dans les objets `Instruction` de Metasm, les transformations d'optimisation s'appliquent de manière transparente aux registres virtuels.
A l'issue des transformations d'optimisation, les mouvements ou opérations inutiles ont disparu, rendant le code très lisible. Nous obtenons un code équivalent à celui avant application du mécanisme de virtualisation de code, ce de manière générique pour la solution de protection logicielle considérée (c'est-à-dire pour n'importe quelle instance de la protection).

Ce travail illustre bien la force de l'approche hybride statique/dynamique, pour lever les protections logicielles fondées sur l'obfuscation de code.

### 3.3.2 Instrumentation du contexte d'exécution

Lorsque l'on a levé la protection d'un programme, le programme déprotégé peut être selon les cas exécuté dans n'importe quel environnement, ou au contraire nécessiter un environnement d'exécution modifié (modifications au niveau du système d'exploitation, programme d'instrumentation dynamique devant être attaché au programme à chaque fois qu'il est lancé).
Dans tous les cas, il est nécessaire, lors de la phase d'analyse, d'instrumenter l'environnement d'exécution du programme observé, de manière à récupérer de l'information ou contourner sélectivement les protections.
Les méthodes d'instrumentation de l'environnement doivent être furtives, de manière à ne pas trahir la présence d'un environnement d'analyse aux yeux du programme protégé. Techniquement, cela impose la mise en place des codes d'instrumentation au plus bas niveau possible dans le système hôte : au niveau des API du système d'exploitation, au niveau des pilotes de périphériques, au niveau du matériel dans le cas de l'utilisation d'un composant matériel spécialisé ou dans celui de l'utilisation d'une machine virtuelle.
Remarquons que la conjecture qui stipule que plus le système d'instrumentation est bas niveau plus celui-ci est difficile à détecter n'est pas toujours vérifiée : nous présentons au chapitre 6 des méthodes de détection d'un programme furtif fondé sur l'utilisation d'un hyperviseur de machine virtuelle. Parmi celles-ci, les méthodes de contrôle de modèle (telle que celle proposée par C. Lauradoux [Lau08]) permettent de détecter la présence d'un

hyperviseur. Nous verrons au chapitre 6 que même l'utilisation d'un CPU complètement logiciel (émulateur) peut potentiellement être détectée [Fer06]. Dans tous les cas, le système de virtualisation utilisé doit être spécialisé, de manière à rendre inopérantes les fonctions de détection implémentées par la protection [Jos07a].

### 3.3.3 Unpacking d'exécutable et analyse de code auto-modifiable

Nous verrons au chapitre 5 que l'analyse dynamique permet de supprimer aisément le loader de protection d'un exécutable, dès lors que celui-ci n'est pas suffisamment évolué [Jos06b].
Cependant, observons que l'analyste de sécurité peut être intéressé par le fonctionnement du loader de protection lui-même. Par ailleurs, les méthodes d'analyse dynamique permettant de supprimer le loader de protection d'un exécutable de manière générique, c'est-à-dire sans exploiter leur connaissance du packer utilisé, ne parviennent pas à retrouver le point d'entrée original du programme lorsque ce dernier est lui-même auto-modifiable. L'analyste est alors contraint de retrouver ce point d'entrée manuellement, en croisant plusieurs sources d'informations sur le programme (connaissance du fonctionnement, trace de ses interactions avec les API du système d'exploitation). Le problème réside dans le fait que l'outil d'analyse est alors incapable de distinguer le flot de contrôle du programme de celui de la protection.
Pour mener cette analyse manuelle, l'évaluateur doit pouvoir s'appuyer sur une représentation sous forme de graphe du flot de contrôle de l'application et de son loader de protection.
Il doit s'agir d'une représentation plus complète du flot de contrôle permettant par exemple de raisonner sur le flot de données d'un programme comportant du code auto-modifiable. Une telle représentation a été proposée dans [AMD06] et appelée *State-Enhanced* CFG (SE-CFG). Constatant que la représentation sous forme de graphe CFG traditionnelle ne permet pas de mener une analyse statique suffisament précise du flot de données, les auteurs ont proposé une représentation étendue du graphe de flot. Ces extensions sont :

- une structure permettant de conserver une trace des états de chaque octet de la section statique de code du programme et du code généré en mémoire dynamiquement allouée ;
- des arcs conditionnels à l'état de la zone mémoire destination. Le graphe SE-CFG contient donc une information supplémentaire permettant de faciliter une analyse sensible au chemin d'exécution, en prenant en compte certains chemins non exécutables ;
- la représentation sous forme de donnée d'une instruction : une instruction utilise la zone mémoire (les octets de code) à laquelle elle se trouve. Cette extension permet de traiter le code comme des données dans l'analyse du flot de données.

Ces extensions permettent de s'affranchir de l'hypothèse d'un code constant. Le graphe SE-CFG peut être linéarisé, c'est-à-dire que partant de la seule information du graphe SE-CFG d'un programme comportant du code auto-modifiable, il est possible de reconstruire la section de code d'origine. Le graphe SE-CFG contient une information supplémentaire permettant d'améliorer la précision de l'analyse du flot de données, et donc d'améliorer le processus de normalisation de cette représentation intermédiaire.

Nous avons présenté dans cette section les problèmes théoriques et pratiques rencontrés lors d'une tentative de lever la protection d'une application, ainsi que les principales méthodes d'analyse pouvant être mises en oeuvre par un attaquant. Nous en déduisons des critères théoriques et empiriques de résistance des mécanismes.

## 3.4 Évaluation d'une solution de protection

L'évaluation d'une solution de protection logicielle impose de définir des critères objectifs (théoriques et pratiques) de qualité sur les mécanismes de protection et sur la manière dont ils sont agencés. Rappelons que lors d'une analyse de robustesse des mécanismes, l'analyste adopte le point de vue de l'attaquant.
Une partie des mécanismes de protection logicielle sont des mécanismes cryptographiques. Leur analyse impose donc a priori une cotation des mécanismes cryptographiques[6]. Or, la propriété de résistance face à la rétro-ingénierie impose une étude spécifique de la résistance de l'architecture cryptographique vis-à-vis d'attaques non cryptographiques. Une telle analyse ne fait généralement pas partie du périmètre d'une cotation cryptographique, dans la mesure où la démonstration de résistance face à la rétro-ingénierie n'est pas explicitement mentionnée dans la portée

[6] dans le sens adopté dans le cadre des qualifications mises en oeuvre par la DCSSI. Voir les règles relatives à la qualification des produits de sécurité par la DCSSI, version 1.1 du 26 Février 2004, consultables en ligne à l'adresse http ://www.ssi.gouv.fr/fr/politique_produit/catalogue/Qualification.html.

de l'analyse. L'analyse d'efficacité des mécanismes cryptographiques est principalement fondée sur l'analyse cryptographique du schéma de conception et l'utilisation de tests statistiques. Dans le cadre d'une évaluation fondée sur les Critères Communs, une étude de l'efficacité des mécanismes vis-à-vis d'attaques non cryptographiques et donc notamment de résistance face à la rétro-ingénierie, est plus importante.

Pour traiter la problématique de la résistance à la rétro-ingénierie dans le contexte WBAC, plusieurs modèles concurents ont été proposés dans la littérature, pour formaliser les notions d'obfuscation et de désobfuscation, et définir un modèle de l'attaquant. Nous présentons dans la section 3.4.1 les modèles issus de la théorie de la complexité, hérités pour nombre d'entre eux des résultats théoriques de l'optimisation des programmes. Nous avons vu dans le chapitre 2 les résultats utiles à l'analyse de primitives cryptographiques en contexte boîte blanche. Ces modèles ont donné naissance à des critères de sécurité théoriques et empiriques sur la plupart des mécanismes de protection logicielle que nous avons présentés, et sont utilisés par les concepteurs de nouveaux mécanismes de protection pour justifier de leur pertinence. Nous présentons certains de ces critères dans la section 3.4.2. Nous proposons enfin une démarche (section 3.4.3) et présentons des outils (section 3.4.4) permettant d'assister l'évaluateur dans son analyse d'efficacité des mécanismes.

### 3.4.1 Modèles issus de la théorie de la complexité

Nous avons vu au chapitre 2 une définition formelle de l'obfuscation qui prenait tout son sens dans le cadre de la cryptographie boîte blanche. Appliquée à la formalisation des transformations d'obfuscation non cryptographiques, ce formalisme et le résultat d'impossibilité qui y est attaché paraissent de peu d'utilité en pratique.
Concernant les mécanismes d'obfuscation non cryptographiques, nous pouvons observer que la sécurité de ces mécanismes est souvent justifiée par un argument de complexité, comme par exemple le caractère difficile de l'analyse du résultat produit par application d'une transformation du graphe PCG. L'efficacité de ces mécanismes est le plus souvent justifiée par le fait d'embarquer un ou plusieurs problèmes difficiles de l'optimisation de programme.
Les compilateurs éludent ce problème en résolvant des approximations conservatives de ces problèmes. Cependant, ces approximations conservatives ne peuvent jamais être complètes, et les compilateurs choisissent dans ce cas de laisser le programme partiellement non-optimisé.

Cependant, même si un problème difficile est embarqué, il n'est pas évident que la désobfuscation soit difficile. De manière générale, concernant les techniques d'obfuscation dont la sécurité repose sur la difficulté à résoudre des problèmes difficiles, au regard de la théorie de la complexité, il n'est pas immédiat qu'un attaquant doive nécessairement résoudre le problème difficile sous-jacent pour atteindre ses objectifs. Un attaquant humain ne doit par exemple pas nécessairement défaire toutes les transformations d'obfuscation pour localiser et inhiber une fonction de contrôle des droits (analyse interactive). En outre, nous avons vu qu'une analyse dynamique ou hybride statique dynamique [MADD05] permet de défaire automatiquement des transformations d'obfuscation que l'analyse statique va conserver.

Concernant l'utilisation d'arguments issus de la théorie de la complexité pour établir une preuve de la sécurité d'un mécanisme cryptographique, nous pouvons également faire l'observation suivante : même si les problèmes NP-complets sont généralement considérés intraitables, cette difficulté ne rend finalement compte que du pire cas. En effet, certains problèmes sont faciles en moyenne et sont résolubles en temps polynomial avec des algorithmes d'approximation fournissant une solution très proche de l'optimale. Des réductions plus complexes du type connection cas moyen - pire cas [AD97] ou encore des réductions de problèmes réputés difficile en moyenne au mécanisme cryptographique considéré doivent alors être envisagées.

La définition de la notion de désobfuscation est tout aussi controversée : de manière informelle, on peut définir simplement un désobfuscateur comme un algorithme capable, partant d'un programme $\mathcal{O}(P)$, de trouver un programme moins complexe et fonctionnellement équivalent. Les transformations de désobfuscation implémentées par un tel programme sont alors des transformations d'optimisation, visant à réduire la complexité générale du code au regard de métriques appropriées (voir [CTL97, page 8] pour des exemples de métriques).
Il est possible d'imposer une propriété plus forte sur le programme résultant, en stipulant par exemple que les transformations de désobfuscation doivent converger vers une version normalisée du programme, unique à un isomorphisme de graphe PCG près. Dans ce modèle, le problème de l'arrêt est clairement réductible au problème

général de la désobfuscation et est donc indécidable [BM08, BKM08].

### 3.4.2 Critères d'analyse

Evaluer la robustesse d'une transformation d'obfuscation n'est pas une tâche aisée. Il n'y a pas de métrique unifiée qui apporte une réponse univoque à la question : quelle est la robustesse de cette transformation d'obfuscation ? De nombreuses transformations d'obfuscation sont très utiles, même s'il n'est pas toujours possible de prouver leur pertinence au regard de la théorie de la complexité. Des critères plus souples permettent de décrire et de classifier ces transformations. Ces critères, même s'ils reposent sur des mesures de complexité, sont empiriques.
La notion de mesure de complexité est un concept général qui recouvre des notions plus intuitives comme le temps nécessaire à l'exécution d'un programme, la place mémoire nécessaire à cette exécution et la taille du programme. Plus formellement, on appelle mesure de complexité [Blu67] toute application $\mu$ de $\mathbb{N} \times \mathbb{N}$ dans $\mathcal{F}^*$ telle que $\forall (k, z) \in \mathbb{N} \times \mathbb{N}$, $\mu(k, z) = \mu_z^k \in \mathcal{F}^{k+1}$ et le prédicat $\mu_z^k(x) = y$ est décidable.
On trouve dans la littérature un certain nombre de critères fondés sur une ou plusieurs mesures de complexité, permettant de justifier de l'efficacité d'une transformation d'obfuscation ou plus prosaïquement d'étalonner un obfuscateur. Nous donnons ici quelques uns de ces critères, tirés de [CTL97]. Il est ainsi possible de quantifier la résistance imposée à une analyse humaine par la quantité :

$$\mathcal{T}_{pot}(P) = \frac{c_\mu(\mathcal{O}(P))}{c_\mu(O)} - 1,$$

mesurer l'effort requis pour défaire une transformation d'obfuscation, en fonction de l'effort $\mathcal{T}_1$ à produire pour forger un outil de désobfuscation et de la classe de complexité $\mathcal{T}_2$ de l'analyse :

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

où $R$ est une matrice de cotation définie par le concepteur. On peut mesurer le coût en terme de ressources (temps CPU/espace mémoire) induit par la protection : $\mathcal{T}_{cost}(P)$. Il est possible de définir un autre critère, sur la base de l'observation suivante : si une transformation introduit du code nouveau dans le programme protégé $\mathcal{T}(P)$, un attaquant pourra facilement le détecter et exploiter cette faiblesse pour lever la protection. Définissons donc un ensemble de caractéristiques du langage utilisé par un programme, $\mathcal{L}(P)$. Notons $\mathcal{L}(\mathcal{O})$ l'ensemble des caractéristiques du langage introduites par la transformation d'obfuscation $\mathcal{O}$. La furtivité de la transformation $\mathcal{O}$ relativement au programme $P$ peut être définie par :

$$\mathcal{T}_{ste}(P) = 1 - \frac{|\mathcal{L}(\mathcal{O}) - \mathcal{L}(P)|}{|\mathcal{L}(\mathcal{O})|}$$

Si la transformation n'enrichit pas le langage des programmes qu'elle protège, la furtivité $\mathcal{T}_{ste}(P)$ est maximale et égale à *1*. Il est finalement possible de considérer l'ensemble de ces critères simultanément, en définissant la qualité d'une transformation par :

$$\mathcal{T}_{app}(P) = \frac{\omega_1 \mathcal{T}_{pot}(P) + \omega_2 \mathcal{T}_{res}(P) + \omega_3 \mathcal{T}_{ste}(P)}{\mathcal{T}_{cost}(P)}$$

où $\omega_i,\ i = 1, 2, 3$ sont des constantes fixées en fonction du contexte opérationnel d'utilisation des transformations d'obfuscation.
Ces critères, même s'ils reposent sur des mesures de complexité, sont empiriques, et sont plus utiles à l'étalonnage d'un obfuscateur qu'à la preuve de son efficacité face à la rétro-ingénierie.
Remarquons que dans le contexte de la détection virale, le critère de résistance $\mathcal{T}_{res}(P)$ peut être vu de manière informelle comme la difficulté à détecter un programme métamorphe [BM08].

### 3.4.3 Démarche d'analyse

Avant d'exposer notre démarche d'analyse [DJ07], nous pouvons faire l'observation suivante relative à la définition d'une attaque effective et au modèle de l'attaquant : Le fait pour un système de protection de reposer sur des mécanismes dont la sécurité est prouvée vis-à-vis d'une attaque statique exacte ou conservative n'est pas suffisant. Si certaines attaques approximatives sont effectives, la sécurité en confidentialité des données critiques peut être compromise complètement.

Un système de protection logicielle est un produit de sécurité comme un autre. Sa sécurité peut donc être évaluée suivant la même démarche. Les mécanismes de sécurité mis en oeuvre ont cependant été moins étudiés que les mécanismes cryptographiques traditionnels. Les règles concernant le dimensionnement des mécanismes et la propagation de la confiance au sein de telles architectures reste donc à définir[7].

Cette problématique a été éprouvée dans le cadre d'une analyse de sécurité de produits DRM du marché civil et dans le cadre d'une étude de robustesse de mécanismes de protection logicielles. Quelques critères et repères méthodologiques semblent émerger.
En se basant sur le fait que toute solution de protection logicielle se décompose en un ensemble de mécanismes élémentaires, nous avons élaboré une démarche permettant de déduire d'une façon rigoureuse les objectifs de sécurité, vulnérabilités et hypothèses de sécurité d'une solution à partir des objectifs de sécurité, vulnérabilités et hypothèses de sécurité des mécanismes élémentaires constituant cette solution. Nous présentons succintement son principe ici. Pour une large gamme de mécanismes élémentaires (anti-dump, anti-déboguage, obfuscation, contrôle d'intégrité, chiffrement) nous décrivons pour chaque mécanisme :

- ses spécifications, exprimées sous la forme d'un ensemble de données participant à sa mise en oeuvre ou à son exécution ;
- ses contraintes d'intégration et d'exploitation, exprimées sous la forme d'un ensemble d'exigences sur l'environnement de mise en oeuvre et d'exécution du mécanisme ;
- ses objectifs de sécurité, exprimés sous la forme d'une ou plusieurs exigences de sécurité satisfaites lorsque le mécanisme est mis en oeuvre ;
- ses vulnérabilités, exprimées sous la forme d'un ensemble des modes de contournement du mécanisme conduisant à transgresser ses objectifs de sécurité ;
- ses hypothèses de sécurité, exprimées sous la forme d'un ensemble d'exigences que doit satisfaire le mécanisme pour qu'il respecte ses objectifs de sécurité ;
- ses degrés de liberté, exprimés sous la forme d'un ensemble des données pour lesquelles les objectifs et hypothèses de sécurité du mécanisme restent inchangées.

Sur la base de cette description, nous pouvons représenter les objectifs et hypothèses de sécurité d'un mécanisme à l'aide des caractéristiques suivantes : résistance à l'analyse statique, à l'analyse dynamique, à la modification de données (ou de code), diversification et couplage entre une section de code et l'ensemble de l'application protégée.

Partant de l'analyse des hypothèses de sécurité d'une solution, nous pouvons alors évaluer le niveau de sécurité de cette solution. Cette démarche s'adapte par ailleurs facilement à l'introduction de mécanismes supplémentaires dans une solution de protection logicielle sur étagère et permet de maîtriser l'apport en terme de sécurité de cette introduction dans la solution existante.

### 3.4.4 Outils d'analyse

Une analyse de sécurité ne peut pas se dispenser de tests intrusifs, visant à éprouver la sécurité d'une application en exploitant ses failles pour lever la protection, c'est à dire dans notre contexte parvenir à exhiber les algorithmes propriétaires d'un programme ou parvenir à contourner une fonction de contrôle des droits. Nous avons vu que l'analyse hybride statique-dynamique est probablement la méthode la plus efficace en pratique. L'analyse manuelle d'un programme protégé nécessite la maîtrise de plusieurs types d'outils spécialisés :

- instrumentation du contexte d'exécution,
- analyse statique,
- analyse dynamique.

#### Instrumentation du contexte d'exécution

Concernant l'instrumentation du contexte d'exécution, de très nombreux outils sont disponibles. Les kits de développement en proposent pour chaque plate-forme d'exécution (outils Windows Driver Devel Kit [BB97] par exemple). Les outils de diagnostic système peuvent également être utilisés (outils MS. SysInternals [RS05] par exemple).

[7] A titre d'exemple, les contraintes liées à l'implémentation de primitives cryptographiques dans un contexte boîte blanche ne sont pas prises en compte dans le référentiel documentaire de la DCSSI (règles et recommandations concernant la gestion des clés et le dimensionnement des primitives cryptographiques). Voir http ://www.ssi.gouv.fr/fr/sciences/publications.html.

En outre, certains outils d'analyse dynamique spécialisés proposent des mécanismes d'instrumentation furtive de l'environnement d'exécution [BH99, BKK06, Jos07b].

#### Analyse statique

De très nombreux outils d'analyse statique dédiés à la vérification ou à l'optimisation de programme ont été développés ces vingt dernières années, et peuvent potentiellement servir de support à la rétro-ingénierie. L'ambition de ce chapitre n'est pas de les présenter. Citons simplement une suite logicielle développée récemment et permettant d'éprouver par la pratique les quelques concepts que nous avons évoqués dans ce papier. Loco [MVD06a], est un outil permettant d'implémenter à la fois des transformations d'obfuscation et de désobfuscation. Il s'appuie sur l'outil Diablo [DVCD07], un éditeur de liens (édition statique de liens uniquement) particulier, permettant de réécrire un exécutable ELF après avoir effectué des modifications sur son graphe PCG (*link-time binary rewriting framework*). Ces deux outils s'appuient sur une interface graphique appelée Lancet [VDM+05]. Nous pouvons également citer la suite logicielle SOLAR [SOL07], qui comprend des systèmes d'optimisation *link-time* pour trois architectures : ALTO (pour l'architecture Compaq Alpha), PLTO (pour l'architecture IA-32) et ILTO (pour l'architecture Intel/HP IA-64).

#### Analyse dynamique

Les outils d'analyse dynamique spécialisés pour la rétro-ingénierie logicielle sont plus rare. Certains débogueurs proposent des fonctionnalités intéressantes (script d'OllyDbg [Ope07] ou d'IDAPro [Dat07], par exemple), mais imposent à l'attaquant de laborieux efforts pour surmonter certaines protections logicielles. Les outils les plus efficaces pour récupérer de l'information sur un exécutable sans être détecté (et parfois lever automatiquement les premières protections) sont probablement les machines virtuelles spécialisées. Les outils CWSandbox [WHF07], Norman SandBox [Nor07], TTAnalyze [Bay05], Cobra [VY06] et VxStripper [Jos07b] permettent de charger un exécutable dans une machine virtuelle et de l'analyser automatiquement.
L'outil d'analyse VxStripper permet en outre de supprimer automatiquement le loader de protection mis en place par la pluspart des packers du marché et de défaire automatiquement certaines transformations d'obfuscation. Il propose également des fonctionnalités avancées d'instrumentation du contexte d'exécution et d'analyse forensique du système d'exploitation, permettant de surveiller les interactions du programme cible avec celui-ci.

## 3.5 Conclusion

Nous avons présenté dans ce chapitre les caractéristiques, en terme d'architecture, des solutions de protection logicielle du marché civil et décrit les mécanismes fondamentaux qu'elles implémentent.
Nous avons présenté les problèmes théoriques et pratiques rencontrés lors d'une tentative de lever la protection d'une application, ainsi que les principales méthodes d'analyse pouvant être mises en oeuvre par un attaquant.
Nous avons proposé une démarche et des outils permettant de mener à bien une analyse de robustesse des mécanismes, sur la base de critères théoriques et empiriques. Une telle approche peut être utilisée par un évaluateur, pour mesurer le niveau de confiance qu'il peut porter à une solution ; par le concepteur de solution, afin d'en améliorer l'efficacité ; ou par l'utilisateur, afin d'intégrer une ou plusieurs solutions et éventuellement développer des protections supplémentaires.

Nous avons observé lors de cette introduction (aux aspects techniques et théoriques liés à la protection et à la rétro-ingénierie logicielle au regard de la théorie de la complexité et de l'optimisation des programmes) que les critères et preuves de sécurité des mécanismes de protection semblent mal adaptés à la mesure de résistance face à une attaque approximative, dynamique ou hybride statique/dynamique. Nous avons observé en outre que si l'analyse statique et dynamique sont clairement duales, elle apparaissent souvent complémentaires. L'utilisation conjointe de méthodes statiques et de méthodes dynamique donne de meilleurs résultats que l'utilisation seule d'une des deux approches.
Nous avons en outre remarqué que les méthodes existantes d'analyse statique et dynamiques présentent souvent une approche technique similaire. Il semble même qu'elles se heurtent souvent aux même limitations. Nous avons en particulier vu l'importance des modèles de contrôle de flot et de données d'un programme en analyse dynamique. Ces modèles ont d'abord été conçus pour l'analyse statique des programmes et correspondent à un important savoir

faire hérité des travaux portants sur l'optimisation et la vérification statique de programme. Ce savoir-faire se révèle fort utile dans le cadre d'une activité visant à augmenter la lisibilité d'un programme, comme c'est le cas pour la rétro-ingénierie. Les difficultés technique et les limites théoriques rencontrées servent de base dans la construction de programme difficile à comprendre ou à rétro-analyser, et sont donc des fondamentaux de la protection logicielle.

Les méthodes d'analyse statiques et dynamiques font encore aujourd'hui l'objet de travaux de recherches séparés. Dans le cadre de la rétro-ingénierie, cette séparation paraît artificielle et à certains égards contre-productive. Il apparaît nécessaire de supprimer la frontière entre ces deux spécialités et de concevoir des méthodes d'analyse unifiées qui puissent soit opérer dans les deux modes, soit opérer dans un mode qui tire partie au mieux des deux approches.

Nous présentons dans le chapitre suivant un des modèles fondamentaux de l'analyse statique, celui de l'interprétation abstraite. Ce modèle permet de formaliser la plupart des méthodes d'analyse statique comme une abstraction de l'ensemble des exécutions possibles d'un programme. Remarquons que ce qui réunit le plus l'analyse statique et l'analyse dynamique est que dans les deux cas, elles proposent un modèle de l'ensemble des chemins d'exécution d'un programme, qui leurs est propre et supporte une classe spécifique de questionnements et de raisonnement. Toutes ces techniques présentent une abstraction du programme sous la forme d'un modèle comportant généralement un nombre fini d'états.
Nous proposons dans le cadre formel de l'interprétation abstraite une formalisation de la rétro-ingénierie comme approximation discrète des sémantiques concrètes d'un programme. L'objectif est de poser les premières bases d'une reflexion permettant de définir des critères objectifs permettant de qualifier les transformations de désobfuscation dans le contexte de l'analyse statique et de l'étendre au cas de l'analyse dynamique.

# Deuxième partie

# Lutte anti-virale

Chapitre 4

# Modélisation de la rétro-ingénierie par interprétation abstraite

Nous avons étudié dans la première partie de cette thèse la menace virale, à travers :
- la présentation les principaux modèles issus de la théorie de la complexité s'appliquant à la description formelle des virus ;
- l'étude des mécanismes cryptographiques susceptibles d'être utilisés par un virus pour résister à l'analyse dynamique ;
- l'étude des mécanismes non cryptographiques susceptibles d'être utilisés par un virus pour asseoir la robustesse des mécanismes cryptographiques qu'il implémente et parer la rétro-ingénierie.

Nous allons à présent étudier les moyens dont dispose la lutte anti-virale pour parer cette menace. Nous présentons en premier lieu le cadre formel de l'interprétation abstraite et proposons une formalisation de la rétro-ingénierie dans ce cadre.

## 4.1 Introduction

Ce chapitre propose une formalisation de la rétro-ingénierie comme approximation discrète des sémantiques concrètes d'un programme. Nous souhaitons pouvoir disposer d'un outil formel permettant de hiérarchiser les informations sémantiques extraites d'un programme au regard d'une classe de mécanismes d'obfuscation et d'assurer la correction et la complétude des transformations de désobfuscation.

Il est possible de formaliser en terme d'interprétation abstraite les transformations d'analyse statique et d'optimisation des programmes [CC77a, CC77b, CC79], mais les notions d'obfuscation [DPG05] et de désobfuscation ne sont pas encore convenablement décrites dans ce modèle. En outre, l'analyse dynamique et hybride statique/dynamique n'est pas bien représentée dans ce formalisme.
Je pose dans ce chapitre les premières bases d'une réflexion permettant de définir des critères objectifs permettant de qualifier les transformations de désobfuscation dans le contexte de l'analyse statique et de l'étendre au cas de l'analyse dynamique.

Je propose une définition alternative de l'obfuscation [DPG05] dans le cadre de l'interprétation abstraite et une caractérisation de la désobfuscation en terme de correction et de complétude au regard d'une classe de transformations d'obfuscation ainsi qu'une étude des relations entre :
- la caractérisation d'une transformation de désobfuscation en terme de correction et de complétude vis-à-vis d'une classe de transformations d'obfuscation ;
- ces mêmes notions appliquées à la caractérisation d'un moteur de détection viral [CJS+05].

Ce modèle va nous être précieux pour qualifier l'information sémantique extraite lors du processus d'analyse d'un programme viral (voir section 8.3).

Ce modèle offre en outre la possibilité de hiérarchiser les transformations de désobfuscation au regard d'une classe identifiée de mécanismes d'obfuscation. Il est donc possible de prouver l'efficacité d'un désobfuscateur dans ce modèle. Il est enfin possible d'évaluer la correction et la complétude d'une composition de transformations de désobfuscation. Cette possibilité est précieuse lors de l'évaluation de la robustesse d'un moteur de détection en boîte blanche ou à partir des spécifications détaillées d'un produit antivirus.

Le reste de ce chapitre est organisé de la manière suivante :
- la section 4.2 présente et discute la première formalisation de l'obfuscation en terme de tranformation de programme proposée dans [DPG05]. J'y propose une définition alternative de l'obfuscation dans ce cadre et une caractérisation de la désobfuscation en terme de correction et de complétude au regard d'une classe de transformations d'obfuscation ;
- la section 4.3 présente la notion de schéma de détection dans ce cadre formel et démontre que la caractérisation d'une transformation de désobfuscation en terme de correction et de complétude vis-à-vis d'une classe de transformations d'obfuscation s'étends naturellement à ces mêmes notions appliquées à la caractérisation d'un moteur de détection viral [CJS+05].
- la section 4.4 présente la problématique de l'extension de ces critères à l'analyse dynamique dans le cadre formel de l'interprétation abstraite ;
- la section 4.5 présente mes conclusions concernant les limitations actuelles de cette approche et les axes de recherches futures que j'identifie.

J'ai placé en annexe D les rappels théoriques concernants l'interprétation abstraite [CC77a] et la formalisation des transformations de programmes dans ce cadre [CC02].

## 4.2 Formalisation du glaive et du bouclier par interprétation abstraite

Nous présentons dans cette section la première formalisation de l'obfuscation proposée dans [DPG05]. Celle ci se fonde sur le cadre formel de l'interprétation abstraite et des transformations de programmes, définies sur une base sémantique. Nous discutons de sa pertinence et proposons une définition alternative, plus proche de la définition proposée dans [BGI+01]. Nous proposons ensuite une formalisation de la désobfuscation dans ce cadre formel.

**Définition 4.2.1** *($\delta$-obfuscateur [DPG05]). Une transformation $\mathtt{t} : \mathbb{P} \to \mathbb{P}$ est un $\delta$-obfuscateur si :*
- *une partie des propriétés sémantiques $\varphi \in uco(\mathcal{P}(\Sigma^+))$ est préservée par la transformation $\mathtt{t}$. $\delta$ est alors défini comme la propriété la plus concrète de $uco(\mathcal{P}(\Sigma^+))$ préservée par la transformation $\mathtt{t}$ :*

$$\delta = \bigsqcap \{\varphi \in uco(\mathcal{P}(\Sigma^+)) \,|\, \forall P \in \mathbb{P},\, \varphi(S^+[\![P]\!]) = \varphi(S^+[\![\mathtt{t}[\![P]\!]]\!])\}$$

- *une partie des propriétés sémantiques $\varphi \in uco(\mathcal{P}(\Sigma^+))$ est masquée - i.e. non préservée - par la transformation $\mathtt{t}$ : $\varphi \ominus (\delta \sqcup \varphi) \neq \top$.*

Le principe de la $\delta$-obfuscation consiste à mesurer la perte d'information imposée par l'obfuscateur $\mathtt{t}$ à toute abstraction $\rho$ non préservée par $\mathtt{t}$. Le pseudo-complément $\rho \ominus (\rho \sqcup \rho_{\mathtt{t}})$ permet d'isoler l'information qui n'est pas préservée par $\mathtt{t}$[1]. Plus cette information est concrète, plus l'obfuscateur est robuste vis-à-vis de cette abstraction.

Cette caractérisation permet de comparer deux obfuscateur $\mathtt{t}$ et $\mathtt{t}'$ en terme de robustesse vis-à-vis d'une abstraction $\rho$ - non préservée par $\mathtt{t}$ : si $\rho \ominus (\rho \sqcup \rho_{\mathtt{t}}) \sqsubseteq \rho \ominus (\rho \sqcup \rho_{\mathtt{t}'})$, alors la transformation $\mathtt{t}$ est plus robuste que la transformation $\mathtt{t}'$ vis-à-vis de l'abstraction $\rho$.

Le concept de $\delta$-obfuscateur consiste donc à caractériser une transformation d'obfuscation par la propriété la plus concrète qu'elle préserve - i.e. qu'elle préserve une certaine abstraction des sémantiques du programme et que cette abstraction - est la meilleure dans le sens où elle - capture l'information la plus précise que le programme d'origine et le programme transformé ont en commun.

[1] La plus petite abstraction commune $\rho \sqcup \rho_{\mathtt{t}}$ représente ce que les deux domaines abstraits ont en commun. Le pseudo complément revient à soustraire cette partie commune de la propriété $\rho$ pour ne garder que ce qui les différencie, c'est à dire l'information qui a été supprimée par la transformation $\mathtt{t}$. Lorsqu'une propriété $\rho$ est préservée par la transformation $\mathtt{t}$, on a : $\rho_{\mathtt{t}} \sqsubseteq \rho$ et donc $\rho \ominus (\rho \sqcup \rho_{\mathtt{t}}) = \lambda x.\top$. En fait, $\rho \sqcup \rho_{\mathtt{t}} = \rho \Leftrightarrow \rho \in \uparrow \rho_{\mathtt{t}} = \{\rho \in uco(\mathcal{D}) \,|\, \rho_{\mathtt{t}} \sqsubseteq \rho\}$.

Nous pensons que cette définition est trop générale, et s'applique mieux à la caractérisation d'une transformation d'optimisation aggressive : en effet, avec cette définition, toute transformation - y compris les transformations d'optimisation - est une transformation d'obfuscation. En outre, on attend d'une transformation d'obfuscation qu'elle préserve les sémantiques dénotationnelles - i.e. le comportement observable - d'un programme.
En revanche, cette définition permet de modéliser correctement l'attaquant : une transformation d'optimisation peut être caractérisée par la propriété la plus concrète qu'elle préserve [CC02]. Un mécanisme d'obfuscation résistera à cette transformation si le domaine abstrait qui paramètre l'optimisation n'est pas assez precis, i.e. qu'il ne fournit pas toute l'information nécessaire. Inversement, le niveau de complétude du domaine abstrait requis pour pouvoir supprimer ce mécanisme donne une mesure du niveau de résistance de ce mécanisme en formalisant la quantité d'information nécessaire pour le casser.
Ces deux remarques me conduisent à proposer une formalisation alternative de l'obfuscation, donnée dans la suite de cette section.

### 4.2.1 Formalisation du bouclier

Nous attendons d'un obfuscateur vu comme une transformation syntaxique $\mathtt{t}$ sur les programmes qu'il préserve au minimum le comportement observable des programmes, en terme d'entrées/sorties, c'est-à-dire qu'il préserve l'observation $\alpha_d$ correspondant à la sémantique dénotationnelle :

$$\begin{tikzcd}
t(S[\![P]\!]) = S[\![\mathtt{t}[\![P]\!]]\!] \arrow[dr, "\alpha_d"] & \\
S[\![P]\!] \arrow[u, "t"] \arrow[r, "\alpha_d"] & \alpha_d(t(S[\![P]\!])) = \alpha_d(S[\![\mathtt{t}[\![P]\!]]\!) \arrow[d, equal] \\
& \alpha_d(S[\![P]\!])
\end{tikzcd}$$

Soit $\mathtt{t}$ une transformation syntaxique, on définit sur le treillis des interprétation abstraite $uco(\mathcal{D})$ le domaine le plus concret des domaines préservés par la transformation $\mathtt{t}$ :

$$\rho_{\mathtt{t}} = \sqcap\{\rho \in uco(\mathcal{D}) \,|\, \forall P \in \mathbb{P},\, \rho(S[\![\mathtt{t}[\![P]\!]]\!]) = \rho(S[\![P]\!])\}.$$

La transformation syntaxique $\mathtt{t}$ préserve le comportement observable des programmes si $\rho_{\mathtt{t}} \sqsubseteq \rho_d$, où $\rho_d = \gamma_d \circ \alpha_d$ est le domaine abstrait correspondant aux sémantiques dénotationnelles sur le treillis des interprétations abstraites $uco(\mathcal{D})$.

Nous pouvons énoncer la propriété caractérisant une transformation d'obfuscation de la manière suivante :

$$\forall \rho \in uco(\mathcal{D}),\, \rho_{\mathtt{t}} \sqsubseteq \rho \Rightarrow \rho_d \sqsubseteq \rho.$$

En d'autres termes, quelle que soit l'interprétation abstraite préservée par la transformation $\mathtt{t}$, elle n'apporte pas plus d'information que n'en apporte la simple observation des entrées/sorties. Nous retrouvons ici le paradigme de la boîte noire virtuelle, tel que proposé dans [BGI+01]. La formalisation de cette propriété dans le cadre de l'interprétation abstraite nous dote d'un outil formel permettant de comparer entre elles des transformations d'obfuscation, en terme de robustesse et de statuer sur la correction et la complétude d'une transformation d'obfuscation, dans le cadre des transformations de programmes et de l'interprétation abstraite.

**Définition 4.2.2** *(Obfuscateur sémantique idéal). Un obfuscateur $\mathtt{t}$ est défini comme une transformation vérifiant les propriétés suivantes :*

- *La transformation sémantique correspondante vérifie :*

$$\forall P \in \mathbb{P},\, t[S[\![P]\!]] = S[\![\mathtt{t}[\![\mathtt{p}(S[\![P]\!])]\!]]\!].$$

  *En d'autres termes, il s'agit d'un compilateur ou d'une transformation algorithmique ;*
- *La transformation $\mathtt{t}$ préserve la sémantique dénotationnelle des programmes :*

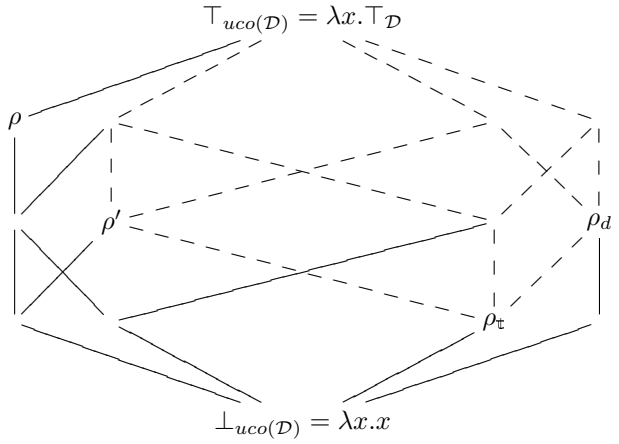
$$\forall P \in \mathbb{P},\, \alpha_d(t(S[\![P]\!])) = \alpha_d(S[\![\mathtt{t}[\![P]\!]]\!]) = \alpha_d(S[\![P]\!])\,;$$

– *Le programme obfusqué est inintelligible dans le sens où :*

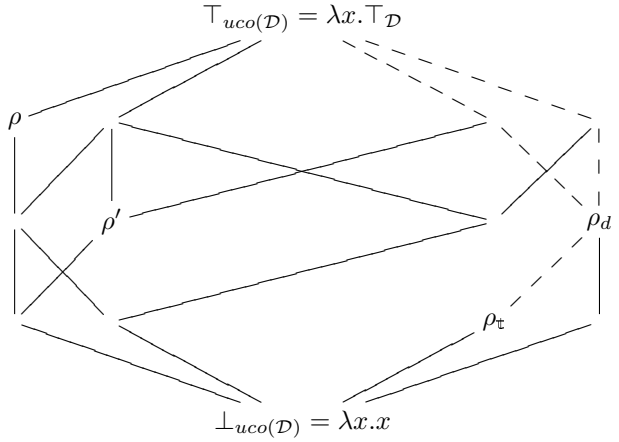
$$\forall \rho \in uco(\mathcal{D}),\ \rho_{\mathtt{t}} \sqsubseteq \rho \Rightarrow \rho_d \sqsubseteq \rho.$$

*En d'autres termes, aucune abstraction plus précise que celle de l'observation des entrées/sorties n'est préservée par la transformation* $\mathtt{t}$.

Par exemple, sur la figure suivante, on a représenté en pointillé les domaines abstraits qui sont préservés par la transformation $\mathtt{t}$. Le domaine abstrait $\rho$ n'est pas préservé par la transformation $\mathtt{t}$, tandis que le domaine $\rho'$ est préservé. On observe que l'on a pas : $\rho_d \sqsubseteq \rho'$. Donc $\mathtt{t}$ n'est pas un obfuscateur.




Sur la figure suivante, on a représenté un treillis des interprétations abstraites correspondant au cas où $\mathtt{t}$ est un obfuscateur :




La troisième propriété caractérisant un obfuscateur sémantique idéal est sans doute trop restrictive pour être utilisée en pratique, dans la mesure où l'on est pas certain de son existence. Nous pouvons cependant remarquer qu'intuitivement, une transformation de programme $\mathtt{t}$ sera d'autant plus difficile à défaire que l'ensemble des domaines abstraits qui ne sont pas préservés par $\mathtt{t}$ est important. Cela nous donne un moyen de comparer deux transformations d'obfuscation $\mathtt{t}$ et $\mathtt{t}'$ : Si $\forall \rho \in uco(\mathcal{D})$ tel que $\rho_{\mathtt{t}} \sqsubseteq \rho_d$ et $\rho_{\mathtt{t}'} \sqsubseteq \rho_d$, on a :

$$\neg(\rho_{\mathtt{t}} \sqsubseteq \rho) \Rightarrow \neg(\rho_{\mathtt{t}'} \sqsubseteq \rho),$$

alors la transformation d'obfuscation $\mathtt{t}$ est moins robuste que la transformation d'obfuscation $\mathtt{t}'$.

Nous avons vu que la propriété d'inintelligibilité d'un programme obfusqué est : $\forall \rho \in uco(\mathcal{D}),\ \rho_{\mathtt{t}} \sqsubseteq \rho \Rightarrow \rho_d \sqsubseteq \rho$. Cette propriété n'est plus vérifiée si $\neg(\forall \rho \in uco(\mathcal{D}),\ \rho_{\mathtt{t}} \sqsubseteq \rho \Rightarrow \rho_d \sqsubseteq \rho) = \neg(\forall \rho \in uco(\mathcal{D}),\ \rho_{\mathtt{t}} \sqsubseteq \rho \vee \rho_d \sqsubseteq \rho) = \exists \rho \in uco(\mathcal{D}),\ \rho_{\mathtt{t}} \sqsubseteq \rho \wedge \neg(\rho_d \sqsubseteq \rho)$.

Faire l'hypothèse que l'abstraction $\rho$ formalisant l'attaquant n'est pas conservée par la transformation $\mathbb{t}$ peut apparaître restrictive dans la définition de la robustesse de $\mathbb{t}$. Afin de s'affranchir de cette hyptothèse, on peut considérer le cas où :

$$\rho_{\mathbb{t}} \sqsubseteq \rho \wedge \neg(\rho_d \sqsubseteq \rho)$$

et considérer que dans ce cas le pouvoir d'obfuscation de $\mathbb{t}$ est d'autant meilleur que le pseudo-complément $\rho \ominus (\rho \sqcup \rho_d)$ est abstrait.

Cette définition nous paraît plus représentative de la notion d'obfuscation que la première définition proposée dans [DPG05].

### 4.2.2 Formalisation du glaive

Nous voyons qu'intuitivement, la lutte du glaive contre le bouclier formalisée en terme d'interprétation abstraite permet de caractériser une transformation de désobfuscation par la fonction d'abstraction $\alpha$ qui la paramètre. Si le degré d'abstraction de $\alpha$ est élevé, le désobfuscateur va être en mesure de traiter correctement un plus grand nombre de transformations d'obfuscation, mais son efficacité vis-à-vis d'un obfuscateur donné sera d'autant plus limitée. Inversement, si le domaine abstrait induit par $\gamma \circ \alpha$ dans le treillis des interprétations abstraites est trop concret, i.e. proche de $\lambda x.x$, le désobfuscateur sera plus sensible aux modifications induites par une transformation d'obfuscation, mais sa correction ne sera garantie que pour une classe réduite de transformations d'obfuscation.

Je propose dans cette section une formalisation de la notion de transformation de désobfuscation $\mathbb{t}$ correcte vis-à-vis d'un obfuscateur $\mathbb{o}$. Rappelons qu'une transformation de désobfuscation correcte doit garantir la préservation des sémantiques du programme. En contrepartie, certaines optimisations potentielles ne seront pas détectées. Toutes les obfuscations ne seront pas défaites. Le processus d'extraction d'information ne sera donc pas parfait, en terme de précision.
L'idée est d'exprimer, outre la préservation de la sémantique dénotationnelle du programme, l'idée que la fonction d'abstraction est insensible aux modifications induites par la transformation d'obfuscation sur les traces d'exécution du programme. La transformation $\mathbb{t}$ est correcte vis-à-vis de l'obfuscation $\mathbb{o}$ dans la mesure où elle applique des optimisations en raisonnant éventuellement sur une sur-approximation conservative des sémantiques du programme d'origine, mais sans que le résultat de l'analyse soit perturbé par la présence d'une protection. Cette idée peut être formalisée de la manière suivante dans l'ensemble partiellement ordonné $\mathcal{D}$ :

$$\forall P \in \mathbb{P},\ \alpha(S[\![\mathbb{o}[\![P]\!]]\!]) = \alpha(S[\![P]\!]).$$

Ces considérations me conduisent à proposer la définition suivante pour caractériser une transformation de désobfuscation en terme de correction :

**Définition 4.2.3** *(Désobfuscateur correct). Étant donnée une transformation d'obfuscation $\mathbb{o} : \mathbb{P} \to \mathbb{P}$, un désobfuscateur $\mathbb{t}$ correct pour $\mathbb{o}$ est défini comme une transformation vérifiant les propriétés suivantes :*

- *La transformation sémantique correspondante vérifie :*

$$\forall P \in \mathbb{P},\ t[S[\![P]\!]] = S[\![\mathbb{t}[\![\mathbb{p}(S[\![P]\!])]\!]]\!].$$

  *En d'autres termes, il s'agit d'un compilateur ou d'une transformation algorithmique ;*
- *La transformation $\mathbb{t}$ préserve la sémantique dénotationnelle des programmes :*

$$\forall P \in \mathbb{P},\ \alpha_d(t(S[\![P]\!])) = \alpha_d(S[\![\mathbb{t}[\![P]\!]]\!]) = \alpha_d(S[\![P]\!])\,;$$

- *Quelle que soit la fonction d'abstraction $\alpha$ préservée par le désobfuscateur, alors $\alpha$ est également préservée par la transformation $\mathbb{o}$ :*

$$\forall P \in \mathbb{P},\ \alpha(S[\![\mathbb{o}[\![P]\!]]\!]) = \alpha(S[\![P]\!]).$$

  *En d'autres termes, la fonction $\alpha$ fait abstraction des détails modifiés par la transformation d'obfuscation tout en préservant la sémantique originale du programme $P$.*

**Proposition 1** *Il n'existe pas de désobfuscateur correct pour l'obfuscateur idéal $\mathbb{o}$.*

**Preuve** Soit $\rho$ une abstraction conservée par $\mathtt{t}$ : $\rho_{\mathtt{t}} \sqsubseteq \rho$. Supposons que cette abstraction ne conserve pas la sémantique dénotationnelle des programmes. On a donc $\neg(\rho_d \sqsubseteq \rho)$.
Supposons l'obfuscateur $\mathfrak{o}$ idéal et $\rho_{\mathfrak{o}} \sqsubseteq \rho$, on a alors $\rho_d \sqsubseteq \rho$. Contradiction. Donc sous l'hypothèse de l'existence d'un obfuscateur idéal, on a trouvé une abstraction $\rho$ telle que $\rho_{\mathtt{t}} \sqsubseteq \rho \wedge \neg(\rho_{\mathfrak{o}} \sqsubseteq \rho)$. Donc le désobfuscateur $\mathtt{t}$ n'est pas correct pour l'obfuscateur idéal $\mathfrak{o}$. Donc il n'existe pas de désobfuscateur correct pour l'obfuscateur idéal $\mathfrak{o}$. $\square$

Avec cette définition de la désobfuscation, on a le résultat suivant, concernant la propagation de la propriété de correction d'une transformation résultant de la composition de transformations de désobfuscation :

**Proposition 2** *Si deux transformations de désobfuscation $\mathtt{t}_1$ et $\mathtt{t}_2$, caractérisées respectivement par les fonctions d'abstraction $\alpha_1$ et $\alpha_2$ sont correctes pour les transformations d'obfuscation $\mathfrak{o}_1$ et $\mathfrak{o}_2$ respectivement, alors une condition suffisante pour que la transformation de désobfuscation $\mathtt{t}_1 \circ \mathtt{t}_2$ soit correcte pour les obfuscateurs $\mathfrak{o}_1 \circ \mathfrak{o}_2$ et $\mathfrak{o}_2 \circ \mathfrak{o}_1$ est que les fonctions d'abstractions commutent : $\alpha_1 \circ \alpha_2 = \alpha_2 \circ \alpha_1$.*

**Preuve** Supposons que $\alpha_1(S[\![\mathfrak{o}_1[\![P]\!]]\!]) = \alpha_1(S[\![P]\!])$ et $\alpha_2(S[\![\mathfrak{o}_2[\![P]\!]]\!]) = \alpha_2(S[\![P]\!])$, on a :

$$\begin{array}{rcl}
\alpha_1 \circ \alpha_2(S[\![(\mathfrak{o}_1 \circ \mathfrak{o}_2)[\![P]\!]]\!]) & = & \alpha_1 \circ \alpha_2(S[\![(\mathfrak{o}_2 \circ \mathfrak{o}_1)[\![P]\!]]\!]) \\
& = & \alpha_1 \circ \alpha_2(S[\![\mathfrak{o}_1[\![P]\!]]\!]) \\
& = & \alpha_2 \circ \alpha_1(S[\![\mathfrak{o}_1[\![P]\!]]\!]) \\
& = & \alpha_2 \circ \alpha_1(S[\![P]\!]) \\
& = & \alpha_1 \circ \alpha_2(S[\![P]\!])
\end{array}$$

On démontre de même que $\alpha_1 \circ \alpha_2(S[\![(\mathfrak{o}_2 \circ \mathfrak{o}_1)[\![P]\!]]\!]) = \alpha_1 \circ \alpha_2(S[\![P]\!])$. $\square$

Ce résultat s'étend naturellement à la composition d'un nombre quelconque de transformations de désobfuscation.

Dans le cas d'une analyse complète, l'idée est d'exprimer la spécialisation du désobfuscateur au regard d'une classe de mécanismes d'obfuscation. Toutes les optimisations potentielles liées à la présence de la protection doivent être appliquées, éventuellement au détriment de la préservation de la sémantique dénotationnelle du programme. L'idée est que pour garantir que la transformation de désobfuscation est complète vis-à-vis d'une classe d'obfuscateurs, elle ne doit appliquer les transformations d'optimisation effectivement que pour défaire au moins une des transformation d'obfuscation $\mathfrak{o} \in \mathbb{O}$. Cette idée peut être formalisée de la manière suivante dans l'ensemble partiellement ordonné $\mathcal{D}$ :

$$\forall P, Q \in \mathbb{P},\ \alpha_t(S[\![P]\!]) \subseteq \alpha_t(S[\![Q]\!]) \Rightarrow \exists \mathfrak{o} \in \mathbb{O} : S[\![\mathfrak{o}[\![P]\!]]\!] \subseteq S[\![Q]\!]$$

Cette idée a également sans doute un sens dans le contexte des transformations de programme vues comme l'application d'une stratégie de réécriture de termes : la transformation doit être confluente vis-à-vis d'une classe de transformations d'obfuscation et aboutir à une unique forme normale pour une classe donnée de transformations d'obfuscation.

La fonction d'abstraction caractérisant le processus de désobfuscation doit agir comme une sorte de fonction de hachage sémantique : l'information abstraite doit caractériser un programme de manière suffisamment précise pour éviter le risque de collision avec une autre information. Cette idée peut être formalisée de la manière suivante dans l'ensemble partiellement ordonné $\mathcal{D}$ :

$$\forall A, B \in \mathcal{D},\ \alpha_t(A) \subseteq \alpha_t(B) \Rightarrow A \subseteq B \text{ (la fonction d'abstraction est dite } \textit{order-reflecting}\text{)}.$$

Ces considèrations me conduisent à proposer la définition suivante pour caractériser un désobfuscateur complet au regard d'une classe $\mathbb{O}$ de mécanismes d'obfuscation :

**Définition 4.2.4** *(Désobfuscateur complet). Étant donné un ensemble $\mathbb{O}$ de transformations d'obfuscation $\mathfrak{o} : \mathbb{P} \to \mathbb{P}$, un désobfuscateur $\mathtt{t}$ complet pour $\mathbb{O}$ est défini comme une transformation vérifiant les propriétés suivantes :*

- *La transformation sémantique correspondante vérifie :*

$$\forall P \in \mathbb{P},\ t[S[\![P]\!]] = S[\![\mathtt{t}[\![\mathtt{p}(S[\![P]\!])]\!]]\!].$$

*En d'autres termes, il s'agit d'un compilateur ou d'une transformation algorithmique ;*

– *La fonction d'abstraction $\alpha_t$ satisfait les conditions suivantes :*

$$\begin{cases} \forall P, Q \in \mathbb{P},\ \alpha_t(S[\![P]\!]) \subseteq \alpha_t(S[\![Q]\!]) \Rightarrow \exists \mathbb{o} \in \mathbb{O} : S[\![\mathbb{o}[\![P]\!]\subseteq S[\![Q]\!] \\ \forall A, B \in \mathcal{D},\ \alpha_t(A) \subseteq \alpha_t(B) \Rightarrow A \subseteq B \text{ (la fonction d'abstraction est order-reflecting).} \end{cases}$$

*En d'autres termes, la fonction $\alpha_t$ est order-embedding (une fonction d'abstraction est toujours supposée monotone. Elle est supposée ici être de plus order-reflecting) et supporte la classe de transformations d'obfuscation $\mathbb{O}$.*

Avec cette définition, on a le résultat suivant, concernant la composition de transformations de désobfuscation :

**Proposition 3** *Si deux transformations de désobfuscation $\mathbb{t}_1$ et $\mathbb{t}_2$, caractérisées respectivement par les fonctions d'abstraction $\alpha_1$ et $\alpha_2$ sont complètes pour les ensembles de transformations d'obfuscation $\mathbb{O}_1$ et $\mathbb{O}_2$ respectivement, alors la transformation de désobfuscation $\mathbb{t}_1 \circ \mathbb{t}_2$ est complète pour l'ensemble des obfuscateurs $\{\mathbb{o}_1 \circ \mathbb{o}_2,\ \mathbb{o}_1 \in \mathbb{O}_1,\ \mathbb{o}_2 \in \mathbb{O}_2\}$.*

**Preuve** Supposons les transformations de désobfuscation $\mathbb{t}_1$ et $\mathbb{t}_2$ (caractérisées respectivement par les fonctions d'abstraction $\alpha_1$ et $\alpha_2$) complètes pour les ensembles de transformations d'obfuscation $\mathbb{O}_1$ et $\mathbb{O}_2$ respectivement. Considérons deux programmes $P, Q \in \mathbb{P}$ tels que : $\alpha_1 \circ \alpha_2(S[\![P]\!]) \subseteq \alpha_1 \circ \alpha_2(S[\![Q]\!])$. La complétude de $\mathbb{t}_1$ ($\alpha_1$ est *order-reflecting*) implique que $\alpha_2(S[\![P]\!]) \subseteq \alpha_2(S[\![Q]\!])$. On en déduit, puisque $\mathbb{t}_2$ est complète, que $S[\![P]\!] \subseteq S[\![Q]\!]$ et qu'il existe $\mathbb{o}_2 \in \mathbb{O}_2 : S[\![\mathbb{o}_2[\![P]\!]\subseteq S[\![Q]\!]$. La continuité de $\alpha_1$ implique $\alpha_1(S[\![\mathbb{o}_2[\![P]\!]) \subseteq \alpha_1(S[\![Q]\!])$. La complétude de la transformation de désobfuscation $\mathbb{t}_1$ nous permet de déduire l'existence de $\mathbb{o}_1 \in \mathbb{O}_1 : S[\![(\mathbb{o}_1 \circ \mathbb{o}_2)[\![P]\!] \subseteq S[\![Q]\!]$. □

Ce résultat s'étend naturellement à la composition d'un nombre quelconque de transformations de désobfuscation. Illustrons à présent la notion de désobfuscation par le cas de l'analyse du flot de donnée par propagation de la constante [CC02] (un exemple d'application de cette méthode est détaillé dans la section D.3 de l'annexe D).
Soit $\alpha_C$ l'abstraction correspondant à l'application d'une analyse statique préliminaire de détection des constantes, définie par :

$$\alpha_C(S[\![P]\!])(L)(X) = \sqcup\{\rho(X) \mid \exists \sigma \in S[\![P]\!],\ \exists C \in \mathbb{C},\ \exists i\ :\ \sigma_i = \langle \rho, C\rangle \wedge lab[\![C]\!] = L\}.$$

Étant donné un programme $P$, une étiquette $L \in lab[\![P]\!]$ et une variable $X \in var[\![P]\!]$, on a :

$$\alpha_C(S[\![P]\!])(L)(X) = \begin{cases} \rho(X) & \text{si la variable } X \text{ est constante à l'étiquette } L \\ \top & \text{sinon.} \end{cases}$$

La transformation de désobfuscation syntaxique s'appuyant sur cette fonction d'abstraction est donnée par :

$$\mathbb{t}_{CP}[\![P]\!] = (\mathbb{p} \circ t_{CP} \circ S)[\![P]\!],$$

où la transformation sémantique correspondante est définie par :

$$t_{CP}(S[\![P]\!], \alpha_C(S[\![P]\!])) = \{t_{CP}(\sigma, \alpha_C(S[\![P]\!])) \mid \sigma \in S[\![P]\!]\}$$

Pour chaque état $\sigma = \langle \rho, C\rangle$, la transformation est définie par :

$$t_{CP}(\langle \rho, C\rangle, \alpha_C(S[\![P]\!])(lab[\![C]\!])) = \langle \rho, t_{CP}(C, \alpha_C(S[\![P]\!])(lab[\![C]\!]))\rangle$$

où la transformation appliquée à une instruction $C = L : A \rightarrow L'$ est définie par :

$$t_{CP}(L : A \rightarrow L', \alpha_C(S[\![P]\!])(L)) = L : t_{CP}(A, \alpha_C(S[\![P]\!])(L)) \rightarrow L'$$

et la transformation appliquée à une action $A$ est définie par la spécialisation :

$$t_{CP}(A, \alpha_C(S[\![P]\!])(L)) = R[\![A]\!]\rho$$

Rappelons que la sémantique de spécialisation d'une action $X := E$ est donnée par :

$$R[\![X := E]\!]\rho = \begin{cases} \langle \rho \backslash X, X := R[\![E]\!]\rho\rangle & \text{si } E \text{ est dynamique} \\ \langle \rho(X = R[\![E]\!]\rho), skip\rangle & \text{si } E \text{ est statique} \end{cases}$$

En d'autres termes, après analyse préliminaire par détection des constantes, la propagation des constantes par la transformation d'optimisation $\mathbb{t}_{CP}$ consiste à remplacer les assignation de ces variables lorsqu'elles sont statiques par $skip = true$.

Considérons l'ensemble $\mathbb{O}$ des transformations d'obfuscation qui modifient les instructions du programme sans en modifier le contexte d'exécution. La transformation d'optimisation $\mathbb{t}_{CP}$ est par définition $\alpha_C$-conservative :

$$\forall P \in \mathbb{P},\ \alpha_C(S[\![\mathbb{t}_{CP}[\![P]\!]]\!]) = \alpha_C(S[\![P]\!]).$$

Par ailleurs, on vérifie facilement que les éléments de $\mathbb{O}$ sont $\alpha_C$-conservatifs puisque $\mathbb{t}_{CP}$ laisse l'environnement inchangé :

$$\forall \mathbb{o} \in \mathbb{O},\ \forall P \in \mathbb{P},\ \alpha_C(S[\![\mathbb{o}[\![P]\!]]\!]) = \alpha_C(S[\![P]\!]).$$

La transformation $\mathbb{t}_{CP}$ est donc une transformation de désobfuscation correcte pour la classe d'obfuscateurs $\mathbb{O}$.

En revanche, nous avons prévu dans la syntaxe générale de nos programmes plusieurs types d'actions $A \in \mathbb{A}$ : l'affectation $X := E$, l'affectation aléatoire $X :=?$, le test $B$ avec la valeur particulière $true \in \mathbb{B}$ qui correspond à l'action $skip$. Si une transformation d'obfuscation $\mathbb{o}$ modifie le type des actions, alors elle ne sera pas $\alpha_C$-conservative. La transformation $\mathbb{t}_{CP}$ est alors une transformation de désobfuscation incorrecte vis-à-vis de l'obfuscateur $\mathbb{o}$.

## 4.3 Extraction d'un schéma de détection

Des travaux récents [DPCJD07] ont permi de modéliser un moteur de détection dit sémantique [CJS+05] dans le cadre de l'interprétation abstraite et de prouver dans ce cadre la correction et la complétude de ce détecteur, sous certaines hypothèses (ils considèrent en particulier que certaines réponses à des questions difficiles sont fournis par des oracles, comme le problème de l'extraction du graphe CFG d'un programme ou la détection de junk code), vis-à-vis d'une classe de transformations d'obfuscation. Ils ont complété cette démonstration par une preuve des limites de leur schéma de détection au regard de certaines transformations d'obfuscation.

Nous étudions dans la suite de cette section le modèle qu'ils ont utilisé pour raisonner et évaluer la résistance d'un moteur de détection virale sémantique au regard d'une classe de transformations d'obfuscation. Je propose une généralisation de ce modèle permettant de prendre en compte l'étape de normalisation ou de réécriture caractérisant l'extraction d'un schéma de détection virale. J'étudie en particulier la propagation des critères de correction et de complétude du processus d'extraction d'information, vu comme une transformation de désobfuscation, à la caractérisation d'un moteur de détection. Je m'assure également que ces résultats sont en cohérence avec les résultats obtenus dans [DPCJD07].

Nous avons défini au paragraphe précédent un désobfuscateur $\mathbb{t}$ correct pour un obfuscateur $\mathbb{o}$ comme une transformation telle que l'ensemble des fonctions d'abstraction préservées par $\mathbb{t}$ soit inclu dans l'ensemble des fonctions d'abstraction préservée par l'obfuscateur (définition 4.2.3).
Nous allons voir que cette définition induit celle de complétude d'un schéma de détection, lorsque le processus d'extraction est fondé sur l'utilisation d'une transformation de désobfuscation lors de l'étape de normalisation ou de réécriture implémentée par le moteur de détection.
Considérons le schéma de détection $(\alpha_t(S[\![M]\!]), \alpha)$ défini comme la donnée :

- d'une information utile $\alpha_t(S[\![M]\!])$ extraite de la sémantique $S[\![M]\!]$ du virus $M$ ;
- d'une fonction d'abstraction $\alpha_t$ qui extrait l'information sémantique utile des programmes.

Le processus de détection consiste à extraire l'information sémantique du programme $P$, par utilisation de la fonction d'abstraction $\alpha_t$ déjà utilisée pour extraire l'information utile du virus $M$, puis à vérifier qu'il existe une fonction d'abstraction $\alpha_r$ telle que $\alpha_t(S[\![M]\!]) \subseteq \alpha_t(\alpha_r(S[\![P]\!]))$.
La fonction d'abstraction $\alpha_r$ peut être définie comme une restriction du contexte d'exécution aux instructions étiquetées par un ensemble $lab_r[\![P]\!]$ [DPCJD07].
La fonction d'abstraction $\alpha_t$ est supposée préservée par la transformation de désobfuscation $\mathbb{t}$. Si la fonction de désobfuscation est correcte, nous allons voir que le schéma de détection est complet au sens suivant :

**Définition 4.3.1** *(Schéma de détection complet, définition adaptée de [DPCJD07]).*
*Un schéma de détection* $(\alpha_t(S[\![M]\!]), \alpha_t)$ *est complet vis-à-vis d'une transformation d'obfuscation* $\mathfrak{o}$ *si et seulement si :*

$$P \text{ infecté par } \mathfrak{o}(M) \Rightarrow \exists \alpha_r : \alpha_t(S[\![M]\!]) \subseteq \alpha_t \circ \alpha_r(S[\![P]\!]).$$

Remarquons que cette définition est une simple généralisation de la définition donnée dans [DPCJD07] à un détecteur sémantique de virus complet pour une fonction d'abstraction. Nous n'adoptons pas cette définition car elle correspond au schéma de détection d'un moteur de détection particulier, appelé par les auteurs détecteur sémantique, fondé sur une fonction d'abstraction qui ne considère que les effets de l'exécution d'un programme sur son environnement, en faisant abstraction des instructions.

**Proposition 4** *Une transformation de désobfuscation correcte pour un obfuscateur* $\mathfrak{o}$ *induit un schéma de détection complet pour* $\mathfrak{o}$*.*

**Preuve** Supposons la transformation de désobfuscation $\mathfrak{t}$ correcte pour $\mathfrak{o}$. Si $P$ est infecté par $\mathfrak{o}(M)$, alors il existe une restriction $\alpha_r$ telle que $\alpha_r(S[\![P]\!]) = S[\![\mathfrak{o}[\![M]\!]]\!]$. Nous supposons $\mathfrak{t}$ $\alpha_t$-conservative ($\alpha_t(S[\![\mathfrak{t}[\![P]\!]]\!]) = \alpha_t(S[\![P]\!])$) donc puisque $\mathfrak{t}$ est correcte pour $\mathfrak{o}$, $\mathfrak{o}$ est également $\alpha_t$-conservative. Donc $\alpha_t(S[\![\mathfrak{o}[\![M]\!]]\!]) = \alpha_t(S[\![M]\!])$. D'où le résultat : $P$ infecté par $\mathfrak{o}(M) \Rightarrow \exists \alpha_r : \alpha_t(S[\![M]\!]) \subseteq \alpha_t \circ \alpha_r(S[\![P]\!])$. $\square$

Il est donc possible d'inférer la complétude d'un schéma de détection $(\alpha_t(S[\![M]\!]), \alpha_t)$ au regard d'une transformation d'obfuscation $\mathfrak{o}$ (ou d'un ensemble de transformations d'obfuscation) à partir de la correction de la transformation de désobfuscation $\mathfrak{t}$ caractérisant son extraction. Nous pouvons définir la notion de correction d'un schéma de détection de la manière suivante :

**Définition 4.3.2** *(Schéma de détection correct, définition adaptée de [DPCJD07]).*
*Un schéma de détection* $(\alpha_t(S[\![M]\!]), \alpha_t)$ *est correct vis-à-vis d'un ensemble de transformations d'obfuscation* $\mathbb{O}$ *si et seulement si :*

$$\exists \alpha_r : \alpha_t(S[\![M]\!]) \subseteq \alpha_t \circ \alpha_r(S[\![P]\!]) \Rightarrow \exists \mathfrak{o} \in \mathbb{O} : P \text{ infecté par } \mathfrak{o}(M).$$

Remarquons que la propriété d'$\alpha_t$-conservation des obfuscateurs $\mathfrak{o} \in \mathbb{O}$ n'est pas suffisante pour garantir la correction d'un schéma de détection : à titre d'exemple, considérer la fonction d'abstraction $\alpha_\top = \lambda.x\top$. Toutes les transformations de programme sont $\alpha_\top$-conservatives. Pourtant, un moteur de détection utilisant un processus d'extraction d'information fondé sur cette fonction d'abstraction est assuré d'un taux de fausse alerte maximal puisqu'il va classifier tous les programmes comme infectés.

La définition 4.2.4 caractérise un désobfuscateur complet par des conditions portant sur la fonction d'abstraction $\alpha_t$. Celle-ci doit satisfaire les conditions suivantes :

$$\begin{cases} \forall P, Q \in \mathbb{P},\ \alpha_t(S[\![P]\!]) \subseteq \alpha_t(S[\![Q]\!]) \Rightarrow \exists \mathfrak{o} \in \mathbb{O} : S[\![\mathfrak{o}[\![P]\!]\!] \subseteq S[\![Q]\!] \\ \forall A, B \in \mathcal{D},\ \alpha_t(A) \subseteq \alpha_t(B) \Rightarrow A \subseteq B \end{cases}$$

Nous allons voir que cette propriété induit celle de correction d'un schéma de détection, lorsque le processus d'extraction est fondé sur l'utilisation d'une transformation de désobfuscation lors de l'étape de normalisation ou de réécriture implémentée par le moteur de détection.

**Proposition 5** *Une transformation de désobfuscation complète pour un ensemble d'obfuscateurs* $\mathbb{O}$ *induit un schéma de détection correct pour* $\mathbb{O}$*.*

**Preuve** Supposons la transformation de désobfuscation $\mathfrak{t}$ complète pour $\mathbb{O}$. Si $\exists \alpha_r : \alpha_t(S[\![M]\!]) \subseteq \alpha_t \circ \alpha_r(S[\![P]\!])$, alors $\exists \mathfrak{o} \in \mathbb{O} : S[\![\mathfrak{o}[\![M]\!]]\!] \subseteq \alpha_r(S[\![P]\!])$, d'où le résultat : $\exists \alpha_r : \alpha_t(S[\![M]\!]) \subseteq \alpha_t \circ \alpha_r(S[\![P]\!]) \Rightarrow \exists \mathfrak{o} \in \mathbb{O} : P$ infecté par $\mathfrak{o}(M)$. $\square$

On en déduit les résultats suivants, concernants la complétude et la correction de la composition de schémas de détection :

**Proposition 6** *Étant donnés deux schémas de détection* $(\alpha_1(S[\![M]\!]), \alpha_1)$ *et* $(\alpha_2(S[\![M]\!]), \alpha_2)$*,*

- *si ces schémas de détection sont issus de processus d'extraction ($\mathbb{t}_1$ et $\mathbb{t}_2$) corrects vis-à-vis des classes d'obfuscateurs $\mathbb{O}_1$ et $\mathbb{O}_2$ respectivement alors le schéma de détection $(\alpha_1 \circ \alpha_2(S[\![M]\!]), \alpha_1 \circ \alpha_2)$ est complet pour l'ensemble des obfuscateurs $\{\mathbb{o}_1 \circ \mathbb{o}_2,\ \mathbb{o}_1 \in \mathbb{O}_1,\ \mathbb{o}_2 \in \mathbb{O}_2\} \cup \{\mathbb{o}_2 \circ \mathbb{o}_1,\ \mathbb{o}_1 \in \mathbb{O}_1,\ \mathbb{o}_2 \in \mathbb{O}_2\}$.*
- *si ces schémas de détection sont issus de processus d'extraction ($\mathbb{t}_1$ et $\mathbb{t}_2$) complets vis-à-vis des classes d'obfuscateurs $\mathbb{O}_1$ et $\mathbb{O}_2$ respectivement alors le schéma de détection $(\alpha_1 \circ \alpha_2(S[\![M]\!]), \alpha_1 \circ \alpha_2)$ est correct pour l'ensemble des obfuscateurs $\{\mathbb{o}_1 \circ \mathbb{o}_2,\ \mathbb{o}_1 \in \mathbb{O}_1,\ \mathbb{o}_2 \in \mathbb{O}_2\}$.*

**Preuve** Le premier résultat est un corrolaire immédiat des propositions 2 et 4. Le second résultat est un corrolaire immédiat des propositions 3 et 5. □

Remarquons que les conditions à imposer au processus d'extraction d'information (conditions portant sur la fonction d'abstraction $\alpha_t$) et permettant d'induire un schéma de détection correct sont difficiles à exploiter en pratique. Pour évaluer la correction d'un moteur de détection dans le cadre formel de l'interprétation abstraite, il est donc nécessaire en pratique de faire des hypothèses très fortes sur la correction du processus d'extraction, par le biais de l'utilisation d'oracles par exemple [DPCJD07].

## 4.4 Vers une modélisation de l'analyse dynamique par interprétation abstraite

Nous avons vu dans la section 4.2 l'intérêt d'une approche fondée sur l'analyse des sémantiques d'un programme pour qualifier en terme de correction et de complétude une transformation de désobfuscation. Le cadre de l'interprétation abstraite se prête naturellement à cette formalisation. En outre, ces critères s'étendent naturellement à la qualification du processus d'extraction d'information d'un moteur de détection virale et nous permettent de statuer sur la correction et la complétude d'un schéma de détection au regard d'une classe identifiée de mécanismes (cf. section 4.3).
Cependant, cette formalisation s'applique uniquement à l'analyse statique des programmes.

Nous avons observé au chapitre 3 que les méthodes les plus performantes, en terme de rétro-ingénierie logicielle, se fondent sur une analyse dynamique ou hybride statique/dynamique.
En effet la résolution de certains problèmes de l'analyse statique comme l'analyse des pointeurs ou des alias restent au dessus de l'état de l'art. Même des algorithmes de complexité exponentielle ne fournissent pas de résultat suffisamment précis pour désobfusquer un programme protégé par embarquement de certains problèmes complexes de l'analyse statique et de l'optimisation. Ces problèmes ne sont pas toujours aussi difficiles lorsqu'ils sont appréhendés par l'analyse dynamique. À titre d'exemple, déterminer en cours d'exécution si deux pointeurs vérifient une relation d'alias ne nécessite que quelques cycles CPU pour comparer les deux pointeurs.

L'analyse hybride statique/dynamique permet de tirer partie des atouts de ces deux approches, duales par nature. Malheureusement, peu de travaux existent permettant de lier ces deux domaines de recherche [Ern03].

En pratique, l'analyse statique est lente lorsqu'elle est doit être précise ou que le programme est de grande taille. À l'inverse, l'analyse dynamique peut être presque aussi rapide que l'exécution du programme. Cependant, l'analyse dynamique est lente lorsqu'elle est doit être exhaustive. Compte tenu des contraintes en terme de vitesse d'analyse d'un produit antivirus, l'analyse dynamique est une alternative séduisante à l'analyse statique.

Nous avons vu que l'analyse statique peut généralement être représentée par un domaine abstrait. Cette abstraction correspond à une approximation conservative, c'est-à-dire qu'elle utilise un modèle abstrait des états du programme qui perd de l'information, mais est une représentation compacte et facile à manipuler du programme. Des propriétés faibles du programmes sont assurées d'être vérifiées.
L'analyse dynamique est précise car aucune approximation ou abstraction n'est nécessaire : l'analyse se fonde sur le comportement exact du programme lors de son exécution. Ces deux approches sont donc fondamentalement duales par nature, au regard des critères de correction et de complétude. Le tableau suivant résume cette dualité :

| méthode d'analyse | correction | complétude |
|---|---|---|
| analyse statique | **Oui**. L'analyse statique est généralement correcte. L'approche conservative (pessimiste) induite par l'abstraction assure que les résultats de l'analyse restent valables, indépendamment des données fournies en entrée au programme et indépendamment de l'environnement d'exécution du programme. | **Non**. Les transformations de code qui ont besoin d'entrées précises ne peuvent pas exploiter le résultat d'une analyse statique. |
| dynamique | **Non**. L'analyse dynamique est généralement incorrecte car les résultats ne se généralisent pas à d'autres exécutions, correspondant à des données différentes fournies en entrées au programme ou à un environnement d'exécution différent. Nous n'avons pas de garantie que les comportements observés pour l'ensemble des données d'entrées caractérisant une suite de test est caractéristique de toutes les exécutions possibles du programme. | **Oui**. L'analyse dynamique est complète, dans la mesure où elle se fonde sur une exécution concrète du programme. |

Le principal problème de l'analyse dynamique est qu'elle est approximative. Nous ne pouvons pas construire de fonction d'abstraction $\alpha$ permettant de raisonner sur la correction ou la complétude d'une transformation de programme lorsque cette dernière est approximative[2].

Une première formalisation de l'analyse dynamique en terme d'abstraction des sémantiques concrètes a pourtant été proposée par S. Lucas [Luc00]. Considérant l'analyse dynamique d'un programme $P$ comme une abstraction $\alpha(S[\![P]\!])$ des sémantiques concrètes $S[\![P]\!]$ de ce programme, où seules les traces correspondant à une exécution observée sont collectées, il introduit la notion de propriété dynamiquement observable.

Tout problème d'analyse de programme peut être formulé comme un problème de vérification d'une propriété. Etant donnée une fonction $S : \mathbb{P} \to \mathcal{D}$, où $\mathbb{P}$ désigne un langage de programmation et $\mathcal{D}$ un domaine abstrait de valeurs, la question est de savoir si une propriété $\pi : \mathcal{D} \to \{0, 1\}$ est satisfaite par un programme $P \in \mathbb{P}$, c'est-à-dire si $\pi(S[\![P]\!]) = 1$.
Dans le cas de l'analyse statique, l'objectif est de donner une réponse à ce problème sans exécuter le programme. Dans le cas de l'analyse dynamique, l'objectif est de donner une réponse à ce problème, sur la base d'une information partielle relative à l'exécution du programme. Nous ne disposons que de la description $\lhd P \rhd$, approximative, résultant de l'observation dynamique d'une partie de l'exécution de $P$.

Notons $\mathcal{R}(\lhd P \rhd)$ l'ensemble des descriptions plus précises qu'il est possible d'obtenir de l'exécution du programme $P$, nous exigeons d'une propriété $\pi$ qu'elle vérifie la condition :

$$\pi(\lhd P \rhd) = 1 \Rightarrow \forall X \in \mathcal{R}(\lhd P \rhd),\ \pi(X).$$

En d'autres termes, si la propriété $\pi$ est satisfaite pour une observation dynamique du programme $P$, elle l'est également pour toutes les observations dynamiques qu'il est possible d'obtenir de l'exécution du programme $P$.

[2] Une transformation $\mathbb{t}$ est dite conservative vis-à-vis d'une fonction d'abstraction $\alpha$ si chaque trace de $S[\![P]\!]$ est une sous-trace de $S[\![\mathbb{t}[\![P]\!]]\!]$ à l'abstraction $\alpha$ près :

$$\forall \sigma \in S[\![P]\!],\ \exists \sigma' \in S[\![\mathbb{t}[\![P]\!]]\!] \ :\ \alpha(\sigma) \preceq \alpha(\sigma').$$

En d'autres termes, une transformation caractérisée par une abstraction $\alpha$ est conservative si, avec un point de vue filtré par cette abstraction, on retrouve toutes les traces du programme d'origine dans le programme transformé.

Dans le cas contraire, une transformation $\mathbb{t}$ est dite *approximative vis-à-vis de la fonction d'abstraction* $\alpha$, i.e. qu'il existe des traces du programme d'origine qu'on ne retrouve pas - comme sous-trace - dans le programme transformé :

$$\exists \sigma \in S[\![P]\!],\ \forall \sigma' \in S[\![\mathbb{t}[\![P]\!]]\!] \ :\ \alpha(\sigma) \not\preceq \alpha(\sigma').$$

De telles propriétés sont dites dynamiquement observables [Luc00]. Nous pouvons identifier ces propriétés avec des prédicats monotones vis à vis d'un ordre $\sqsubset_{Dyn}$ sur $\mathcal{D}$ permettant d'exprimer les raffinements de $\lhd P \rhd$.
La correction d'une telle approche est assurée en montrant que tout prédicat $\pi$ monotone vis-à-vis de $\sqsubset_{Dyn}$ est dynamiquement observable.
Dans ce cadre, le problème de l'analyse dynamique revient à montrer que :

$$\lhd P \rhd \sqsubset_{Dyn} S[\![P]\!] \text{ et } \pi(\lhd P \rhd) = 1$$

La monotonie de $\pi$ nous assure que $\pi(\lhd P \rhd) = 1$. Si en outre $\lhd P \rhd \sqsubset_{Dyn} S[\![P]\!]$, alors $\lhd P \rhd$ est une observation dynamique complète de $P$.
Nous voyons donc que le choix de l'ordre $\sqsubset_{Dyn}$ est crucial, pour permettre une grande variété de propriétés dynamiquement observables.

Je pense que l'étude des propriétés dynamiquement observables est un axe de recherche prometteur pour permettre l'extension des critères de correction et de complétude d'une transformation de désobfuscation statique, fondée sur l'étude de propriétés statiquement observables, à des critères permettant de qualifier une transformation de désobfuscation dynamique ou hybride statique/dynamique dans le cadre formel de l'interprétation abstraire.

## 4.5 Conclusion

Nous avons vu dans ce chapitre l'intérêt d'une approche fondée sur l'analyse des sémantiques d'un programme pour qualifier en terme de correction et de complétude une transformation de désobfuscation (section 4.2). Le cadre de l'interprétation abstraite se prête naturellement à cette formalisation. En outre, ces critères s'étendent naturellement à la qualification du processus d'extraction d'information d'un moteur de détection virale et nous permettent de statuer sur la correction et la complétude d'un schéma de détection au regard d'une classe identifiée de mécanismes (section 4.3).

L'une des limitation majeure de ce formalisme est qu'il s'applique uniquement à l'analyse statique des programmes. Nous avons observé au chapitre 3 que les méthodes les plus performantes, en terme de rétro-ingénierie logicielle, se fondent sur une analyse dynamique ou hybride statique/dynamique. L'analyse hybride statique/dynamique permet de tirer partie des atouts de ces deux approches, duales par nature. Malheureusement, peu de travaux existent permettant de lier ces deux domaines de recherche. Dans le contexte de la détection virale, il nous serait pourtant très utile de disposer de résultats équivalents nous permettant de qualifier le processus dynamique d'extraction d'information d'un moteur de détection fondé par exemple sur l'émulation.

Nous avons présenté à la section 4.4 la synergie et la dualité existant entre l'analyse statique et l'analyse dynamique au regard des critères de correction et de complétude, ainsi que la notion de propriété dynamiquement observable, dans le cadre de l'analyse des sémantiques d'un programme.
J'identifie comme axe de recherche future l'étude de l'extension des critères de correction et de complétude d'une transformation de désobfuscation statique, fondée sur l'étude de propriétés statiquement observables, à des critères permettant de qualifier une transformation de désobfuscation dynamique ou hybride statique/dynamique dans le cadre formel de l'interprétation abstraite. Je pense que l'étude des propriétés dynamiquement observables est un axe de recherche prometteur. Dans le contexte de l'évaluation des produit anti-virus, cela nous permettrait de formaliser la rétro-ingénierie et l'analyse dynamique comme approximation discrète des sémantiques concrètes d'un programme. Un tel cadre formel nous serait en outre précieux pour qualifier l'information sémantique extraite lors du processus d'analyse dynamique d'un programme viral.

Chapitre 5

# Unpacking

Nous avons introduit aux chapitres précédents les principaux aspects techniques et théoriques liés à la protection logicielle.
Nous avons vu au chapitre 3 que les solutions de protection logicielle du marché permettent d'intégrer des mécanismes de protection à chacune des étapes de la compilation. Les packers d'exécutable, dont les principaux mécanismes ont été présentés (section 3.2.2), constituent la dernière ligne de protection contre la rétro-ingénierie.
Concernant la formalisation de la rétro-ingénierie dans un cadre théorique adapté, nous avons constaté au chapitre 4 que les méthodes dynamiques d'extraction d'information d'un programme protégé sont aujourd'hui encore mal formalisées.
Nous étudions dans ce chapitre une méthode dynamique d'extraction d'information et discutons de son efficacité d'un point de vue technique.

## 5.1 Introduction

Selon une analyse de l'équipe AV-Test, [BM06, tea06], plus de 92% des infections informatiques de la WildList 03/2006 sont packées. Environ 30 packers différents ont été identifiés lors de leurs tests. Ils ont observé que plusieurs packers différents sont souvent utilisés au sein d'une même famille pour échapper à la détection. Par conséquent, les produits antivirus ont besoin de supporter beaucoup de packers et doivent se préparer à en affronter de nouveaux tous les jours. Selon leur banc de tests, presque tous les produits antivirus utilisent un moteur d'unpacking, mais les taux de détection ne sont pas assez élevés. De nombreux packers ne sont pas détectés du tout par certains produits antivirus [FEG+07, FGJ+08]. Pour certain antivirus, le taux de faux positifs est trop élevé : ils signalent à tort les fichiers packés comme correspondant à certains virus. Les moteurs d'unpacking antiviraux ne sont pas assez génériques et comportent de nombreuses failles. Lorsque l'on évalue la sécurité d'un produit antivirus, ce problème doit être vérifié.

Une rapide analyse avec PEiD [PEi07] d'un ensemble d'infections informatiques téléchargées depuis la collection de virus Vx Heavens [Hea08], 01/2006 confirme les résultats obtenus par l'équipe AV-Test. Le tableau 5.1 présente les dix packers les plus utilisés par les développeurs de code viraux, avec leurs fréquences d'utilisation dans la population virale étudiée. Observons que ces packers sont assez anciens et ne sont pas nécessairement représentatifs de l'état de l'art dans le domaine de la protection d'exécutable, dans la mesure où ils ne sont pas tous maintenus.

J'appelle unpacker un programme qui prend en entrée un exécutable packé, supprime le *loader* de protection, et fournit en sortie l'exécutable embarqué. La routine de packing est spécialement conçue pour être résistante à la rétro-ingénierie et donc pour rendre difficile la construction d'une routine d'unpacking.

Disposer d'un tel outil est un pré-requis à toute analyse en boîte noire d'un virus ou d'un programme antivirus, dans la mesure où de nombreux programmes viraux sont packés. Un tel outil est en fait requis lorsque l'on vérifie en boîte noire les exigences de sécurité d'un produit antivirus. Durant le test d'un produit antivirus, un

| Packer | Fréquence d'utilisation |
|---|---|
| UPX | 59 % |
| ASPack | 16% |
| PECompact | 5% |
| FSG | 4% |
| tElock | 3% |
| LCC Win32 | 2% |
| PE Pack | 1% |
| PEX | 1% |
| Neolite | 1% |
| PKLITE32 | 0.5% |
| y0da's Cryptor | 0.5% |

Tab. 5.1 – Packers les plus utilisés par les virus de la collection Vx Heavens, mars 2006

analyste de sécurité a besoin de fabriquer une population virale nécessaire pour mener à bien plusieurs tests d'efficacité. L'*unpacking* est une première étape indispensable avant de pouvoir appliquer des transformations ou tout algorithme d'extraction d'information sur un ensemble viral. Un unpacker sécurisé est également utile lorsque l'on souhaite vérifier la résistance vis-à-vis de la rétro-ingénierie de tout produit de sécurité packé ou protégé.

Plusieurs outils d'analyse statique et dynamique implémentent d'ores et déjà des algorithmes d'unpacking, mais requièrent le plus souvent une intervention humaine et ne sont pas bien adaptés à l'unpacking automatique de programmes aussi dangereux que les virus.

Je propose dans ce chapitre un nouvel algorithme d'unpacking et un outil, appelé VxStripper, fondé sur un CPU logiciel et dédié à l'analyse de code viral. Cet outil propose un système fiable et sûr pour récupérer automatiquement de l'information sur un virus et ses interactions avec l'exécutif du système d'exploitation Windows XP (hook furtif des API natives et Win32). Son module d'unpacking permet de récupérer le corps d'un virus protégé et de reconstruire l'exécutable.

Une partie du contenu de ce chapitre a été publiée dans les actes de la conférence AVAR 2006 [Jos06b].

Le reste de ce chapitre est organisé comme suit. La section 5.2 répond à la question : qu'est-ce qu'un unpacker sécurisé et pourquoi est-ce utile ? Nous présentons les autres travaux dans ce domaine et les différentes implémentations existantes d'un tel outil. Nous discutons également des contraintes liées à l'unpacking d'infections informatiques. La contribution de nos travaux dans ce domaine, au regard des autres solutions existantes, est également démontrée dans cette section. Les limites et conditions d'utilisation sont données.
La section 5.3 de ce chapitre présente les spécifications de l'outil VxStripper, qui intègre le module d'unpacking et fournit une information fiable sur le programme cible.

## 5.2 Unpacking sécurisé et avancé

### 5.2.1 Qu'est-ce que c'est ?

Cet outil est conçu pour un analyste de sécurité. Il couvre au moins les domaines d'utilisation suivants :

- *Analyse virale* : les virus *In-the-wild* sont souvent packés. Ils implémentent également des mécanismes de protection logicielle, qui interdisent l'utilisation d'outils de développement comme les désassembleurs interactifs et les débogueurs.
- *Vérification de la sécurité de logiciels packés ou protégés* : d'autres applications implémentent des mécanismes de protection de manière à assurer le secret de données critiques ou d'algorithmes. Les logiciels de gestion numérique des droits et autres produits de sécurité (Cloakware e.g. [Clo07]) ont besoin de telles fonctions de sécurité.

Avant de pouvoir contrôler la robustesse de leurs fonctions de sécurité et recouvrer l'information sur leurs fonctionnements internes, l'unpacking est souvent la première étape de l'analyse.

– *Test des produits anti-virus* : lorsqu'il vérifie la robustesse des fonctions de sécurité d'un produit anti-virus, un analyste de sécurité a souvent besoin de forger une population virale se conformant à plusieurs critères avant de pouvoir être utilisée dans des tests. Parmi ces propriétés, la première est que le virus doit être déchiffré, lorsque c'est possible [Fil05c].
  En outre, les transformations d'obfuscation et les algorithmes d'extraction de signature requièrent l'unpacking comme première étape.
  L'utilisation qui peut être faite d'un outil d'unpacking sécurisé dans une plate-forme de tests est décrit dans nos travaux précédents [Jos06a].
– *Système d'injection de fautes* : le test boîte noire comme décrit ci-dessus requière la possibilité de travailler avec un ensemble de données adéquates en entrée du système cible.
  L'analyse en boîte grise procède de même, mais applique également dynamiquement des modifications au programme qui s'exécute lui-même, pas seulement à ses données d'entrées.
  Avec la possibilité d'appliquer des transformations par modification d'instructions ou de blocs de bases, la fonctionnalité d'unpacking de VxStripper est un composant critique de tout système d'injection de fautes[1].

### 5.2.2 Autres travaux

Plusieurs outils d'analyse statique et dynamique implémentent déjà des algorithmes d'unpacking, mais ils requièrent souvent une intervention humaine et ne sont pas adaptés à l'unpacking automatique d'un programme aussi dangereux qu'un virus.

IDA 4.9 propose un plug-in appelé `uunp` (*universal PE unpacker debugger plug-in module* [Dat05, Dat06]) qui automatise l'analyse et l'unpacking de binaires packés. Ce plug-in utilise le débogueur pour laisser le programme se déchiffrer lui-même en mémoire et dès que l'exécution atteint le point d'entrée original, il suspend l'exécution du programme. L'utilisateur peut alors prendre un instantané de la mémoire.

L'algorithme de ce plug-in est le suivant :

1. démarrer le processus et observer l'exécution jusqu'à ce que le point d'entrée du programme packé soit atteint ;
2. ajouter un point d'arrêt à l'adresse `kernel32.GetProcAddress`, reprendre l'exécution et attendre jusqu'à ce que le packer appelle la fonction `GetProcAddress`. Si le nom de la fonction passée à `GetProcAddress` n'est pas dans la liste des fonctions à ignorer, alors passer en mode trace.
   Un appel à la fonction `GetProcAddress()` signifie vraisemblablement que le programme a été unpacké en mémoire et est à présent en train de renseigner sa table des imports (IAT) ;
3. tracer le programme en mode pas-à-pas jusqu'à ce qu'une instruction de branchement mène à la zone du point d'entrée original ;
4. dès que le pointeur de l'instruction courante appartient à la zone de l'OEP[2], suspendre l'exécution et informer l'utilisateur.

Donc, en résumé, nous permettons au code de la protection de faire son travail à pleine vitesse jusqu'à ce qu'il commence à renseigner la table des imports. À ce moment, nous passons en mode pas-à-pas et essayons de trouver le point d'entrée original. Bien que cet algorithme fonctionne avec UPX, ASPack et un certain nombre d'autres packers, il peut échouer et perdre le contrôle de l'exécution du programme packé. Par conséquent ce plug-in doit être utilisé avec précaution.

[1] Une telle approche est à de nombreux égards comparable aux techniques de *fuzzing*, utilisées pour valider la sûreté de fonctionnement des logiciels, mais également leurs sécurité. Le principe du *fuzzing* est d'injecter des données aléatoires dans les entrées d'un programme. Si le programme échoue, alors il y a des défauts à corriger. La différence de notre approche avec le *fuzzing* est que les fautes ne sont pas injectées uniquement au niveau des données d'entrées du programme, mais également au niveau des données manipulées et transformées par le programme lors de son exécution et au niveau du code du programme. L'objectif n'est plus de détecter une erreur de programmation mais de récupérer de l'information sur le fonctionnement du programme (en particulier concernant les protections qu'il implémente ou les secrets qu'il manipule) en vue de faciliter sa rétro-conception.

[2] Original Entry Point

Les plug-ins d'Ollydbg [Yus07, Ope07] (*FindCrypt, DeJunk, Ollybone, OllyDump, OllySnake, Polymorphic Breakpoint Manager, PE Dumper, Universal Hooker, Virtual2Physical* et *OllyScript*) sont très utiles pour unpacker manuellement un exécutable PE protégé.
Le plug-in *OllyScript* permet à un analyste de sécurité d'automatiser Ollydbg en écrivant des scripts dans un langage proche de l'assembleur. Ce plug-in est particulièrement pratique car lorsque l'on analyse manuellement un binaire protégé, de nombreuses tâches très répétitives doivent être effectuées simplement pour trouver l'OEP au sein de l'application protégée. En utilisant ce plug-in, c'est possible en un seul script. À titre d'exemple, le script suivant atteint automatiquement l'OEP d'un exécutable packé avec UPX :

```
        eob Break
        findop eip, #61#
        bphws $RESULT, "x"
        run
    Break:
        sto
        sto
        bphwc $RESULT
        ret
```

De nombreux scripts OSC[3] ont déjà été écrits et permettent de trouver l'OEP d'un exécutable packé, de réparer l'IAT (*Import Address Table*), de supprimer le junk code et de trouver la table des relocations et le code déplacé (*stolen code*).

Tous ces outils d'analyse dynamique doivent être utilisés avec précaution, tout particulièrement lorsque l'on a affaire à des infections informatiques hostiles. Parce que le programme cible s'exécute sur le système hôte, il peut échapper au contrôle du débogueur (qu'il s'exécute en mode utilisateur ou en mode noyau) et se propager.

Force est de constater que les désassembleur statiques sont faciles à induire en erreur (cf. chapitre 3 l'utilisation de sauts à des adresses variables, ou [LD03] par exemple) et que les débogueurs sont faciles à détecter et éventuellement à contourner (cf. chapitre 3 section 3.2.2). L'exécution au sein d'un environnement contrôlé, comme une machine virtuelle, peut résoudre les problèmes d'isolation et de furtivité. Nous allons à présent présenter dans cette section plusieurs approches basées sur l'émulation ou la virtualisation permettant d'unpacker des programmes protégés de manière sécurisée et fiable.

Plusieurs algorithmes ont déjà été proposés, fondés sur l'analyse de la mémoire virtuelle et sur des hypothèses faites à propos des mécanismes de protection sous-jacents. Ces approches sont proches de la notre. Nous pouvons citer de nombreux projets qui focalisent sur des algorithmes alternatifs et utiles pour tracer l'utilisation mémoire d'un processus cible et éventuellement d'unpacker un programme auto-protégé. Parmi ceux-ci, nous allons focaliser sur trois approches intéressantes :

- *Algorithme d'analyse de la mémoire corrompue* : le projet Argos [PSB06b, PSB06c, PSB06a] utilise l'analyse de la mémoire corrompue [Bos06] de manière à détecter un programme hostile (et potentiellement auto-protégé) en mémoire. Argos est un émulateur du système conçu pour être utilisé dans les Honeypots [Por04]. Il utilise une version modifiée de QEMU [Bel05, Bal06] pour tracer les accès mémoire.

  Les auteurs d'Argos ont étendu l'émulateur QEMU pour lui permettre de détecter les tentatives d'accès à distance visant à compromettre le système invité émulé (comme par exemple le *shellcode* d'un ver). En utilisant l'analyse dynamique de la mémoire corrompue, Argos trace les données du réseau durant l'exécution du processeur et détecte toute tentative de les utiliser de manière malicieuse. Lorsqu'une attaque est détectée, un instantané de la mémoire est journalisé et l'émulateur s'arrête.

  Le principe de l'analyse de la mémoire corrompue est le suivant : les données provenant d'une source non sûre sont marquées comme étant corrompues (par exemple, les données provenant du réseau). Les données corrompues sont tracées durant l'exécution (lorsqu'elles sont copiées en mémoire ou dans un registre, le nouvel emplacement est également marqué comme étant corrompu). De manière à identifier et prévenir tout usage

[3] OllyScript plugin's script format

non sûr d'une donnée corrompue, une alarme est levée (lorsqu'elle est utilisée par exemple comme cible d'une instruction de branchement).

- *Algorithme de normalisation* : le projet WiSA utilise un algorithme de normalisation [CKJ+05] de manière à unpacker les virus avant d'appliquer un algorithme d'extraction de signature ou des transformations d'obfuscation. Ce projet utilise également une version modifiée de l'émulateur QEMU de manière à appliquer l'algorithme de normalisation.
  Leur approche est très intéressante mais est plus intrusive et complexe que la notre. L'émulateur est modifié de façon à surveiller la mémoire et à collecter toutes les écritures. Si une tentative d'exécuter du code depuis une zone mémoire qui a été précédemment écrite est faite par le programme cible, l'instruction de déclenchement est capturée et l'exécution est terminée. En utilisant les données capturées, un programme équivalent est reconstruit. Cet algorithme est appliqué itérativement jusqu'à ce que tout le code de la protection ait été supprimé.

  Pour qui souhaite implémenter et tester l'algorithme de normalisation du projet WiZA, un bon point de départ est la version modifiée de l'émulateur QEMU du projet Argos qui implémente tout ce qui est requis pour pouvoir tracer les accès mémoire.
  Malheureusement, le papier du projet WiZA n'explique pas comment reconstruire l'exécutable après unpacking. Nous présentons dans ce chapitre au moins une méthode qui peut être appliquée pour reconstruire un exécutable déprotégé après unpacking, basée sur le hook des fonctions d'API Win32 et natives.

- *Algorithme forensique* : la procédure d'unpacking couvre souvent deux problèmes :
  - premièrement, *dumper* la mémoire,
  - deuxièmement, partant de cet instantané de la mémoire, trouver les zones qui appartiennent au programme cible et reconstruire l'exécutable.

  Lorsque l'on fait un dump mémoire en aveugle, il est difficile de recouvrer un exécutable complètement fonctionnel. Cependant, des heuristiques forensiques peuvent donner de bons résultats et peuvent être appliquées au problème qui consiste à retrouver un processus dans un instantané mémoire et à retrouver une image binaire du processus cible.

  Parmi les outils d'analyse forensique les plus connus, nous trouvons les suivants intéressants : MemParser [Bet06], idetect, ProcEnum, WMFT [Bur06a, Bur06b, Bur05, Bur06c, Bur06d], Ramdump, lsproc, lspm, ReadPE [Car06b, Car06a], Dd, md5lib, md5sum, VolumeDump, Wipe, ZlibU, nc, GetOpt [Gar06], Kntlist [GM06], Dd for Windows [New06], Minidumps [Pen02b, Pen02a], les scripts PERL MemDump et PTFinder [Sch06f, Sch06h, Sch06i, Sch06d, Sch06a, Sch06b, Sch06c, Sch06g, Sch06e], Mdump [Wea03b, Wea03a], etc.

Avant de présenter la méthode que j'ai développée pour supprimer le loader de protection d'un exécutable packé puis reconstruire l'exécutable, après projection de son image mémoire, je signale que d'autres méthodes ont été proposées depuis la publication de mes résultats dans les actes de la conférence AAVAR, en 2006 [Jos06b]. Parmi celles-ci, la solution décrite dans [KPY07] (publiée dans les actes de la conférence WORM'07) me paraît la plus pertinente. Fondée sur l'utilisation de l'émulateur TEMU (nouveau nom donné à TTAnalyze [BKK06]), leur méthode est très similaire à celle proposée dans [CKJ+05], dans la mesure où elle se propose de détecter l'exécution d'un code nouvellement généré en surveillant les écritures en mémoire après démarrage du programme. À la différence du papier [CKJ+05], leur papier décrit complètement les choix de conception et d'implémentation : l'outil Renovo créé une structure de donnée comparable à celle des *page tables* pour maintenir une mémoire secondaire permettant de reproduire l'espace mémoire du processus observé. À chaque octet de cette mémoire est associé un bit, indiquant s'il l'octet correspondant est ou non corrompu. Cette mémoire est initialement considérée comme non corrompue. Durant l'exécution du programme, ses opérations d'écritures sont reproduites en mémoire secondaire, en marquant les zones correspondantes (les instruction d'écriture en mémoire sont instrumentées par l'émulateur TEMU). Avant l'exécution d'un bloc de code, un contrôle de la mémoire correspondante (occupée par ce bloc de code) est effectué. Si la mémoire est marquée comme corrompue, c'est qu'il s'agit d'une portion de code nouvellement générée. Dans ce cas, les pages contenant les octets marqués comme corrompus sont sauvegardées sur le système de fichier.
De façon à éviter les fausses alertes dues au chargement/déchargement dynamique de DLL dans l'espace d'adressage du processus, les écritures correspondantes ne sont pas marquées.

Pour extraire le code d'un programme packé par utilisation de plusieurs couches de chiffrement, la mémoire secondaire est remise à zéro avant de reprendre le processus de surveillance de la mémoire. Remarquons que la solution décrite dans [KPY07] ne propose pas de méthode pour recouvrer un exécutable fonctionnel. Leur objectif est de récupérer automatiquement le code chiffré sous forme déchiffrée.

Je pense que Renovo offre, concernant la fonction de restauration, des fonctionnalités équivalentes à celles que je décris dans la section suivante. Si elle peut à certains égards paraître plus élégante (notamment l'utilisation d'un masque associé aux octets de la mémoire secondaire quand ma solution impose une comparaison octet par octet de deux basics blocs), je pense qu'elle implémente une approche très similaire et se heurte donc aux mêmes limitations : lorsque le code protégé comporte du code auto-modifiable, nos deux approches se heurtent à un écueil incontournable, dans la mesure où nous ne sommes pas capables de distinguer le flot de contrôle de la protection de celui du programme protégé.
C'est malheureusement souvent le cas des programmes viraux.
Le tableau 5.2 présente de manière synthétique les différentes solutions, en terme de couverture des mécanismes de protection (X signifie que le produit est capable de contourner le mécanisme de protection). Ces mécanismes de protection sont décrits au chapitre 3 (section 3.2.2). Nous pouvons remarquer que les modules d'unpacking fondés sur

| Technique de protection | IDA Pro UUnP | OllyBone | WiZa | Renovo | VxStripper |
|---|---|---|---|---|---|
| Mesure des temps d'exécution | | | X | X | X |
| Contrôle d'intégrité du code | | | X | X | X |
| Mise à zéro des registres de déboguage | | | X | X | X |
| Auto-déboguage | | | X | X | X |
| Chiffrement simple | X | X | X | X | X |
| Chiffrement par couches | | | X | X | X |
| Détournement des interruptions | | | X | X | X |
| Redirection des appels aux fonctions importées par l'application | | | | | X |
| Redirection avec émulation partielle des imports | | | | | X |
| Redirection avec émulation totale des imports | | | | | |
| Altération de la section code | | | | | |
| Virtualisation de code | | | | | |
| Dissimulation du processus | | | X | X | X |
| Protection du processus | | | X | X | X |
| Altération de l'entête | | | | | X |

Tab. 5.2 – Récapitulatif des différentes solutions

l'utilisation d'un débogueur ring3 ne permettent pas de contourner systématiquement les mécanismes de protection implémentés dans les packers, i.e. qu'ils ne permettent généralement pas de supprimer le loader de protection de manière automatique. Une interaction humaine est donc le plus souvent indispensable.
Concernant les modules d'unpacking fondés sur l'utilisation d'un CPU logiciel, la résistance vis-à-vis des protection anti-déboguage, chiffrement et contrôle d'intégrité est identique. La solution VxStripper est la seule à proposer une fonction de reconstruction de l'exécutable, après projection de l'image mémoire. Pour les autres solutions, l'utilisation d'outils tiers est donc nécessaire. L'avantage d'intégrer les deux types d'outils (unpacker et reconstruction de l'exécutable) est que cela permet à la fonction de reconstruction d'utiliser une partie de l'information récupérée lors de la première phase de l'analyse (interaction avec les fonctions d'API en particulier).

Nous pouvons remarquer que VxStripper n'est pas capable de contourner efficacement les mécanismes de redirection avec émulation totale des imports et d'altération de la section code. Pour contrer efficacement ce mécanisme, il serait nécessaire d'implémenter quelques heuristiques supplémentaires, impliquant de mener l'analyse conjointe de l'exécutable protégé et de l'exécutable déprotégé, afin de recouvrer les portions de code manquantes (embarquées dans le code de la protection). Une solution algorithmique permettant de lever automatiquement ce type de pro-

tection reste un problème ouvert.

Nous avons présenté au chapitre 3 le mécanisme de protection par virtualisation de code et des travaux récents [GG09] menés pour lever presque automatiquement ce type de protection. Les auteurs utilisent une méthode d'analyse hybride statique/dynamique, fondée sur l'utilisation et l'extension de l'outil Metasm [Gui07]. VxStripper n'est pas capable aujourd'hui de contourner efficacement ce type de mécanisme.

Concernant le mécanisme de chiffrement par couches, il convient de remarquer que les algorithmes implémentés dans WiZa, Renovo et VxStripper souffrent de la même limitation lorsque le programme protégé comporte lui même du code auto-modifiable. Une analyse manuelle du programme est alors nécessaire pour identifier le point d'entrée original. Remarquons que cette phase d'analyse manuelle est considérablement simplifiée dans le cas des CPU logiciels Renovo et VxStripper, dans la mesure où ces outils tracent les interactions avec les API du système.

### 5.2.3 Notre approche

Avant de présenter le fonctionnement du moteur d'unpacking de VxStripper, je vais rappeler le fonctionnement d'un émulateur de PC, et celui de QEMU en particulier. Je vais essayer d'expliquer en partie l'engouement dont semble bénéficier cette solution auprès des chercheurs. Un émulateur de PC est avant tout constitué d'un émulateur de CPU. Un émulateur de CPU interprétatif, comme BOCHS, effectue une boucle principale infinie durant laquelle il interprète i.e. lit une instruction, la décode puis l'exécute. Cela pose un problème de performance, dans la mesure où au moins deux instructions de saut sont requise à chaque cycle.

QEMU est un émulateur de PC utilisant la translation binaire dynamique : le code écrit pour un jeu d'instructions CPU est traduit à la volée en un code pour un autre jeu d'instructions CPU. Ce mécanisme permet une exécution plus rapide, en permettant l'utilisation d'un cache : l'idée est de traduire un morceau de code, de le mettre en cache, et de le réutiliser si besoin est. L'unité d'exécution étant le *basic block*, un certain nombre d'optimisations deviennent possibles. Le principal intérêt d'un émulateur de PC fondé sur la traduction binaire dynamique est sa rapidité d'émulation.

Pour chaque processeur émulé par QEMU, la traduction suivante est effectuée : L'ensemble des instructions cible est traduit en une représentation intermédiaire (micro opérations) elle même traduite dans le jeu d'instruction hôte. Dans QEMU, la représentation intermédiaire est indépendante du jeu d'instruction hôte. Le moteur de traduction binaire dynamique se fonde sur cette représentation. Il est pour cette raison portable.

J'ai modifié le mécanisme de traduction binaire dynamique de telle sorte que ma fonction d'instrumentation du CPU virtuel soit invoquée systématiquement avant l'exécution d'un basic block. Pour y parvenir, j'ai ajouté une micro opération supplémentaire qui prends en opérande l'adresse de la fonction d'instrumentation dans le code de VxStripper.

#### Unpacking d'exécutable protégé

L'idée sous-jacente à mon algorithme de déprotection est un simple contrôle d'intégrité du code exécutable du programme cible.

L'algorithme 2 décrit le principe du processus d'unpacking.

```
Pour chaque TB faire
    Si (vcpu.eip = _entry_point (point d'entrée de la protection)) Alors
        _Dump_Init(vcpu.state,vcpu.eip) (initialisation du mapping)
        _IAT_Init(vcpu, PEB) (initialisation de la table triée des imports)
    Fin Si
    _Hook_OUT_Ring0 (instrumentation de certaines API natives - OUT)
    Si (vcpu.eip ∈ [espace_processus_inf, espace_processus_sup] ) Alors
        _Hook_OUT_Ring3 (instrumentation de certaines API Win32 - OUT)
        Si (Changement de section) Alors
            changement_section = 1
        Fin Si
        raw_addr=get_raw_addr(vcpu.eip) (récupération de l'adresse correspondant au pointeur d'ins-
        truction courant)
        Si (read_TB(vcpu.eip) ≠ read_File(raw_addr) (différence binaire)) Alors
            Si ((tainted_block == 0) ∨ (changement_section == 1)) Alors
                _Dump_Block(vcpu.state,vcpu.eip) (dump mémoire du basic block)
                _Fix_OEP(vcpu.eip) (modification de l'entête PE)
                _Fix_Section_Header(vcpu.eip)
                _Fix_IAT()
                tainted_block = 1
                changement_section = 0

            Sinon
                _Dump_Block(vcpu.state,vcpu.eip) (dump mémoire du basic block)
            Fin Si
        Sinon
            tainted_block = 0
        Fin Si
    Fin Si
    _Hook_IN (instrumentation de certaines API natives et Win32 - IN)

Fin Pour
```

Algorithme 2: L'algorithme d'unpacking

Le programme est exécuté dans l'environnement émulé. Pour chaque *basic block*, une comparaison entre sa valeur dans la mémoire virtuelle du système invité et sa valeur sur le système de fichier de l'hôte est effectuée. Tant que les valeurs sont identiques, rien n'est fait. Dès lors qu'une différence est identifiée, le *basic block* courant est écrit dans le fichier en lieu et place de l'ancien *basic block*.

La zone mémoire surveillée (par défaut) est l'image entière du programme cible lors de son chargement en mémoire :

```
[image_base,   image_base + virtual_sizeof_image]
```

Ce même algorithme de contrôle est appliqué pour chaque *basic block*. Le *loader* de protection de l'exécutable packé peut avoir plusieurs couches de chiffrement. Dès lors que le dernier *basic block* déchiffré a été atteint, les seules choses qui restent à faire consiste à projeter l'image mémoire du processus puis réparer l'exécutable cible[4].

[4] Observons que lorsque l'exécutable protégé comprend lui-même du code auto-modifiable, i.e. le programme exécutable implémente des protections anti- rétro-ingénierie au niveau du code source, il n'y a pas d'algorithme exact permettant d'obtenir une version déprotégée de l'exécutable original, i.e. sans son enveloppe de protection.
Dans ce cas, le processus d'unpacking doit être mené à la main, ou des heuristiques doivent être appliquées en faisant des hypothèses sur la routine d'unpacking et l'exécutable protégé, de façon à être capable de distinguer leurs flots de contrôle respectifs.

Finalement, on ne s'écarte pas fondamentalement des étapes traditionnelles d'une rétro-analyse manuelle, par utilisation d'un débogueur. On peut considérer que dès lors que l'on a dépassé le point d'entrée d'origine du programme protégé, une projection de l'image mémoire du processus sur le système de fichier peut être effectuée. Dans notre cas, ce dump mémoire ne se fait pas sans problème.

### Projection de l'image mémoire

Il est en fait parfois nécessaire d'injecter du code dans la VM pour obtenir une image mémoire complète. Rappelons que lorsque l'on souhaite accéder à une zone de la mémoire virtuelle, l'adresse virtuelle est d'abord convertie en adresse physique. Cependant, il est possible que tout ou partie de la zone mémoire correspondante soit *page out*, i.e. qu'elle soit sur le disque dur émulé. Par ailleurs, la MMU (*Memory Management Unit*) d'un système d'exploitation comme Windows utilise un mécanisme d'évaluation paresseux pour optimiser l'utilisation des ressources. Cela signifie qu'il n'effectue une tâche que lorsqu'elle est absolument requise[5]. Pour résoudre ce problème, l'idée consiste à injecter le code nécessaire pour induire une faute de page, laisser le gestionnaire de faute de page de l'OS faire son travail puis lire à nouveau la zone mémoire concernée[6].

Parce que nous ne sommes pas toujours intéressés par l'obtention d'un exécutable unpacké fonctionnel, mais plutôt par l'obtention d'une version du programme d'origine ne comportant plus aucune partie chiffrée, dans le but de désassembler statiquement une partie de son code ou de récupérer une information sur le code désassemblé, nous pouvons décider de conserver l'enveloppe de protection.

### Reconstruction de l'exécutable déprotégé

Cependant, il peut être utile après unpacking, de tenter de recouvrer certaines des informations qui ont été supprimées ou altérées durant le processus de déprotection.
De façon à recouvrer la structure PE d'un exécutable déprotégé, plusieurs tâches doivent être menées à bien :
- renseigner le point d'entrée original ;
- vérifier la cohérence de l'entête PE ;
- reconstruire l'information d'import ;
- éventuellement retrouver d'autres informations de l'entête, telle que la table des relocations.

Après projection de l'image mémoire du processus, la première étape est naturellement de modifier le point d'entrée dans l'entête PE du binaire cible.

D'après les spécifications du format PE [Mic06c], plusieurs parties de l'entête PE sont intéressantes.
Plusieurs champs de l'entête PE peuvent être modifiés de manière à perturber les outils d'analyse [Bru05]. Avec des modifications adéquates des champs `ImageBase`, `LoaderFlags`, `NumberOfRvaAndSizes`, etc., un débogueur peut très bien considérer que le binaire n'a pas une bonne structure et décider de lancer l'exécutable sans s'arrêter au point d'entrée. Cela peut être très gênant si un analyste veut déboguer une infection informatique sur son ordinateur, car il peut être infecté.
Les valeurs de certains champs de l'entête PE peuvent également être utilisés pour stocker des clés nécessaires au déchiffrement de quelques couches de la protection. Dans ce cas, toute modification de ces valeurs rendra l'exécutable non fonctionnel. Comme nous pouvons le constater, réparer l'exécutable peut être une tâche délicate.

Avant de présenter la méthode que j'ai développée pour retrouver l'information d'import (ce qui est le plus souvent suffisant pour recouvrer un exécutable fonctionnel), il est important de rappeler la manière dont cette information est retrouvée par le loader du système d'exploitation, lors du chargement d'un exécutable. L'entête PE prévoit deux

[5] A titre d'exemple, au début de la vie d'un processus, ses page tables ne contiennent généralement pas encore les bibliothèques partagées utilisées par ce processus. Ces pages tables ne sont mises à jour que lorsque le CPU référence effectivement la mémoire dans ces bibliothèques.

[6] Pour que cette injection de code soit furtive, elle ne doit pas modifier l'état du CPU. Parce que la fonction d'instrumentation du CPU est invoquée avant exécution du basic block courant, il est possible de ne pas exécuter ce dernier lorsqu'un accès en lecture à une zone mémoire n'est pas possible. Je remplace l'exécution du basic block courant par celle du bloc responsable de la faute de page. Après exécution de l'opération de lecture de la zone mémoire invalide injectée, puis exécution du gestionnaire de faute de page, le basic block d'origine est enfin exécuté.

tables, appelées IDT et IAT. La première contient l'information nécessaire au loader de l'OS pour renseigner la seconde :
- avant chargement de l'exécutable, l'IAT et les ILT sont identiques ;
- après chargement en mémoire, l'IAT contient les adresses des APIs importées par le programme.

L'IAT apparaît comme un invariant. Cependant, après dé-protection, les adresses qu'elle contient peuvent pointer par exemple :
- vers un tampon global alloué dynamiquement par la protection ;
- ou vers une zone du code de la protection.

Dans les deux cas, après projection d'une image mémoire, ces adresses ne sont plus accessibles. L'IAT est inexploitable. L'IDT est potentiellement complètement détruite.

Pour résoudre ce problème, l'idée est la suivante : avant l'exécution d'un bloc de code, on analyse la dernière instruction. S'il s'agit d'un appel de fonction ou d'un saut, on analyse l'opérande :
- s'il s'agit d'une adresse dans le code du programme, on la désigne comme offset candidat dans l'IAT ;
- s'il s'agit d'un registre, on récupère un offset candidat dans l'IAT par une méthode de slicing dynamique fondée par exemple sur une analyse de la mémoire corrompue.

Après exécution du bloc de code, s'il s'agit d'un appel vers une fonction d'API, je renseigne l'IDT avec cette fonction d'API.

En résumé, l'analyse dynamique et l'instrumentation des APIs nous fournissent, pour chaque fonction importée :
- le nom de la DLL ;
- le nom de la fonction importée ;
- un offset dans l'IAT que l'on va exploiter pour reconstruire l'IDT, moyennant une simple transformation affine.

Voyons à présent les différentes étapes permettant par instrumentation des APIs de récupérer l'information nécessaire à la reconstruction d'une entrée de l'ILT correspondant à une fonction de bibliothèque dans l'IDT.
Nous localisons dans le PEB la liste des modules chargés dans l'espace du processus.
Pour chaque DLL, nous analysons la structure `IMAGE_DIRECTORY_ENTRY_EXPORT` de l'entête.
Nous utilisons le nom de la DLL et celui de la fonction pour déterminer l'ordinal de la fonction d'import. Cet index nous permet de retrouver :
- soit l'adresse de la fonction dans le code de la DLL ;
- soit l'adresse d'une *forwarder chain* donnant le nom et le module dans laquelle se trouve effectivement le code la fonction.

Dans ce dernier cas, l'adresse de la fonction est recherchée dans cette autre DLL.

A l'issue de cette analyse purement dynamique, on a retrouvé l'ensemble des fonctions importées utilisées durant une exécution contrôlée du programme.

Remarquons que reconstruire l'IDT n'est pas la seule option : une autre option consiste à adjoindre un loader à l'application déprotégée, chargé de reconstruire l'IAT après résolution dynamique des APIs importées.

Remarquons également que des heuristiques supplémentaires peuvent certainement être appliquées pour augmenter le taux de recouvrement de l'IAT ou de l'IDT, en tentant d'abord de localiser puis d'analyser le wrapper de redirection. Rappelons cependant que ce dernier peut être situé n'importe où en mémoire. En outre, ce dernier peut lui-même servir de tremplin vers un autre morceau de code chargé d'invoquer la fonction d'API. Le nombre de degrés d'indirections est potentiellement arbitraire. Enfin, les entrées des tables de redirection ne sont pas nécessairement adjacentes en mémoire. On peut même imaginer que leurs emplacements soient aléatoires. Dans ces conditions, l'augmentation du taux de couverture peut être ardu.
Une autre idée consiste à forcer l'exploration de plusieurs chemins d'exécutions [MKK07].

La méthode utilisée par notre moteur d'unpacking pour reconstruire l'information d'import est donc fondée sur le hook des API natives et Win32.
Durant le processus d'unpacking, tous les appels aux fonctions d'API sont tracés. Une table triée des adresses correspondantes est initialisée au chargement du programme en explorant les structures internes de l'exécutif NT. Ensuite, après reprise de l'exécution du processus cible, chaque appel à une fonction d'API est tracé. Cette table est mise à jour régulièrement durant l'exécution du processus cible, et est utilisée pour résoudre dynamiquement les noms de fonctions d'API invoquées par le programme. Finalement, après projection de l'image mémoire du processus cible sur le système de fichier, cette table est utilisée pour fixer l'IDT dans l'exécutable PE.
À cette étape de la reconstruction de l'exécutable déprotégé, nous fixons également les relocations. Cette information est très utile avant d'appliquer des transformations d'obfuscation sur du code binaire[7].

J'utilise ici des heuristiques très similaires à celles implémentées par MackT [Mac06] et y0da [Y0d07] dans leurs moteurs de reconstruction d'exécutable PE. La table des imports est triée de manière à contourner certaines protections qui émulent les premiers octets des fonctions d'API et par conséquent n'effectue par de saut vers le point d'entrée original des fonctions d'API. En maintenant une table triée des fonctions d'API, les fonctions d'API ne sont plus indexées par leur point d'entrée mais par un intervalle d'adresses en mémoire. Nous sommes alors en mesure de tracer les appels de fonctions d'API même si les premiers octets ont été volés et déplacés dans la protection.

À l'issue de cette dernière étape, nous obtenons une version déprotégée de la cible. À présent, le binaire peut être chargé par le système d'exploitation sans déclencher d'erreur. Le binaire résultant est un exécutable complètement fonctionnel.

Plusieurs heuristiques peuvent être appliquées afin de distinguer le flot de contrôle de la protection de celui de l'exécutable d'origine. Le problème correspondant est de déterminer où est localisé en mémoire le point d'entrée original du programme protégé. Ce problème est probablement indécidable. Mais l'utilisation conjointe de notre heuristique de contrôle d'intégrité avec plusieurs autres heuristiques classiques (surveillance des sauts inter-sections, hooks des fonctions d'API résolues au démarrage comme dans le plug-in IDA uunp, etc.) donne de bons résultats. Si l'utilisation des heuristiques échoue, le dernier ressort est un processus d'unpacking dirigé par l'humain.

Lorsqu'elle est conduite par un humain, la procédure d'unpacking est souvent menée en deux étapes :
- D'abord, déterminer le point d'entrée original, à partir de l'information fournie par une première exécution contrôlée de l'exécutable cible.
- Ensuite, poser un point d'arrêt virtuel au point d'entrée original supposé et exécuter le moteur d'unpacking à nouveau.

### 5.2.4 Quelles sont les limitations ?

Les programmes d'unpacking se heurtent classiquement à plusieurs limitations :
- En premier lieu, il est parfois difficile de reconstruire un exécutable même après qu'une procédure d'unpacking ait été appliquée avec succès.
- En second lieu, des mécanismes de protection plus sophistiqués peuvent interdirent l'analyse d'un binaire et donc faire échouer une procédure d'unpacking [BF07, Fil05c].

Notre méthode peut être appliquée efficacement sur des exécutables simplement packés, mais doit être adaptée pour cibler des exécutable qui implémentent des mécanismes de protection plus sophistiqués, comme le chiffrement par couche ou la génération environnementale de clé/code, les routines avancées de détection d'émulateur (scan des fonctionnalités matérielles par exemple), $\tau$-obfuscation [BF07].
Ces limitations s'appliquent à tout outil d'analyse fondé sur l'émulation de PC et seront discutées dans la section 5.3 de ce chapitre (les méthodes de détection d'un environnement émulé par un programme protégé seront discutées dans la section 5.3.4).

[7] En général, cette étape de recouvrement de l'information de relocation n'est pas requise. Elle concerne plus souvent les DLL. Nous travaillons sur ce problème uniquement parce que cette information est indispensable pour plusieurs moteurs d'obfuscation, comme le moteur Mistfall de Z0mbie [Z0m00b]. L'idée est de calculer une différence binaire entre les dumps mémoires correspondant à plusieurs relocation en mémoire de l'exécutable ou de la DLL cible.

## 5.3 Vers un outil d'analyse fiable

Dans la section 5.2 de ce chapitre, nous avons répondu à la question : comment récupérer automatiquement de l'information d'une infection informatique packée ? Plusieurs algorithmes et heuristiques nous permettant de déchiffrer son code ont été présentées. Le propos de cette section est de répondre à la question suivante : comment récupérer automatiquement une information fiable d'une infection informatique en utilisant des techniques furtives et sûres ?

Parmi les informations qui peuvent être extraites d'un exécutable, la plus importante est relative à la nature de ses interactions avec le système d'exploitation. Notre approche nous conduit au minimum à être en mesure de tracer à la fois les appels aux fonctions des API natives et Win32. Être furtif impose d'être en mesure d'aller le plus profondément possible dans le système d'exploitation.

*Comment hooker furtivement les API natives et Win32 ?* Pour hooker les API Win32, plusieurs méthodes s'appliquent en mode utilisateur. Le problème est que quelle que soit la méthode choisie, le hook des API en mode utilisateur n'est ni furtif ni fiable.
Pour hooker à la fois les API natives et Wn32 [Neb00], les meilleurs méthodes reposent sur l'installation d'un pilote. Le hook des API en mode noyau est précis. Cependant, il n'est pas absolument furtif (cf. [Hog05], dernier chapitre). L'utilisation d'un émulateur est une approche complémentaire pour hooker les API. Cette approche est à la fois précise et furtive.

Dans cette section, nous présentons tout d'abord des méthodes pour hooker les API natives et Win32 lorsque l'exécutable cible est confiné dans une machine virtuelle. Nous discutons ensuite des aspects liés à la sécurité de l'analyse d'infections informatiques et des techniques qui peuvent s'appliquer pour augmenter le niveau d'isolation d'une plate-forme de tests. Finalement, nous présentons l'outil d'analyse VxStripper et discutons de l'avantage d'un processus de détection reposant sur une décision humaine.

### 5.3.1 Trois moyens d'injecter du code (furtivement) dans une machine virtuelle

Nous présentons dans cette section plusieurs méthodes qui peuvent être utilisées pour instrumenter l'espace d'exécution d'un processus cible. L'utilisation d'un émulateur fournit un intervalle de possibilités puissant pour l'observation du flot d'exécution d'un processus. Des hooks peuvent être insérés furtivement et du code peut être injectés en utilisant au moins trois mécanismes :
- Injection de *shellcode* forensique
- Injection de code intermédiaire forgé manuellement
- L'ancienne méthode (réécriture de code en mode noyau à l'intérieur de l'émulateur).

#### *shellcode* forensique

Une première méthode pour injecter du code dans une machine virtuelle consiste à écrire le code directement en mémoire virtuelle avant de rediriger le pointeur d'instruction du CPU virtuel vers cette zone mémoire.
Argos [PSB06a] implémente un injecteur de *shellcode* forensique [`Argos-0.1.4/target-i386/argos-csi.c`]. Un extrait du code source de l'injecteur de *shellcode* d'Argos est donné ci-dessous :

```
#define WIN32SC_RID_OFF 257
#define WIN32SC_LENGTH 293

static char win32_shellcode[] = "...";

void argos_csi(CPUX86State *env, int type, argos_rtag_t *eiptag) {
...
int rid = rand();
...
argos_inject_sc(env, type, rid);
...
}
static void argos_inject_sc(CPUX86State *env, int type, int rid){
...
```

```
sclen = WIN32SC_LENGTH;
scp = win32_shellcode;
scrid = WIN32SC_RID_OFF;
...
vaddr = env->eip & TARGET_PAGE_MASK;
if ((paddr = cpu_get_phys_page_debug(env, vaddr)) == -1)
goto address_lookup;
while (!ARGOS_MAP_ISTAINTED(argos_memmap, paddr) &&
(paddr & TARGET_PAGE_MASK) != -1)
paddr++;
for (tlen = 0; ARGOS_MAP_ISTAINTED(argos_memmap, paddr);
paddr++, tlen++)
;
if (tlen >= sclen)
goto code_inject;
address_lookup:
vaddr = 0;
paddr = find_page(env, &vaddr, max_vaddr, PG_USER_MASK | PG_RW_MASK,
PG_USER_MASK);
if (paddr == -1)
{
printf("[ARGOS] Forensics shellcode will not be injected - "
"No available page found\n");
return;
}

code_inject:
cpu_physical_memory_rw(paddr + (TARGET_PAGE_SIZE - sclen), scp,
sclen, 1);
cpu_physical_memory_rw(paddr + (TARGET_PAGE_SIZE - sclen) + scrid,
(unsigned char *)&rid, 4, 1);
env->eip = vaddr + (TARGET_PAGE_SIZE - sclen);
}
```

Le *shellcode* forensique est injecté, remplaçant le *shellcode* viral, de manière à récupérer de l'information relative au processus attaqué. Lorsqu'une attaque est détectée, un *shellcode* spécifique à l'OS est injecté. En d'autres termes, le code attaqué est exploité, alors que l'attaque est en cours, pour extraire une information supplémentaire relative à l'attaque qui est ensuite utilisée pour générer une signature virale.
Après avoir détecté une attaque et journalisé l'état de la machine virtuelle, le *shellcode* forensique est placé directement dans l'espace d'adressage virtuel du processus. L'emplacement où est injecté le code est la dernière page du segment texte au début de l'espace d'adressage. Il est important de placer le code dans le segment texte de manière à garantir qu'il ne sera pas réécrit par le processus, puisqu'il est en lecture seule. Une fois le *shellcode* en place, il ne reste plus qu'à faire pointer le registre EIP vers sa première instruction pour démarrer son exécution.
À titre d'exemple, un *shellcode* qui extrait le PID du processus victime, et le transmet via une connexion TCP a été implémenté dans le projet Argos.

### Code intermédiaire forgé manuellement

Une autre manière d'injecter du code dans une machine virtuelle consiste à l'injecter au niveau du CPU virtuel. Nous présentons d'abord dans cette section les principes de conceptions d'un CPU virtuel basé sur le translation binaire dynamique [Tro04], puis expliquons l'utiliser pour injecter un code VCPU forgé manuellement.

La translation binaire dynamique permet de transformer *à la volée* des séquences d'instructions en code natif pour le CPU. Lorsque le code natif est mis en cache ou optimisé, il peut s'exécuter de manière significativement plus rapide.

Le moteur de translation dynamique effectue au moment du démarrage une conversion des instructions du CPU cible en ensemble d'instructions pour le CPU hôte. Le code binaire résultant est stocké dans un cache, appelé cache de translation, de telle sorte qu'il peut être réutilisé. L'avantage de cette méthode, comparée à celle de l'utilisation d'un interpréteur est que les instructions sont recherchées et décodées qu'une seule fois.

Les moteurs de translation dynamique sont en général difficiles à porter d'un hôte à un autre car tout le générateur de code doit être réécrit. Cela représente environ la même quantité de travail que l'ajout d'une nouvelle cible à un compilateur C. QEMU [Bel05] est plus simple car il concatène des morceaux de code générés au préalable par le compilateur GNU C.
La première étape consiste à découper chaque instruction CPU cible en plusieurs instructions plus simples appelées micro-opérations. Chaque micro-opération est implémentée par un petit morceau de code C. Ce petit code source C est compilé par GCC en un fichier objet. Les micro-opérations sont choisies de telle sorte que leur nombre est beaucoup plus petit (typiquement, quelques centaines) que toutes les combinaisons d'opérations et d'opérandes du CPU cible. La translation des instructions CPU cible vers les micro-opérations est faite entièrement par un codage manuel.
Un outil de compilation appelé `dyngen` [Bal06] utilise les fichiers objets contenant les micro-opérations en entrée pour fabriquer un générateur de code dynamique. Ce générateur de code dynamique est invoqué au moment de l'exécution pour générer une fonction hôte complète qui concatène plusieurs micro-opérations.

De manière à injecter un code qui déclenche une faute de page dans le système d'exploitation invité, les auteurs de TTAnalyze [BKK06] décrivent une méthode qui peut être implémentée de la manière suivante dans QEMU :

```
uint16_t opc_buf[OPC_BUF_SIZE]; // intermediate code
uint32_t opparam_buf[OPPARAM_BUF_SIZE]; //
uint8_t code_buf[4096]; // host code

int code_size; // generated host code size

if (cpu_memory_rw_debug(env, addr, buf, len, is_write)!=0 && is_write == 0){

opc_buf[0]=INDEX_op_movl_A0_im;
opparam_buf[0]=addr;
opc_buf[1]=INDEX_op_ldsb_user_T0_A0;
opparam_buf[1]=0;
opc_buf[2]=INDEX_op_exit_tb;
opparam_buf[2]=0;
opc_buf[3]=INDEX_op_end;
opparam_buf[3]=0;

code_size=dyngen_code(code_buf, NULL, NULL,
                      opc_buf, opparam_buf, NULL);
                     // intermediate code to host code translation

// TB Execution
gen_func=(void *)code_buf;
gen_func();
```

L'injection de code intermédiaire forgé manuellement est un moyen très intéressant pour injecter du code furtivement dans une machine virtuelle. Mais à la différence de l'injection de *shellcode* forensique, il est difficile de forger des fonctions complexes ou de réutiliser des fonctions d'instrumentation existantes.

Enfin, nous présentons l'ancien moyen d'injecter du code de manière fiable dans un processus. Cette méthode s'applique également au contexte d'un outil d'analyse basé sur l'émulation. Même s'il est plus facile pour un programme de détecter la présence de hooks d'API en mode noyau au sein du fonctionnement interne de l'exécutif NT, les méthodes sous-jacentes sont plus documentées et aisées à implémenter.

#### Réécriture de code en mode noyau à l'intérieur de l'émulateur

Le hook en mode noyau est un troisième moyen intéressant pour injecter du code dans un processus cible. S'il est moins furtif que les méthodes décrites précédemment, les méthodes sont déjà bien documentées et ont prouvé leur efficacité vis-à-vis d'infections informatiques pas trop évoluées, donc en fait la plupart d'entre elles.

### 5.3.2 Vers un moteur de désassemblage fiable (un moyen d'améliorer le moteur de recherche de motif d'un produit antivirus ?)

L'utilisation de l'émulation fournit un moyen fiable pour obtenir le code désassemblé d'un programme protégé. La plupart des protections anti-désassemblage ne sont pas efficaces contre un tel outil d'analyse dynamique de code, car le processus de désassemblage est effectué pour chaque *basic block* de code. Les outils d'analyse statique de code doivent appréhender le code exécutable dans son intégralité en tant qu'unité d'analyse.
L'obtention du code désassemblé exact correspondant au flot d'exécution du programme cible est crucial pour pouvoir extraire l'information relative au fonctionnement interne d'un programme viral. C'est également primordial pour pouvoir forger un ensemble viral qui pourra ensuite être utilisée lors de tests de résistance en boîte noire d'un produit antivirus vis-à-vis de transformations classiques (packing, obfuscation).
Même si nous n'obtenons pas toujours une version complètement fonctionnelle d'un virus après unpacking, nous pouvons observer que le taux de détection de la plupart des programmes informatiques s'améliore sur un ensemble viral lorsqu'une transformation d'unpacking est appliquée.
Ce résultat est intuitif et confirme le fait que la plupart des techniques de détections courantes implémentées par les produits antivirus peuvent être réduites à de la recherche de motif [Fil06, Fil07c] et que le processus d'unpacking implémenté par leur moteur d'émulation souffre de limitations.

L'utilisation de mécanismes d'émulation par translation dynamique au sein d'un produit antivirus a été étudié dans [Ste05]. Dans cette étude, l'auteur propose un algorithme permettant à un antivirus de rechercher un motif dans chaque *translation block* durant l'exécution. Il défend également la possibilité d'utiliser un tel émulateur dans les produits antivirus du marché.

### 5.3.3 Pourquoi utiliser un émulateur comme méthode de hook complémentaire ?

L'utilisation d'un émulateur est une méthode complémentaire pour comprendre les mécanismes de protection d'une infection informatique. Par exemple, si un *hooker* en mode utilisateur est détecté par le processus cible, l'analyste de sécurité peut récupérer une information utile expliquant l'échec de son analyse : la cible implémente des mécanismes de protection à certains points de son exécution, comme un contrôle d'intégrité par exemple.
Si l'utilisation d'un émulateur pour tracer les appels aux fonctions d'API est une méthode puissante, elle souffre de limitations spécifiques, comme ne pas être immune à des mécanismes anti-émulation spécifiques. Par conséquent, un analyste de sécurité doit avoir d'autres outils à sa disposition de manière à contourner tous les mécanismes de protection implémentés par une infection informatique.

### 5.3.4 Comment accroître le niveau de sécurité d'un émulateur ?

Lors de leur analyse de la capacité de l'architecture Intel Pentium à supporter un VMM sûr [RI00], les auteurs conclurent que les VMM actuels pour l'architecture Intel ne devraient pas être utilisés pour faire appliquer des politiques de sécurité critiques. En outre, il serait imprudent d'essayer d'implémenter un moniteur de machine virtuelle à haut niveau d'assurance en tant que VMM de type II ou CSIM hébergée sur un système d'exploitation commercial générique. Asseoir un VMM hautement sécurisé sur un système d'exploitation qui ne se conforme pas aux critères de référence d'un moniteur ne fournira pas un haut niveau de sécurité.
Les VMM/CSIM actuels ne sont pas conformes aux exigences de sécurité d'une assurance élevée bien que certains vendeurs affirment que la sécurité est intrinsèque. Par exemple, des problèmes potentiels existent lorsqu'un VMM est utilisé pour séparer des domaines de sensibilité différentes ou pour compartimenter en zones un réseau.

Dans le contexte de l'analyse d'infections informatiques, l'objectif est de séparer et d'isoler deux domaines : le domaine invité, contrôlé, où le programme cible est exécuté et isolé, et le domaine du système hôte. Chacun de ces deux domaines peut mettre en jeu plusieurs VM et composants réseau physiques ou virtuels.

Un problème résulte de l'utilisation de ressources partagées entre machines virtuelles ou entre systèmes d'exploitation hôte et invité. Si les systèmes hôte et invité ont accès au même lecteur de disquette, périphérique USB ou CD/DVDROM, l'information peut fuire d'un système à l'autre.
Un problème similaire résulte du support réseau ou du partage de fichier. Des transferts de fichiers ou la communication d'informations entre OS hôte et invité sont possibles par ce biais.

Les outils qui sont embarqués dans l'OS invité peuvent également causer des problèmes. Par exemple, après avoir installé les outils VMware dans un OS invité, une fonctionnalité est la possibilité de couper/coller entre les bureaux des systèmes d'exploitation hôte et invité.

De nombreuses vulnérabilités apparaissent en raison du manque d'assurance disponible dans l'OS hôte sous-jacent. Des failles dans la conception et l'implémentation de l'OS hôte peuvent rendre l'OS invité vulnérable. Pour atteindre un certain niveau d'assurance, un OS hôte sécurisé est requis.

Par ailleurs, la virtualisation n'est pas nécessairement complète. En effet, pour des raisons de performance, il peut être intéressant de laisser un système invité accéder directement à certaines ressources du système sans passer par la couche d'abstraction matérielle. Il peut par exemple être intéressant de laisser au domaine invité la possibilité d'accéder directement à certains ports PIO. Dans sa thèse [Duf07], L. Duflot montre notamment que, bien souvent, les privilèges d'entrées-sorties sont suffisants pour qu'une entité logicielle puisse obtenir les privilèges maximaux sur le système. Ainsi donc, tous les systèmes de virtualisation qui déléguent des privilèges d'entrées-sorties à l'un ou l'autre de ses compartiments invités mettent en danger leur capacité à faire respecter leur politique de sécurité et à empêcher une escalade de privilège depuis un domaine invité non privilégié vers des privilèges maximaux sur le système. De telles considérations devront être prises en compte lors de la construction ou de la configuration des futures solutions basées sur la virtualisation.

Cependant, les vendeurs de VMM essayent d'augmenter le niveau de sécurité de leurs produits, en créant et renforçant les fonctions de sécurité qui sont nécessaires pour couvrir les vulnérabilités mentionnées ci-dessus. (par exemple, VMware ACE - *Assured Computing Environment* [VMw07]).
Partant de leur analyse de sécurité, nous avons implémenté plusieurs fonctions de sécurité supplémentaires qui sont obligatoires pour accroître le niveau de sécurité de l'outil VxStripper.

#### Politiques de gestion des droits virtuels

Pour être isolé et sécurisé, un émulateur doit implémenter une protection robuste des périphériques, des mécanismes de filtrage et de quarantaine réseau.

#### Politique de quarantaine réseau

De manière à accroître la sécurité d'un émulateur, un pare-feu doit être embarqué dans la machine virtuelle. En outre, un pare-feu doit également être installé sur le système hôte, de manière à contrôler les interfaces de communication réseau entre systèmes d'exploitation hôte et invité.

#### Moniteur temps réel

Un moniteur temps réel doit être installé sur le système hôte de manière à se prémunir de toute exécution accidentelle et pour bloquer toute propagation d'une infection informatique à partir de la station de travail de l'analyste de sécurité.

Nativement, QEMU [Bal06] ne supporte pas les communication invité vers hôte ou hôte vers invité car il est conçu pour se comporter comme une machine isolée. Il est possible de charger l'exécutable dans la VM, sans utiliser un quelconque canal de communication, en utilisant des techniques forensiques : l'idée est d'écrire les fichiers cibles directement sur le disque virtuel brut. Cette modification du mode de fonctionnement de VxStripper est envisagée comme tâche future.

#### Détection d'un environnement émulé par un virus

Nous allons à présent rappeler le problème de détection d'un environnement émulé par un virus, qui est crucial dans le contexte de la détection de virus par émulation : puisque savoir que l'on s'exécute sur une machine virtuelle est la première étape avant toute tentative d'attaque virale par évasion, un émulateur sécurisé doit simuler les composants matériels (CPU, chipset, etc.) aussi précisément que possible et doit en particulier être résistant aux algorithmes de

reconnaissance par recherche de motifs et tout scan des fonctionnalités du matériel[8]. Il est très difficile de simuler parfaitement en logiciel les composants matériels d'un PC.

D'autres mécanismes peuvent être implémentés par une infection informatique pour reconnaître un environnement émulé. Plusieurs méthodes de détection sont documentées depuis longtemps pour la détection des VMM (Virtual Machine Monitor) du commerce comme VMware ou VirtualPC [Z0m00c, Rut05b, Eli05]. De nouvelles méthodes ont été proposées récemment pour la détection de VMware, VirtualPC, Bochs [Law07], Hydra, QEMU, Plex86 [L+01] et Xen [Fer06, Lau08].
Nous décrirons en détail la méthode proposée par C. Lauradoux, fondée sur le contrôle de modèle, au chapitre 6, (section 6.3.11).
Selon l'auteur de [Fer06], seule l'approche CSIM (*Complete Software Interpreter Machine*) peut atteindre une transparence complète. Les CSIM comme Bochs [Law07], Hydra, et QEMU [Bel05, Bal06] souffrent tous d'erreurs et de limitations qui rendent possible leur détection, mais ces problèmes peuvent être potentiellement corrigés.

Concernant l'émulateur QEMU, parmi les problèmes les plus fréquemment rencontrés, en raison d'un émulation incorrecte, on trouve notamment :
- des problèmes dans l'émulation du code automodifiable (une séquence d'instructions `rep` se réécrivant elles mêmes par exemple) en raison de l'utilisation de la translation dynamique,
- des valeurs de retour erronées pour certaines instructions CPU (mauvaise valeur de retour de l'instruction `cpuid` concernant le nom du processeur ou Easter egg sur le CPU AMD par exemple),
- comportement inattendu résultant de l'exécution de certaines instructions CPU (l'instruction `CMPXCHG8B` n'écrit pas toujours en mémoire, par exemple),
- limitations dans le code de gestion des exceptions (la double exception de faute n'est pas supportée, par exemple).

De manière à être aussi furtif que possible, les erreurs et limitations au coeur du moteur d'émulation doivent être corrigées. Il y a donc du travail à réaliser pour accroître le niveau de sécurité d'un outil d'analyse virale fondé sur un moteur d'émulation.

### 5.3.5 Architecture de VxStripper et expérimentations

Nous décrivons dans cette section la conception et l'implémentation de l'outil d'analyse VxStripper et donnons les résultats obtenus en utilisant la version courante de ce programme.

#### Architecture et implémentation

Je travaille au développement d'un outil (utilisant Lex & Yacc) qui, partant d'un ensemble de prototypes de fonctions et de déclarations de structures, pourrait forger les fonctions de détournement des appels aux API (partant du fichier PDB de `NtOsKrnl`). Cet outil pourrait également être utilisé pour générer automatiquement le code des hooks de Detours ou de tout autre programme d'instrumentation des API par réécriture de code (en mode utilisateur ou noyau).

Par exemple, étant donné le prototype d'une fonction :

```
NTSTATUS ZwLoadDriver(
IN PUNICODE_STRING DriverServiceName
);
```

nous voulons générer automatiquement le code d'instrumentation :

```
if (addrMatch("ZwLoadDriver")){
ulong _addr0;
wchar_t _arg0[255];
uint8_t buffer[255];
```

[8] Un scan des fonctionnalités du matériel est une méthode d'authentification proposée par Microsoft. Le pilote teste le fonctionnement d'un composant matériel en exerçant des stimuli complexes et vérifie la correction des réponses. De cette manière, le pilote peut authentifier le fait que le chip est un matériel valide.

```
readVM(env, (ulong)(vcpu->regs[R_ESP]+4), buffer, 4, 0);
_addr0=*(DWORD*)buffer;
readVM(env, (unsigned long int)(_addr0+2*sizeof(USHORT)), buffer, 4, 0);
_addr0=*(DWORD*)buffer;
readVM(env, _addr0, (uint8_t *)_arg0, 255, 0);
fwprintf(logfile, L"[HOOK]  ZwLoadDriver[IN]  ObjectName=%s\n", _arg0);
ZwLoadDriver_WAS_CALLED=1;
}
```

J'ai adapté l'émulateur QEMU de manière à implémenter le moteur d'émulation qui est au coeur de l'outil VxStripper. D'autres émulateurs pourraient avoir été utilisés, comme Bochs [Law07] ou Plex86 [L+01]. J'étudie actuellement la possibilité d'utiliser un VMM de type I, comme Xen [MDD+08], en tant que composant central de l'outil d'analyse, pour des raisons de performances principalement.

La machine virtuelle embarque un service noyau qui communique via une interface réseau virtuelle avec le moniteur de machine virtuelle. Ce canal de communication est utilisé pour transférer les binaires cibles dans la machine virtuelle, pour démarrer l'exécution du programme cible principal et pour récupérer l'information du noyau qui permet de diriger l'exécution du processus invité depuis le système hôte.
Cette information est localisée dans les structures et tables de exécutif NT du système invité. La nature de cette information et la manière de la récupérer en mode noyau est bien documentée dans [SR00, RS05].
Le système hôte doit être protégé par un pare-feu et par un moniteur temps-réel, de manière à contrôler à la fois le canal de communication réseau virtuel et d'empêcher l'analyste de sécurité de commettre une erreur irrévocable.

L'outil VxStripper récupère de l'information sur :
- les appels aux API natives et Win32 du programme cible
- le code désassemblé (séquentiellement) du programme cible
- la structure de l'exécutable en mémoire invitée (et le résultat de la comparaison dynamique avec le binaire sur le système de fichiers de l'hôte)

Le fichier journal pourrait être donné (dans l'avenir) dans le format standardisé IDMEF, de manière à faciliter son utilisation par un moteur de corrélation tiers ou par un composant spécialisé dans la détection d'intrusion.

Le programme principal peut être utilisé dans deux modes consoles :
- le mode par défaut télécharge automatiquement l'exécutable cible dans la VM, l'unpack et récupère l'information relative à ses interactions avec le système d'exploitation invité ;
- le mode interactif permet à un analyste de sécurité de diriger dynamiquement l'exécution du programme cible et d'interagir avec la VM, en contrôlant ses états.

Le mode interactif est principalement utilisé lorsque l'exécutable cible lance un de ses composants en tant que nouveau processus, de manière à tracer son exécution. Lorsque l'on passe en mode interactif, les commandes suivantes sont utilisables :

```
C:\PARANO> make si
[INFO] System initialisation ...
[INFO] Connection to server (192.168.100.2) ...
P> help
[USAGE] command [args]
              vbp <virtual break point>
              upload <file>
              setmin <ctrl vmem min boundary addr>
              setmax <ctrl vmem max boundary addr>
              logapicall
              logdisasm
              logverbose
              unpack
                setoep
                execsuspend <file args>
```

```
                setpdb <pid>
                setpeb <pid>
                geteprocess <pid>
                summary
                map <file>
                resumeexec
                unmap
                disconnect
P>
```

Le programme principal peut également être utilisé en mode graphique. Ce mode est utile lorsque le processus d'analyse nécessite une interaction entre l'utilisateur et le programme cible par le biais d'une interface graphique. Les mêmes options qu'en mode console (mode graphique par défaut ou interactif) sont utilisables.
Parmi les informations qui peuvent être récupérées en utilisant VxStripper, on peut trouver des informations sur :

- les fonctions de bibliothèque chargées par le *loader* du système d'exploitation (IAT)
- les fonctions de bibliothèque chargées dynamiquement (`LoadLibraryA`, `LoadLibraryExA`, `LoadLibraryExW`, `GetProcAddress`, etc.)
- les modifications effectuées dans le registre et les interactions avec le gestionnaire de services (`ZwOpenKey`, `ZwSetValueKey`, les fonctions `AdvAPI` relatives aux actions sur le registre NT et aux services, etc.)
- les modifications faites sur le système de fichier (`ZwCreateFile`, `NtCreateFile`, etc.)
- l'installation de pilotes et les interactions entre espace utilisateur et pile de pilotes (`ZwLoadDriver`, `ZwSetSystemInformatio` fonctions de l'API `FltLib` relatives aux pilotes filtres IFS, fonctions de l'API du noyau relative aux IOCTL, fonctions de l'API `SetupAPI` pour l'installation des pilotes filtres, etc.)
- les interactions avec le bus d'objets exécutables par le biais des appels aux fonction de l'API `OLE32`
- les connexions réseau par le biais des fonctions d'API `Winsock2`
- etc.

À titre d'exemple, l'utilisation de fichiers SHF (*Super Hidden Files*) peut être aisément détecté. L'extrait suivant du fichier journal illustre l'utilisation de cette méthode par un virus :

```
[HOOK]    ZwOpenSection[IN]  ObjectName=\NLS\NlsSectionCType
[HOOK]    ZwOpenSection[OUT] NTSTATUS=0000007D
[HOOK]    ZwCreateFile[IN]  ObjectName=Hidden
[HOOK]    ZwCreateFile[IN]  FileAttributes=00000006 |FILE_ATTRIBUTE_SYSTEM
|FILE_ATTRIBUTE_HIDDEN |
[HOOK]    ZwCreateFile[OUT] NTSTATUS=00000025
```

Une infection informatique peut être installée en tant que pilote de périphérique et démarrée avec `ZwLoadDriver` ou peut être chargée directement par l'utilisation de la fonction `ZwSetSystemInformation` [Neb00, p.434]. L'extrait suivant du fichier journal illustre l'utilisation de cette méthode par un rootkit bien connu (Migbot) :

```
[HOOK]    FindResourceA[IN]  lpType=MIGBOT
[HOOK]    FindResourceA[IN]  lpName=BINARY
[HOOK]    FindResourceA[OUT] HRSRC=00407048
[HOOK]    ZwCreateFile[IN]  ObjectName=\??\C:\MIGBOT.SYS
[HOOK]    ZwCreateFile[IN]  FileAttributes=00000000 |
[HOOK]    ZwCreateFile[OUT] NTSTATUS=00000025
[HOOK]    GetProcAddress[IN]  hModule=7C910000
[HOOK]    GetProcAddress[IN]  lpProcName=RtlInitUnicodeString
[HOOK]    GetProcAddress[OUT] ProcAddress=7C9112D6
[HOOK]    GetProcAddress[IN]  hModule=7C910000
[HOOK]    GetProcAddress[IN]  lpProcName=ZwSetSystemInformation
[HOOK]    GetProcAddress[OUT] ProcAddress=7C91E729
[HOOK]    ZwSetSystemInformation[IN]
          SystemInformationClass=SystemLoadAndCallImage(38)
[HOOK]    ZwSetSystemInformation[OUT] NTSTATUS=000000F0
```

Pour tester le moteur d'unpacking, je propose la démarche suivante :

- dans un premier temps, nous validons l'algorithme d'unpacking en utilisant un exécutable de test connu, ne comportant pas de code automodifiable. Nous vérifions que le moteur d'unpacking permet de recouvrer l'exécutable d'origine.
- ensuite, pour une population virale packée et bien identifiée, nous vérifions que le moteur d'unpacking permet de recouvrer l'exécutable d'origine. La vérification est effectuée en recoupant l'information donnée par les fichiers journaux de VxStripper, concernant les interactions du virus avec les API du système d'exploitation, avec l'information mise à disposition sur le site web de certains éditeurs de produits antivirus.

La mise en oeuvre de la seconde étape de ce plan de test est très longue, bien qu'indispensable pour démontrer l'efficacité en pratique du module d'unpacking de VxStripper. La première étape a été mise en oeuvre (en 2006 [Jos06b]) pour quelques packers :

| Packer | version | Mode | Durée de l'analyse (s) |
|---|---|---|---|
| Armadillo | 4.05 | GI | 300 |
| ASPack | 2.12 | - | 150 |
| ASProtect | 1.23 RC4 | GI | 200 |
| PECompact | 1.56 | I | 70 |
| PEtite | 2.3 | - | 90 |
| PolyCryptPE | 2.1 | I | 80 |
| Shrinker | 3.4 | GI | 100 |
| UPX | 1.24w | - | 50 |
| Vbox | 4.3 | I | 80 |
| WWPack32 | 1.20 | - | 60 |
| yC | 1.2 | I | 70 |

Ces packers ont été utilisés avec le même programme cible, un exécutable PE Win32 qui affiche une petite fenêtre. Nous indiquons, outre le nom et la version du packer utilisé, le mode requis et la durée de l'analyse en seconde. Le mode (I)nteractif est requis par exemple lorsque la cible est constituée de plusieurs composants binaires, ou lorsque la protection interagit avec l'utilisateur. Le mode (G)raphique est requis lorsque cette interaction est graphique.

Pour un packer comme yC (version 1.2), nous obtenons le résultat suivant pour l'analyse :

```
[INFO]   Initialisation du systeme...
P> upload test.exe
[INFO]   Envoi de la commande: DOWNLOAD_TARGET test.exe 7238 (30)
[INFO]   Status: OK
P> logapicall
[INFO]   Set log API Call on
P> logdisasm
[INFO]   Set log disasm on
P> unpack
+-----------------------------------------------------------------------------------------+
| Informations sur le processus                        test.exe                           |
| KPEB=8158C020     PID=000003DC      Page Directory:0B228000     PEB=7FFDA000            |
+-----------------------------------------------------------------------------------------+
[INFO]   Envoi de la commande: RESUME_EXECUTION (17)
[INFO]   Analyse de l'intervalle [00400000,00407000]
[INFO]   Recuperation d'une information sur les imports potentiels...
[INFO]   db_nb_elts=13259
[INFO]   _PEB_ADDR=7FFDA000
[INFO]   _MODULE_NAME = test.exe
[INFO]   _MODULE_NAME=C:\WINDOWS\system32\ntdll.dll
[INFO]   _MODULE_NAME=C:\WINDOWS\system32\kernel32.dll
[INFO]   Synchronisation des bases : reparation de s_api
[INFO]   Synchronisation des bases : completion de s_api
[INFO]   Synchronisation des bases : ajout de 9778 entrees
```

```
[INFO]   Sauvegarde de la base
[INFO]   Recuperation d'une information sur les imports potentiels...[OK]
[RAW]     Section yC
          Adresse 00405000
[INFO]   OEP candidat (n°1) = 00405060
[INFO]   OEP candidat (n°2) = 004050AE
[RAW]     Section CODE
          Adresse 00401000
[INFO]   OEP candidat (n°3) = 00401000
[INFO]   Fin du programme
```

L'examen de l'exécutable au moyen d'IDA Pro confirme la reconstruction de l'exécutable :

```
CODE:00401000                 public start
CODE:00401000 start           proc near
CODE:00401000                 push    30h
CODE:00401002                 pop     edx
CODE:00401003                 mov     edx, fs:[edx]
CODE:00401006                 cpuid
CODE:00401008                 push    0
CODE:0040100A                 push    offset aHiBox   ; "HiBox"
CODE:0040100F                 push    offset aHi      ; "Hi!"
CODE:00401014                 push    0
CODE:00401016                 call    sub_40109C
CODE:0040101B                 cpuid
CODE:0040101D                 push    0
CODE:0040101F                 call    sub_401168
CODE:00401024                 jmp     ds:dword_403128
CODE:00401024 start           endp
```

Le fichier log de VxStripper garde trace des interactions du programme avec les APIs du système d'exploitation invité :

```
[INFO]    pc=0040109C, cs_base=00000000, eip=0040109C, tb->size=00000006
[RAW]     Section CODE
[RAW_DBG] raw_addr=13D2069C, lpf=13D20000
[RAW]     Chgt de section
en memoire:
0x0040109c:  jmp    ds:0x403178
sur le mapping:
0x13d2069c:  add    BYTE PTR [eax],al
0x13d2069e:  add    BYTE PTR [eax],al
0x13d206a0:  add    BYTE PTR [eax],al
[APICALL] 77D5050B: Appel de la fonction 77D5050B C:\WINDOWS\system32\USER32.dll::MessageBoxA
[HOOK]    MessageBoxA[IN]  lpText=Hi!
[HOOK]    MessageBoxA[IN]  lpCaption=HiBox
```

Pour un packer comme AsProtect, une première exécution de VxStripper peut être effectuée sans logs, afin d'accélérer le processus d'analyse.

```
[RAW]     Section
          Adresse 00401000
[INFO]   OEP candidat (n°1) = 00401000
[RAW]     Section .data
          Adresse 00405000
[INFO]   OEP candidat (n°2) = 00405001
[INFO]   OEP candidat (n°3) = 004051A8
```

```
[RAW]      Section
           Adresse 00401000
[INFO]    OEP candidat (n°4) = 00401014
[INFO]    OEP candidat (n°5) = 00401000
```

Remarquons que certains noms de section sont vides et que le point d'entrée candidat est égal au point d'entrée de la protection. Une seconde exécution en positionnant un VBP au point d'entrée candidat (`vbp 00401000`) nous permet d'obtenir un exécutable déprotégé.
Nous procédons de même avec le packer Armadillo, pour obtenir les points d'entrée candidats suivants :

```
[INFO]    OEP candidat (n°1) = 004307C3
[INFO]    OEP candidat (n°2) = 004337BF
[INFO]    Status: OK\ü
[INFO]    OEP candidat (n°3) = 0041B529
[RAW]      Section CODE
           Adresse 00401000
[INFO]    OEP candidat (n°4) = 00401000
```

Nous obtenons également un exécutable déprotégé et une trace des interactions du programme avec les APIs du système d'exploitation invité.

Remarquons que Vxstripper peut être utilisé pour faciliter l'analyse de programmes protégés sans utilisation d'un packer. Considérons à titre d'exemple le programme `0x90.exe` qui a fait l'objet du challenge *Scan of the month* n°33 du projet Honeynet [Bru04]. Nous obtenons la trace suivante avec VxStripper (durée de l'analyse : 30s approximativement) :

```
[RAW]      Section DATA
           Adresse 00de2000
[INFO]    OEP candidat (n°1) = 00DE2000
[INFO]    OEP candidat (n°2) = 00DE5423
[INFO]    OEP candidat (n°3) = 00DE59A3
...
[INFO]    OEP candidat (n°177) = 00DEF94D
[INFO]    OEP candidat (n°178) = 00DEF94D
[RAW]      Section CODE
           Adresse 00de1000
[INFO]    OEP candidat (n°179) = 00DE1013
```

En examinant les logs, nous obtenons une information sur les interactions du programme avec les APIs du système d'exploitation invité.

```
[INFO]     pc=00DEF94D, cs_base=00000000, eip=00DEF94D, tb->size=00000005
en memoire:
0x00def94d:  call   0x7c812c8d
sur le mapping:
0x13f6f94d:  call   0x8f992c8d
[APICALL] 7C812C8D: Appel de la fonction 7C812C8D C:\WINDOWS\system32\kernel32.dll::GetCommandLineA
...
[INFO]     pc=00DEF94D, cs_base=00000000, eip=00DEF94D, tb->size=00000005
[RAW]      Section DATA
[RAW_DBG] raw_addr=13F6F94D, lpf=13F60000
[RAW]      Les deux Basic Blocks sont differents
en memoire:
0x00def94d:  call   0x77c1186a
sur le mapping:
0x13f6f94d:  call   0x8f992c8d
[APICALL] 77C1186A: Appel de la fonction 77C1186A C:\WINDOWS\system32\msvcrt.dll::printf
```

```
[HOOK]    printf[IN]  lpFileName=Please Authenticate!
[APICALL] 7C810F9F: Appel de la fonction 7C810F9F C:\WINDOWS\system32\kernel32.dll::WriteFile
[APICALL] 7C81CBED: Appel de la fonction 7C81CBED C:\WINDOWS\system32\kernel32.dll::WriteConsoleA
[HOOK]    WriteConsoleA[IN]  lpBuffer=Please Authenticate! [HOOK]    WriteConsoleA[IN]  nNumberOfCharsToWr
...
[INFO]    pc=00DE1013, cs_base=00000000, eip=00DE1013, tb->size=00000006
[RAW]     Section CODE
[RAW_DBG] raw_addr=13F61013, lpf=13F60000
[RAW]     Chgt de section
en memoire:
0x00de1013:  jmp    ds:0xe28064
sur le mapping:
0x13f61013:  jmp    ds:0xe28064
[APICALL] 7C81CAA2: Appel de la fonction 7C81CAA2 C:\WINDOWS\system32\kernel32.dll::ExitProcess
```

Les tâches suivantes de l'analyse permettant de relever ce défi posé aux spécialistes de la rétro-ingénierie peuvent donc se fonder sur la nouvelle image mémoire du processus. L'analyste est débarrassé du problème posé par les 175 couches de chiffrement.

Je conclue l'exposé de ces expérimentations avec la trace d'exécution suivante :

```
[INFO]   Initialisation du systeme...
P> upload test.exe
[INFO]   Envoi de la commande: DOWNLOAD_TARGET test.exe 155736 (32)
[INFO]   Status: OK
P> setmax 1000000
[INFO]   Virtual Mem Max Boundary = 01000000
P> unpack
+-------------------------------------------------------------------------------------+
| Informations sur le processus                        test.exe                       |
| KPEB=8158C020     PID=000003DC      Page Directory:0B1FC000     PEB=7FFDD000        |
+-------------------------------------------------------------------------------------+
[INFO]   Envoi de la commande: RESUME_EXECUTION (17)
[INFO]    Analyse de l'intervalle [00400000,01000000]
[INFO]   Recuperation d'une information sur les imports potentiels...
...
[INFO]   Recuperation d'une information sur les imports potentiels...[OK]
[RAW]     Section .text
          Adresse 00401000
[INFO]   OEP candidat (n°1) = 004022D0
[INFO]   OEP candidat (n°2) = 004014EB
[RAW]     Section
          Adresse 00400000
[INFO]   OEP candidat (n°3) = 00990000
[INFO]   OEP candidat (n°4) = 0099002C
...
[INFO]   OEP candidat (n°102) = 00991166
[INFO]   OEP candidat (n°103) = 00991193
[INFO]   Status: OK
[INFO]   Envoi de la commande: QUIT (5)
[INFO]   Fin du programme
```

Nous avons étendu la zone VMEM sous contrôle (`setmax 0x01000000`). Avec ce paramétrage, nous pouvons analyser une section de code protégée par `N=100` couches de chiffrement exécutées depuis un tampon alloué dynamiquement en mémoire. En examinant les logs, nous obtenons une information sur les interactions de cette section de code avec les APIs du système d'exploitation invité.

```
[INFO]    pc=00991193, cs_base=00000000, eip=00991193, tb->size=0000001F
en memoire:
0x00991193:  popa
0x00991194:  mov    DWORD PTR [ebp-4],0x420144
0x0099119b:  mov    DWORD PTR [ebp-8],0x42011c
0x009911a2:  push   0x30
0x009911a4:  push   DWORD PTR [ebp-4]
0x009911a7:  push   DWORD PTR [ebp-8]
0x009911aa:  push   0x0
0x009911ac:  call   ds:0x42f28c
[APICALL] 77D5050B: Appel de la fonction 77D5050B C:\WINDOWS\system32\USER32.dll::MessageBoxA
[HOOK]    MessageBoxA[IN]  lpText=Hi!
[HOOK]    MessageBoxA[IN]  lpCaption=HiBox
```

Nous voyons donc que le module d'unpacking de l'outil VxStripper peut également être utilisé pour l'analyse de portions de code comportant du code auto-modifiable, indépendamment de l'utilisation d'un packer d'exécutable. Dans ce dernier exemple, la protection de la portion de code est effectuée au niveau du code source. La portion de code critique peut être par exemple une fonction d'authentification, de contrôle d'intégrité ou de déchiffrement interagissant avec l'utilisateur ou avec un composant matériel de sécurité.

## 5.4 Conclusion

Nous avons présenté dans ce chapitre les spécifications de l'outil VxStripper, dédié à l'analyse de code viral, fondé sur un CPU logiciel. Cet outil est adapté à l'analyse automatique de programmes aussi dangereux que les programme viraux. Il permet une instrumentation furtive des API natives et Win32 et propose un algorithme d'unpacking permettant de récupérer le corps d'un virus protégé et de reconstruire l'exécutable.

Concernant les limites de cet outil, nous avons constaté que les deux étapes de l'analyse (unpacking puis reconstruction de l'exécutable) peuvent échouer pour plusieurs raisons :

- la détection de l'émulation (en raison par exemple d'une imperfection de l'émulation d'un des composants matériel, du CPU notamment) ;
- l'utilisation de conditions de déclenchement environnementales (présence d'un fichier sur le système ou sur un autre ordinateur du réseau, intervalle horaire, ancrage à un matériel spécifique, etc.). Les conditions environnementales doivent être réunies pour que l'algorithme de dé-protection automatique de virus packés soit efficace ;
- l'utilisation d'un mécanisme d'altération de la section code de l'exécutable (le lien entre le loader de protection et l'exécutable protégé est alors suffisamment fort pour rendre hasardeuse la suppression automatique du loader de protection) ;
- l'utilisation d'un mécanisme d'émulation totale des imports : des heuristiques spécifiques doivent être mises en oeuvre pour réparer l'exécutable. Le fait de pouvoir tracer de manière presque omnisciente les interactions avec les API du système n'apporte pas de solution.

L'algorithme d'unpacking est conçu pour déprotéger automatiquement un programme packé. Nous supposons que le programme d'origine ne comporte pas de code auto-modifiable. Dans le cas contraire, il est nécessaire de mener une analyse manuelle sur la base des informations collectées pour retrouver le point d'entrée original. Je pense qu'il serait utile de proposer des fonctionnalités d'aide à l'analyse de code auto-modifiable, permettant de simplifier dans ce contexte l'étude du packer et du code viral protégé.
J'envisage comme tâche future l'implémentation d'une stratégie de réécriture de code (approche structurelle ou morphologique [CJS+05, DR05, BMM06, BKM08]) fondée sur le modèle de flot de contrôle SE-CFG [AMD06] (voir section 3.3.3 du chapitre 3), permettant :

- d'identifier un packer (à la manière de PEiD [PEi07]) ;
- de normaliser un code viral comportant du code automodifiable.

Pour y parvenir, on pourra :

- s'appuyer sur une représentation sous forme de terme [BKM08] du graphe SE-CFG ;
- définir une stratégie de réécriture de terme adéquate ;
- définir un schéma de détection fondé sur la recherche d'isomorphisme de graphe (par application d'heuristiques appropriées comme celles de l'analyse différentielle de binaires [DR05]) ou la recherche de sous-graphe (en s'appuyant sur la représentation sous forme de terme du graphe SE-ICFG).

Une représentation graphique du modèle de flot de contrôle SE-CFG avant et après normalisation pourra servir de support à l'analyse manuelle du packer, du code viral ou de tout autre programme comportant du code automodifiable.

Les fonctionnalités de l'outil VxStripper que nous avons présentées dans ce chapitre permettent de lever les premières protections d'un programme viral et de récupérer une information sur ses interactions avec les interfaces du système. Cette information doit être complétée par des données d'intégrité et de cohérence concernant le système lui-même pour l'analyse des virus furtifs. Je présente au chapitre suivant les spécifications du module spécialisé dans l'analyse des programmes furtifs de l'outil VxStripper.

Chapitre 6

# Analyse des virus furtifs

Nous avons présenté au chapitre précédent les bases du fonctionnement de VxStripper et son module d'unpacking. Nous complétons l'exposé des fonctionnalités de VxStripper par la présentation de son module d'analyse des programmes furtifs.

Ce module est spécialement conçu pour un analyste de sécurité pour l'analyse des rootkits et de tout type de programme qui interagit à un bas niveau avec le système d'exploitation, comprenant les antivirus, les pare-feux personnels et les programmes HIPS (*Host-Based Intrusion Prevention System*). Il met en oeuvre un système fiable pour récupérer automatiquement de l'information sur un programme furtif et ses interactions avec l'exécutif de l'OS, ainsi que des techniques forensiques pour surveiller les composants critiques du système.

Je propose dans ce chapitre une revue des méthodes de détection de rootkits, illustrée par l'étude de leur transposition dans l'outil VxStripper. J'utilise le nouveau point de vue fourni par cet outil pour poser un regard nouveau sur les heuristiques et techniques forensiques qui sont actuellement utilisées par les détecteurs de rootkits.
Parmi les méthodes de détection de rootkit, les méthodes de contrôle de modèle sont sans doute les plus génériques. J'étudie leur transposition dans l'outil VxStripper et propose une démarche générale, fondée sur la randomisation des contrôle et leur itération, pour améliorer la précision de l'estimateur dans le contexte d'une analyse passive.

## 6.1 Introduction

Zuo & Zhou [ZZ04] ont proposé une définition formelle de la notion de virus furtif par l'utilisation des fonctions récursives de la théorie de la complexité (voir section 1.2.1 du chapitre 1). En utilisant ce formalisme, un virus furtif est défini comme un virus qui modifie son environnement d'exécution de telle sorte qu'un programme contrôlant le système par le biais d'appels systèmes ne détectera pas la présence du virus. Concernant la détection des virus furtifs, ils prouvent que le problème général de la détection des virus furtifs est $\Sigma_3$ complet (ce problème est $\Pi_2$ complet pour un noyau viral fixé). Par conséquent, nous pouvons voir que la complexité du problème sous-jacent est trop élevée pour espérer trouver un algorithme générique qui permette de reconnaître ce type d'infection informatique.

E. Filiol a proposé une autre formalisation de la notion de furtivité [Fil07a], fondée sur la théorie de l'information. En exploitant l'analogie existante entre un système furtif et un système stéganographique, il propose des critères permettant de qualifier en terme de sécurité un système de furtivité. Nous présentons ce formalisme à la section 6.3.11. Il propose par ailleurs une nouvelle définition de la furtivité, vue comme une simulation des tests statistiques implémentés par un moteur de détection virale. La notion de simulabilité des tests statistique est présentée au chapitre 7.
Cet effort de modélisation permet de capturer des aspects liés aux interactions d'un programme furtif avec son environnement d'exécution. Nous allons voir qu'il permet de donner une base théorique aux méthodes de détection de rootkits fondées sur le contrôle de modèle.

Un rootkit peut être défini comme un programme qui implémente un ensemble de techniques de furtivité [Hog05]. Par conséquent, nous pouvons comprendre la notion de rootkit comme une technologie. Un rootkit est aussi communément associé avec une classe spécifique de modules noyau, qui s'exécutent au niveau de sécurité le plus privilégié du CPU.

Ils y a de nombreux endroits où un programme peut s'installer et se cacher au sein d'un système d'exploitation standard comme Windows NT, Mac OS ou Linux. Avec l'apparition des rootkits, la lutte du glaive contre le bouclier a un nouveau théâtre d'opérations incluant notamment : l'espace noyau, les registres du CPU, le BIOS (Basic I/O System). En examinant le flot d'exécution, partant d'une requête d'entrée/sortie jusqu'à l'opération matérielle effective, nous pouvons remarquer qu'il y a de nombreuses opportunités pour corrompre et se cacher. Nous pouvons également observer que de nombreux moyens ingénieux (ou non documentés) pour insérer un module noyau dans le système ont toujours été trouvé par les concepteurs de rootkits.

Donc le principal problème concernant la détection des rootkits est l'immense périmètre de ce large champ de bataille : en fait, le système d'exploitation tout entier, en incluant le BIOS, le registre, le système de fichier, les secteurs de démarrage, les registres du CPU, etc. En outre, la technologie rootkit évolue très rapidement, avec des rootkits qui exploitent, pour se cacher, la complexité structurelle croissante des systèmes d'exploitation et les nouvelles fonctionnalités des composants matériels. Par exemple, Blue Pill [Rut06] et Vitriol [Zov06] sont des rootkits preuve de concept qui implémentent une VMM légère sous l'OS hôte en utilisant les extensions de virtualisation AMD Pacifica [Zei05] ou Intel VT [UNR+05] du jeu d'instructions CPU. Dans les deux cas, la VMM du rootkit est installée alors qu'il s'exécute en ring 0, puis le système d'exploitation en cours d'exécution est migré dans une VM.

Le second problème est qu'un moteur de détection de rootkit doit faire son diagnostic sur un système qui peut être déjà corrompu, ou qui peut être corrompu durant l'analyse. Le diagnostic de sécurité s'en trouve donc altéré. Un système d'exploitation qui fonde sa sécurité sur la franche séparation entre deux niveaux de privilèges est un espace ouvert lorsque l'infection informatique et le détecteur s'exécutent dans le même domaine de privilège.

Les concepteurs de produits antivirus, pare-feux personnels et HIPS (*Host Based Protection Systems*) font face à plusieurs difficultés lorsqu'il s'agit de rootkits. Un programme antivirus se heurte aux mêmes contraintes qu'un rootkit lorsqu'il s'exécute sur un système d'exploitation : il doit être installé aussi profondément que possible au sein du système d'exploitation, et son installation doit être robuste vis-à-vis des attaques virales bas-niveau. En particulier, il doit être furtif. Pour être en mesure de récupérer de l'information de manière proactive, il doit corrompre plusieurs des composants du système d'exploitation. Cette contrainte induit des problèmes de stabilité. Dans cette lutte du glaive contre le bouclier, les deux parties utilisent les mêmes armes.

Les analystes humains ne souffrent pas des mêmes contraintes lorsqu'ils étudient le comportement d'un tel programme. Ils peuvent exécuter le programme dans un environnement émulé et récupérer l'information nécessaire pour contrôler les aspects liés à la sécurité d'un programme cible. Ils peuvent diriger un certain nombre de transformations du programme cible, comme le dépacketage, de manière à obtenir une version déprotégée du programme, qui peut ensuite être analysée statiquement. L'analyse statique du programme cible peut mettre en oeuvre des outils et algorithmes puissants, sans limitation concernant l'utilisation des ressources, de manière à obtenir une version désobfusquée d'un programme. Des débogueurs spécialisés peuvent être utilisés pour mener une analyse dynamique (et examiner les objets internes du système). Plusieurs produits anti-rootkits et outils forensiques peuvent être utilisés successivement pour affiner le diagnostic. L'analyse toute entière peut être conduite au sein d'un environnement virtuel qui peut se révéler très difficile à distinguer d'un environnement réel.

Ces degrés de libertés sont beaucoup moins réels pour des produits de sécurité proactifs : pour des raisons de performance, les moteurs d'émulation sont nécessairement plus médiocres. Par conséquent, les rootkits peuvent implémenter des routines de détection qui interdisent tout processus d'émulation efficace.

L'analyse n'est pas conduite par un humain et doit donc faire face à des difficultés qui peuvent être évitées durant un processus d'analyse conduit interactivement par un humain lors du désassemblage et de l'analyse dynamique d'un programme. L'objectif de ce chapitre est de présenter les spécifications générales d'un outil sécurisé et avancé pour

l'analyse des rootkits, fondé sur un CPU virtuel. Cette fonction anti-rootkit ne souffre pas des mêmes limitations que celles décrites ci-dessus. La conception de cet outil d'analyse est orientée isolation et furtivité.

La principale idée et choix de conception de cet outil d'analyse est d'appliquer des méthodes forensiques depuis l'extérieur du PC virtuel. Nous appelons *Matrice* le PC virtuel et son système d'exploitation, dans lequel le rootkit va évoluer. Tel un acuponcteur, nous appliquons des sondes précises et minutieuses de manière à effectuer des mesures furtives et de parvenir à un diagnostic.
Nous présentons ensuite un outil sécurisé, spécialement conçu pour un analyste pour l'étude des virus furtifs et de tout type de programmes qui interagissent à un bas niveau avec le système d'exploitation, ce qui inclut les produits anti-virus, les pare-feux personnels et les systèmes de détection d'intrusion basés sur l'hôte. Nous présentons un état de l'art des algorithmes de détection des rootkits, ainsi que des techniques forensiques et des heuristiques avancées pour la surveillance des composants critiques du système. L'outil d'analyse que j'ai développé propose un système fiable pour récupérer automatiquement de l'information sur un virus furtif et ses interactions avec l'exécutif du système d'exploitation. J'utilise le point de vue fourni pas cet outil pour apporter un nouveau regard aux heuristiques et techniques forensiques utilisées actuellement par les modules spécialisés dans la détection des virus furtifs.

L'objectif est de :
- pouvoir récupérer des informations d'interaction bas-niveau d'un programme avec le système d'exploitation ;
- mieux comprendre le fonctionnement des virus furtifs et être en mesure de mieux les détecter ;
- mieux comprendre le fonctionnement des moteurs de détection spécialisés.

Est-il possible de définir un environnement permettant l'analyse d'un virus furtif ou d'un moteur de détection spécialisé ? Le cas échéant, comment organiser la surveillance des objets du système de manière à ne pas être contourné ?
- Point théorique : le problème général de la détection des virus furtifs est $\Sigma_3$ complet en pire cas (voir )
- Spécification d'une solution approchée : contrôle d'intégrité, contrôle de cohérence (*Cross-view checking*), détection des virus furtifs par émulation.

Nous avons mis en oeuvre une technique de récupération d'informations d'intégrité et de cohérence de certaines parties du système d'exploitation par utilisation d'un émulateur de la plate-forme matérielle supportant le système d'exploitation. Nous proposons également une méthode statistique de détection des hooks.

Une partie du contenu de ce chapitre a été publiée dans les actes de la conférence EICAR 2007 [Jos07a].

Le reste de ce chapitre est structuré comme suit :
- la section 6.2 présente les autres travaux dans le domaine de la détection de rootkits ;
- la section 6.3 discute des choix de conception et des détails d'implémentation de notre solution. Cette section fournit également une évaluation expérimentale de son efficacité, au travers d'exemples d'utilisation.
- la section 6.4 discute de la contribution de ce travail, au regard des autres solutions existantes, puis donne les limitations et conditions d'utilisation.
- la section 6.5 conclut et donne un aperçu de mes travaux futurs.

## 6.2 Autres travaux

De nombreux algorithmes et heuristiques ont été récemment proposés par la recherche antivirale. Parmi ces méthodes, nous trouvons les méthodes de détection visant à une identification exacte fondées sur l'analyse de forme, les méthodes génériques statistiques (appliquées depuis l'intérieur de la matrice), et plusieurs algorithmes qui tentent de révéler une information sur la manière dont un rootkit installe et maintient des hooks dans le système cible.

*Méthodes de contrôle d'intégrité* (comparaison entre deux instantanés des composants critiques d'un système). Un nombre très important de travaux existent dans ce domaine. Parmi ceux-ci, pour ne citer que les plus récents, on trouve : WinPE GhostBuster [WBV+05], SVV (System Virginity Verifier) [Rut05c], CoPilot [PJFMA04] et Microsoft KPP [Mic06b], connu aussi sous le nom de Patch Guard.

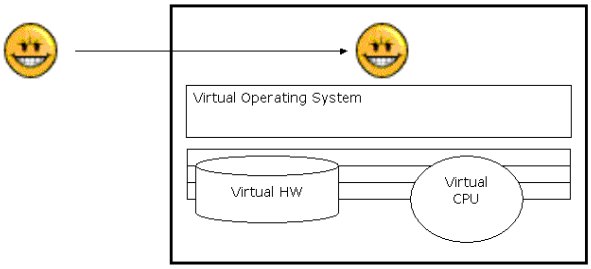



FIG. 6.1 – Rootkit from outside the Matrix

*Méthodes de contrôle de modèle* consistent à faire la comparaison entre deux valeurs d'un paramètre qui est représentatif du modèle statistique du système sous-jacent. La distribution d'un système sain et la distribution d'un système corrompu sont discriminées par utilisation d'un tel estimateur. Les méthodes de contrôle de modèle montre des résultats prometteurs. Une approche statistique, inspirée par la stéganographie, fondée sur la théorie de l'information, a été proposée dans [Fil07c, Fil07a]. Une telle approche est actuellement implémentée dans PatchFinder [Rut05a]. Le principe de cet outil de détection de rootkits est de détecter les hooks d'API en comparant le nombre d'instructions exécutées durant l'invocation de plusieurs fonctions d'API avec une référence saine (sous l'hypothèse que les hooks induisent l'exécution d'instructions supplémentaires qui ne seraient pas exécutées par une fonction non hookée). C. Lauradoux a proposé plus récemment [Lau08] une méthode de détection des rootkits se fondant sur l'utilisation d'une machine virtuelle. Le principe de cette technique de détection consiste à analyser les canaux cachés induits par la présence d'un hyperviseur sur le système (analyse des temps d'exécution de l'ordonnanceur ou des manipulation du cache). Nous reviendons plus en détail sur ces deux techniques dans la section 6.3.11.

*Méthodes de contrôle de cohérence* consistent à contrôler la cohérence de plusieurs objets, étant donnée une information sur les règles intrinsèques auxquelles ils doivent se conformer. Il existe plusieurs outils implémentant ces méthodes, parmi lesquels nous pouvons citer SIG∧2 KprocCheck [SIG06] et VICE [BH04]. Ces deux outils cherchent parmi les entrées de la table des services, celles qui contiennent des pointeurs de fonctions qui pointent en dehors de l'image mémoire de `ntoskrnl.exe` (pour SSDT) ou `win32k.sys` (pour Shadow SSDT).

*Méthodes de détection fondées sur la comparaison croisée* consistent à faire une comparaison entre deux points de vue pour plusieurs objets du système : par exemple, une vue haut niveau de ces objets et une vue bas niveau de ces mêmes objets. L'outil RootkitRevealer [CR06] de Microsoft SysInternals détecte les entrées du registre cachées et les fichiers cachés en analysant les ruches du registre et le système de fichier à un très bas niveau et ce, sans l'aide des API Win32, et compare le résultat avec celui obtenu au travers d'appels aux API Win32 standards. F-Secure BlackLight [FS06] et IceSword [Ice06] utilisent tous les deux des méthodes fondées sur la comparaison croisée pour détecter les processus et les fichiers qui sont cachés pour l'utilisateur et les logiciels de sécurité.

*Méthodes de détection fondées sur des signatures* consistent à comparer un motif donné à une base de signatures, de manière à identifier un certain rootkit. Dans le cas d'infection informatiques furtives, cette approche est limitée. Nous avons à espérer qu'un rootkit laisse suffisamment de trace pour permettre son identification. RAIDE [But06] utilise une méthode de recherche de signatures en mémoire pour trouver les blocs `EPROCESS` cachés par les rootkits. SIG∧2 APIHookCheck détecte la présence des hooks d'API *system-wide* qui sont implémentés en insérant des instruction de sauts (`jmp`) au début de l'API réelle.

Nous pouvons donc observer que de nombreuses méthodes sont disponibles pour détecter l'utilisation de la technologie rootkit. De nouvelles méthodes d'analyse croisée, d'analyse de forme ou de comportement sont régulièrement proposées par la recherche anti-virale. Leur efficacité est cependant un vrai défi, tout particulièrement lorsqu'un rootkit a déjà pris place dans le système hôte. C'est un fait, les outils de détection de rootkit, lorsqu'ils s'exécutent à l'intérieur de la matrice, sont faciles à détecter et éventuellement à contourner. L'exécution au sein d'un environnement contrôlé, comme une machine virtuelle, peut résoudre les problèmes d'isolation et de furtivité.

## 6.3 Conception de notre solution

Nous allons à présent présenter une approche fondée sur l'émulation pour l'analyse sécurisée et fiable des rootkits. Notre outil est conçu pour un analyste de sécurité. Il couvre au moins les domaines d'utilisation suivants :
- analyse des rootkit,
- test des produits antivirus, pare-feux personnels et HIPS.

Nous utilisons le nouveau point de vue fourni par l'outil VxStripper (i.e. nous récupérons de l'information sur le rootkit en observant le système corrompu dans son intégralité depuis l'extérieur) pour porter un regard neuf aux heuristiques et algorithmes qui sont actuellement utilisés par les détecteurs de rootkits.

### 6.3.1 Moteur d'émulation

Nous avons adapté l'émulateur QEMU [Bel05] de manière à implémenter le moteur d'émulation qui est au coeur de l'outil VxStripper.La conception générale du logiciel est très comparable à TTAnalyze [BKK06], un outil dédié à l'analyse des infections informatiques.

La machine virtuelle embarque un service réseau qui communique par le biais d'une interface réseau virtuelle avec le moniteur de machine virtuelle. Ce canal de communication est utilisé pour charger les binaires cibles dans la machine virtuelle, démarrer l'exécution du programme cible principal et récupérer l'information du noyau nécessaire pour pouvoir piloter l'exécution du processus invité depuis le système hôte. Nous pouvons remarquer que cette information, qui est localisée dans les tables et structures de l'exécutif NT, peut être obtenue depuis le mode noyau (le moyen de l'obtenir depuis le mode noyau est bien documenté dans [SR00, RS05]) ou par des méthodes forensiques (et furtives).

VxStripper récupère de l'information sur :
- les appels aux API Win32 et native du programme cible,
- le code assembleur du programme cible, désassemblé séquentiellement,
- la structure de l'exécutable dans la mémoire du système invité (et une comparaison dynamique avec le fichier brut sur le système de fichier),
- plusieurs structures et objets internes du système d'exploitation : descripteur d'interruption, regsitre `SYSENTER_EIP`, descripteurs deservice (shadow) (system), liste doublement chaînée des processus, objets pilotes, tables des paquets de requête d'entrée/sortie, etc.

### 6.3.2 Mode d'analyse

Le programme principal peut être utilisé dans deux modes consoles :
- le mode par défaut charge automatiquement l'exécutable cible dans la VM, supprime le chargeur de protection éventuel et récupère l'information relative à ses interactions avec le système d'exploitation invité ;
- le mode interactif permet à un analyste de sécurité de diriger dynamiquement l'exécution du programme cible et d'interagir avec la VM, en contrôlant ses états.

Le programme principal peut également être utilisé en mode graphique. Ce mode est utile lorsque le processus d'analyse nécessite une interaction entre l'utilisateur et le programme cible par le biais d'une interface graphique. Les mêmes options qu'en mode console (mode graphique par défaut ou interactif) sont utilisables.

### 6.3.3 Tracer les appels aux API

Parmi les informations qui peuvent être extraites d'un exécutable, la plus importante est sans doute la nature de ses interactions avec le système d'exploitation. Donc nous traçons à la fois les appels aux API Win32 et native[1] Cette première information est suffisante pour comprendre comment une infection informatique furtive insère son pilote dans le noyau.

Donnons un exemple. Nous chargeons et démarrons le rootkit `migbot` [Hog05] (à l'intérieur de l'émulateur) :

```
P> upload migbot.exe
[INFO] download target migbot.exe 27136
[INFO] status: OK
P> execsuspend
P> resumeexec
P> disconnect
```

Nous examinons alors le fichier journal :

```
[APICALL] 7C80C7B1: call function 7C80C7B1
                    C:\WINDOWS\system32\kernel32.dll::FindResourceA
[HOOK]    FindResourceA[IN]  lpType=MIGBOT
[HOOK]    FindResourceA[IN]  lpName=BINARY
[HOOK]    FindResourceA[OUT] HRSRC=00407048
                             ...
[APICALL] 7C91D682: call function 7C91D682
                        C:\WINDOWS\system32\ntdll.dll::ZwCreateFile
[HOOK]    ZwCreateFile[IN]  ObjectName=\??\C:\MIGBOT.SYS
[HOOK]    ZwCreateFile[IN]  FileAttributes=00000000 |
[HOOK]    ZwCreateFile[OUT] NTSTATUS=00000025
                             ...
[APICALL] 7C80AC28: call function 7C80AC28
                   C:\WINDOWS\system32\kernel32.dll::GetProcAddress
[HOOK]    GetProcAddress[IN]  hModule=7C910000
[HOOK]    GetProcAddress[IN]  lpProcName=ZwSetSystemInformation
[HOOK]    GetProcAddress[OUT] ProcAddress=7C91E729
                             ...
[APICALL] 7C91E729: call function 7C91E729
              C:\WINDOWS\system32\ntdll.dll::ZwSetSystemInformation
[HOOK]    ZwSetSystemInformation[IN]
                 SystemInformationClass=SystemLoadAndCallImage(38)
[HOOK]    ZwSetSystemInformation[OUT] NTSTATUS=000000F0
```

Nous obtenons une information sur la manière qu'utilise ce rootkit pour insérer son module noyau, d'abord en localisant son corps (une ressource binaire), puis en écrivant ce binaire sur le disque avant d'installer puis démarrer le module au sein du noyau en utilisant l'appel système `ZwSetSystemInformation`.

En utilisant cette fonction de trace, nous sommes en mesure de faire un diagnostic sur la méthode utilisée par un rootkit pour insérer son module noyau, ce quelque soit la méthode utilisée (via SCM avec `OpenSCManager`, `CreateService`, `OpenService` ou via NtDLL avec `ZwLoadDriver`, `ZwSystemDebugControl`, `ZwSetSystemInformation`, ou par toute autre méthode non documentée).

Si le dropper est packé, nous devons d'abord le dépacker en utilisant la procédure décrite dans [Jos06b].

### 6.3.4 Objets sous surveillance

Le second ensemble d'information qui est crucial lorsque l'on s'intéresse aux rootkits est une vision de tous les endroits au sein de la plate-forme hôte où un hook peut être installé. Une première étape dans notre méthode

[1] L'API native de Windows NT est l'ensemble des services systèmes fournis par l'exécutif de Windows NT, à la fois aux programmes en mode utilisateur et en mode noyau.

d'analyse consiste à naviguer à travers les structures exécutives du système d'exploitation de façon à identifier les cibles potentielles d'une attaque virale utilisant la technologie rootkit.

Nous avons également besoin d'identifier et de surveiller tous les composants matériels susceptibles d'être corrompus par un rootkit : mémoire flash du BIOS, registres CPU, secteurs de boot, etc.

Une fois cette information collectée, nous pouvons mettre en oeuvre des algorithmes pour aider l'analyste de sécurité à faire son diagnostic, en commençant par les heuristiques les plus couramment utilisées.

Nous appliquons des contrôles d'intégrité forensiques à différentes étapes du cycle de vie d'un rootkit cible. Nous prenons un instantané des structures et objets cruciaux et surveillons toute modification. Le principe consiste à énumérer les objets cibles en utilisant des méthodes forensiques, puis à utiliser une fonction de hachage ou une comparaison champs à champs des structures.

Les objets suivants, tables et structures sont actuellement sous surveillance :
- BIOS, MBR (*Master Boot Record*), IVT (*Interrupt Vector Table*),
- sections code du processus en mémoire, tables IAT/EAT,
- objets processus et thread, SDT (*Service Dispatch Table*),
- code du noyau, SSDT (*System Service Dispatch Table*) et IDT (*Interrupt Descriptors Table*),
- objets Driver, tables IRP_MJ,
- registres CPU (`SYSENTER_EIP`, `LDT`, `GDT`, `CR0`, `CR3`, `EFLAGS`).

Notre outil est conçu pour exécuter ces contrôles à chaque cycle du CPU virtuel. Par conséquent, il est *a priori* immune aux attaques de type *timing attacks*. Une telle attaque est faisable lorsqu'on utilise une solution matérielle comme Copilot [PJFMA04] car une telle solution est conçue pour effectuer ses contrôles périodiquement (toutes les 30s dans le présent prototype).

Notre outil est également immune aux situations de concurrence (*race conditions*). Nous accédons à la mémoire physique et virtuelle directement depuis l'extérieur, sans aucune interaction avec le matériel virtuel. Il est possible de mener une telle attaque lorsqu'on utilise une solution matérielle comme CoPilot car le moniteur de Copilot n'accède à la mémoire de l'hôte qu'au travers du bus PCI.

Notre outil est immune aux attaque portant sur le cache CPU. À chaque cycle du CPU virtuel, nous pouvons accéder le cache du processeur. Parce qu'il est possible de maintenir une vue consistante de la mémoire surveillée par Copilot tout en cachant du code malicieux ailleurs, comme par exemple dans le cache du processeur, ce code restera non détecté par Copilot.

### 6.3.5 Contrôle d'intégrité des modules

L'intégrité de tous les modules chargés (incluant `NtOsKrnl` et `Hal`) est vérifiée à la demande où à chaque cycle du CPU. Si une corruption est détectée, l'intégrité de la section code, de l'IAT et de l'EAT[2] du module cible sont contrôlés. Donnons un exemple : nous installons un hook inline dans un exécutable cible (à l'intérieur de l'émulateur) :

```
C:\>withdll /d:traceapi.dll parano.exe
```

À présent, examinons le fichier journal :

```
[DETECT] inline hook detected
```

Nous remarquons qu'une alerte a été levée car une des fonctions importées par l'exécutable cible a été modifiée. Nous surveillons la section code, l'EAT et l'IAT de tous les modules en mémoire pour détecter toute tentative faite par un rootkit d'installer un hook.

[2] L'IAT (*Import Address Table*) est utilisée lorsqu'une application utilise une fonction d'API pour importer son adresse. L' EAT (*Export Address Table*) est utilisée par une DLL pour exporter ses fonctions d'API.

### 6.3.6 IDT et registre Sysenter

L'IDT (*Interrupt Descriptor Table*) est utilisée pour trouver les gestionnaires d'interruptions. Nous utilisons le registre du CPU virtuel pour localiser cette table en mémoire virtuelle (SegmentCache `IDTInfo=virtual_cpu->idt`). Nous chargeons ensuite le contenu de la table IDT.

Donnons un exemple : nous démarrons le rootkit strace (à l'intérieur de l'émulateur) :

```
C:\>pslist paranoiac
      Process        PID
      Paranoiac.exe  1436
C:\>strace 1436 10000
```

À présent, examinons le fichier journal :

```
[INFO]   IDT Base: 8003F400 Limit:000007FF
+-------+---------------------+---+---+
| IDT   |  Selector:Handler   |DPL| P |
+-------+---------------------+---+---+
|  00   |   0008:804dfbff     | 0 | 1 |
|  01   |   0008:804dfd7c     | 0 | 1 |
|  02   |   0058:0000112e     | 0 | 1 |
   ..          ...             .   .
|  2e   |   0008:804deea6     | 3 | 1 |
   ..          ...             .   .
|  fd   |   0008:804ded32     | 0 | 1 |
|  fe   |   0008:804ded39     | 0 | 1 |
|  ff   |   0008:804ded40     | 0 | 1 |
+-------+---------------------+---+---+

[INFO]   IDT Base: 8003F400 Limit:000007FF
+-------+---------------------+---+---+
| IDT   |  Selector:Handler   |DPL| P |
+-------+---------------------+---+---+
|  00   |   0008:804dfbff     | 0 | 1 |
|  01   |   0008:804dfd7c     | 0 | 1 |
|  02   |   0058:0000112e     | 0 | 1 |
   ..          ...             .   .
|  2e   |   0008:fa00f2a0     | 3 | 1 |
   ..          ...             .   .
|  fd   |   0008:804ded32     | 0 | 1 |
|  fe   |   0008:804ded39     | 0 | 1 |
|  ff   |   0008:804ded40     | 0 | 1 |
+-------+---------------------+---+---+
[DETECT] IDT hook detected
```

Nous pouvons remarquer qu'une alerte a été déclenchée car l'entrée `INT 2E` dans l'IDT a été modifiée. Tout programme qui modifie l'IDT sera détecté. En outre, nous obtenons une information sur la localisation de la routine du rootkit en mémoire.

Les instructions `INT 2E` et `SYSENTER` sont exécutées par `NtDLL.dll` pour passer la main au noyau. Pour pourvoir détecter une attaque rootkit portant sur `IA32_SYSENTER_EIP`, il suffit de surveiller le registre correspondant `virtual_cpu->sysenter_eip` du CPU virtuel.

Donnons un exemple : nous installons et démarrons le rootkit `sysenter` (à l'intérieur de l'émulateur) :

```
C:\>instdriver -Install sysenter sysenter.sys
C:\>instdriver -Start sysenter
```

À présent, examinons le fichier journal :

```
[INFO]   sysenter_eip=804def6f
[INFO]   sysenter_eip=fa02e49a
[DETECT] sysenter hook detected
```

Nous pouvons remarquer qu'une alerte a été levée car le registre SYSENTER_EIP a été modifié. Tout programme qui modifie le registre SYSENTER_EIP sera détecté. En outre, nous obtenons une information relative à la localisation de la routine du rootkit en mémoire.

Remarquons que cette méthode de contrôle d'intégrité ne nous met pas à l'abris des fausses alarmes, dans la mesure où des programmes non viraux utilisent également des hooks. Cette remarque s'applique aussi aux autres fonctions de contrôle des objets de l'exécutif du système présentées par la suite.

### 6.3.7 Objets processus et pilote

La liste doublement chaînée des structures processus est chargée depuis la mémoire virtuelle, en partant de la localisation du processus système initial. La structure toute entière est surveillée de manière à détecter toute tentative de supprimer un de ses éléments ou de modifier ses valeurs critiques (jeton de sécurité à des fins d'élévation de privilège, etc.).

Donnons un exemple : le rootkit Fu [fuz06] peut dissimuler des processus et des pilotes de périphériques. Il peut également élever le privilège d'un processus et son groupe. La méthode utilisée est appelée manipulation directe des objets du noyau, DKOM (*Direct Kernel Object Manipulation*). Fu est un jeu de mot sur le programme UNIX su, utilisé pour élever le privilège.

Nous démarrons le rootkit Fu (à l'intérieur de l'émulateur) :

```
C:\>pslist paranoiac
      Process        PID
      Paranoiac.exe  1436
C:\>fu -prs 1436 SeLoadDriverPrivilege
```

À présent, regardons le fichier journal :

```
[DETECT] EPROCESS corruption detected (privilege elevation)
```

Nous remarquons qu'une alerte a été déclenchée car la structure EPROCESS a été modifiée. Tout programme qui modifie la structure EPROCESS sera détecté.

Supprimons à présent un élément de la liste doublement chaînée des EPROCESS :

```
C:\>fu -ph 1436
```

À présent, examinons le fichier journal :

```
[INFO]   activeLink=815b3718, PID=0000059c, Name=paranoiac.exe
[DETECT] EPROCESS DKOM detected
```

Nous remarquons qu'une alerte a été levée car la liste doublement chaînée a été modifiée. Tout programme qui modifie la liste doublement chaînée EPROCESS sera détecté.

La liste doublement chaînée de structures module entry est chargée depuis la mémoire virtuelle, en partant de la valeur du symbole PsLoadedModuleList qui est exportée par NtOsKrnl. La structure toute entière est surveillée de manière à détecter toute tentative de supprimer un de ses éléments (dissimulation de module noyau) ou de modifier une de ses valeurs critiques.

Supprimons un élément de la liste doublement chaînée DRIVER_OBJECT :

```
C:\>fu -phd paranoiac.sys
```

À présent, examinons le fichier journal :

```
[INFO]   activeLink=815b2f98, imageBase=f8c95000, imageSize=00053000,
                        drvPath=\SystemRoot\system32\DRIVERS\srv.sys
         activeLink=816f1c50, imageBase=f8d95000, imageSize=0000f000,
  drvPath=\??\C:\Documents and Settings\Sébastien Josse\paranoiac.sys
         activeLink=8055ab20, imageBase=f8aec000, imageSize=00041000,
                       drvPath=\SystemRoot\System32\Drivers\HTTP.sys
[INFO]   activeLink=816f1c50, imageBase=f8c95000, imageSize=00053000,
                        drvPath=\SystemRoot\system32\DRIVERS\srv.sys
         activeLink=815716c0, imageBase=f8aec000, imageSize=00041000,
                       drvPath=\SystemRoot\System32\Drivers\HTTP.sys
[DETECT] DRIVER_OBJECT DKOM detected
```

Nous remarquons qu'une alerte a été levée car la liste doublement chaînée `DRIVER_OBJECT`a été modifiée. Tout programme qui modifie cette liste doublement chaînée sera détecté.

### 6.3.8 IRP Major Functions

Les IRPs (*I/O Request Packet*) sont gérés par l'exécutif NT et les pilotes pour communiquer des tampons de données ou des codes de contrôles à un programme en mode utilisateur. Nous chargeons la table des fonctions IRP Major de chaque pilote au sein de la mémoire virtuelle de l'exécutif du système d'exploitation invité pour détecter tout hook dans ces tables cruciales. De manière à accéder à ces objets pilotes en utilisant uniquement des méthodes forensiques, il est nécessaire de parcourir l'arbre des pilotes tout entier et d'explorer chaque noeud de cette structure en mémoire virtuelle.

Donnons un exemple : nous installons et démarrons le rootkit `irphook` (à l'intérieur de l'émulateur) :

```
C:\>instdriver -Install irphook irphook.sys
C:\>instdriver -Start irphook
```

Examinons à présent le fichier journal :

```
[DETECT] IRP_MJ hook detected
```

Nous pouvons observer qu'une alerte a été levée car la table des fonctions IRP Major au sein d'un objet `DRIVER_OBJECT` a été modifiée. Tout programme qui modifie cette table au sein d'un `DRIVER_OBJECT` sera détectée. En outre, nous obtenons une information relative à la localisation du gestionnaire du rootkit en mémoire.

### 6.3.9 Tables de dispatch des services

La SSDT (*System Service Dispatch Table*) est utilisée par l'exécutif NT pour gérer les appels systèmes. En chargeant le fichier PDB des symboles de `NtOSKrnl`, nous pouvons localiser la table de descripteurs de services en mémoire virtuelle. Nous chargeons la SSDT dans son intégralité et surveillons toute modification effectuée sur cette table. La même opération de contrôle est effectuée avec la table Shadow SSDT.

Donnons un exemple : nous installons et démarrons le rootkit `hideprocess` (à l'intérieur de l'émulateur) :

```
C:\>instdriver -Install hideprocess hideprocess.sys
C:\>instdriver -Start hideprocess
```

À présent, examinons le fichier journal :

```
[INFO]   _service_descriptor[0]=804e2d20
         _service_descriptor[1]=00000000
         _service_descriptor[2]=0000011c
         _service_descriptor[3]=804d8f48
         service_table[000]=80586691, argument_table=18
         service_table[001]=805706ef, argument_table=20
                                   ...
         service_table[173]=8057cc27, argument_table=10

         service_table[282]=80648e3c, argument_table=10
```

```
        service_table[283]=8062c033, argument_table=00
[INFO]
        service_table[000]=80586691, argument_table=18
        service_table[001]=805706ef, argument_table=20
                                        ...
        service_table[173]=fa044486, argument_table=10

        service_table[282]=80648e3c, argument_table=10
        service_table[283]=8062c033, argument_table=00
[DETECT] SSDT hook detected
```

Nous remarquons qu'une alerte a été déclenchée car une entrée dans la SSDT a été modifiée. Tout programme qui modifie la SSDT (ou SHADOW SSDT) sera détecté. En outre, nous obtenons une information relative à la localisation du gestionnaire du rootkit en mémoire.

### 6.3.10 Surveillance du processus de démarrage

Durant le processus de démarrage d'un PC, le BIOS transfère l'exécution à un code situé sur un autre média comme un périphérique disque, un CDROM ou via le réseau (l'agent BIOS PXE[3] effectue alors une requête et télécharge un fichier de démarrage depuis un serveur BOOTP/TFTP).

- Lorsque le processus de démarrage est effectué depuis un périphérique disque, le BIOS charge le premier secteur de démarrage du périphérique disque (MBR[4]) qui localise une partition de démarrage dans la table des partitions puis exécute le premier secteur de cette partition de démarrage. Le secteur de démarrage de la partition de démarrage charge et prend en charge l'étape suivante de démarrage de l'OS.
- Dans le cas où le processus de démarrage s'effectue depuis un CD-ROM, le BIOS charge le code de démarrage depuis le CD-ROM qui prend alors en charge l'étape suivante de démarrage de l'OS.
- Dans le cas d'un démarrage réseau, le BIOS exécute l'agent PXE, qui télécharge et exécute le fichier de démarrage réseau. Le fichier de démarrage réseau prend en charge l'étape suivante de démarrage de l'OS.

Le loader de démarrage de Windows charge et exécute `NtLdr`, qui créé les tables GDT et IDT puis met en place et exécute `OsLoader`. `OsLoader` charge l'OS (`NtDetect`, `Hal`, `NtOsKrnl`, `BootVid`, etc.).

Certains rootkits exploitent le fait que le démarrage de Windows repose sur le BIOS. Le démarrage de Windows utilise les interruptions BIOS, par conséquent la table IVT est généralement conservée. En outre, il doit respecter l'espace mémoire réservé par le BIOS. Les rootkits de BIOS comme eEye BootRoot [SP05] ou BootKit [roo07] exploitent cette confiance pour installer des hooks au niveau du BIOS.

Nous surveillons plusieurs endroits (zones mémoire réservées par le BIOS et le VGA BIOS, MBR, IVT en mode réel, etc.) de manière à détecter toute tentative de prendre le contrôle avant le loader de démarrage de Windows.

Donnons un exemple : *eEye's BootRoot* [SP05] est un rootkit preuve de concept présenté comme une exploration des technologies qu'un secteur de démarrage personnalisé peut utiliser pour corrompre le noyau Windows lors de son chargement. Il s'exécute après le BIOS mais avant le système d'exploitation, autorisant une prise de contrôle complète sur le système d'exploitation.

Nous installons et démarrons `BootRoot` depuis un périphérique disquette virtuel. Examinons à présent le fichier journal :

```
[INFO] IVT
+-----+---------------------+
| INT |    ADDR             |
+-----+---------------------+
| 00  |  f000ff53           |
| 01  |  f000ff53           |
  ..       ...
```

[3] *Preboot eXecution Environment*
[4] *Master Boot Record*

```
| 13  |  f000e3fe          |
  ..       ...
| fe  |  f000ff53          |
| ff  |  f000ff53          |
+-----+--------------------+
[INFO] IVT
+-----+--------------------+
| INT |    ADDR            |
+-----+--------------------+
| 00  |  f000ff53          |
| 01  |  f000ff53          |
  ..       ...
| 13  |  9f80005c          |
  ..       ...
| fe  |  f000ff53          |
| ff  |  f000ff53          |
+-----+--------------------+
[DETECT] IVT corruption detected
```

BootKit [roo07] est un projet relatif à l'utilisation d'un secteur de démarrage personnalisé pour corrompre le modèle de sécurité de Windows NT. Le prototype actuel essaye d'élever les privilèges de `cmd.exe` jusqu'au niveau système toutes les 30 secondes. Il réécrit le noyau au moment du démarrage. Bootkit est compatible PXE. Un résultat similaire au résultat précédent est obtenu en démarrant sur le CR-ROM virtuel BootKit :

```
[INFO] IVT
+-----+--------------------+
| INT |    ADDR            |
+-----+--------------------+
| 00  |  f000ff53          |
| 01  |  f000ff53          |
  ..       ...
| 13  |  9e000068          |
  ..       ...
| fe  |  f000ff53          |
| ff  |  f000ff53          |
+-----+--------------------+
[DETECT] IVT corruption detected
```

Ces deux rootkits hookent l'interruption `INT 13h` pour réécrire les fichiers de l'OS (`OsLoader`, etc.) lors de leur chargement. La fonction de surveillance du processus de démarrage est en développement. En traçant chaque instruction CPU (en partant de l'instant $t_0$ : la première instruction du BIOS est exécutée) et en comparant cette trace avec celle obtenue à partir d'un processus de démarrage sain de référence, nous sommes en mesure de détecter toute tentative d'exécuter du code supplémentaire après le BIOS et avant le système d'exploitation. En surveillant une classe plus large de matériels virtuels, nous pourrions être en mesure de détecter les virus de BIOS nouvelle génération [Fil05a] comme : BootRoot [SP05], les rootkits de BIOS ACPI[5] [Hea06b] et les rootkits PCI[6] [Hea06a].

### 6.3.11 Contrôle de modèle

Le contrôle de modèle consiste à faire la comparaison entre deux valeurs d'un paramètre qui est représentatif du modèle statistique du système sous-jacent. La distribution d'un système sain $DSys$ et la distribution d'un système corrompu $DStealth$ sont discriminées par utilisation d'un tel estimateur.

E. Filiol propose dans [Fil07c, Fil07a] une formalisation du problème de détection des programmes furtifs dans le cadre de la théorie de l'information. Il exploite le parallèle qui peut être fait entre les techniques de furtivités et les techniques stéganographiques pour définir la sécurité d'un système furtif contre une attaque passive :

[5] *Advanced Configuration and Power Interface*, successeur de APM (*Advanced Power Management*)
[6] *Peripheral Component Interconnected*

**Définition 6.3.1 (Sécurité d'un système furtif [Fil07a])** *Un système furtif est dit $\varepsilon$-sûr vis-à-vis des attaques passives si et seulement si :*

$$d_{KL}(p_{DSys}||p_{DStealth}) = \sum_y p_{DSys}(y) \log \frac{p_{DSys}(y)}{p_{DStealth}(y)} \leq \varepsilon.$$

Remarquons que l'utilisation de la distance de Kullback-Leibler entre les deux distributions permet de capturer toutes les dépendances entre les deux distributions, pas seulement par exemple une dépendance du second ordre, telle que capturée par la covariance. Elle définit l'entropie relative entre les deux distributions.
Lorsque $\varepsilon$ est nul, le système furtif est dit parfaitement (ou inconditionnellement) sûr. Cela implique que la détection du système furtif est impossible, i.e. que même avec des ressources illimitées en temps et en espace de calcul, il est impossible de distinguer les distributions $DStealth$ et $DSys$ et donc de détecter le système furtif. Il est peu probable que les méthodes classiques de furtivités puissent appartenir à cette classe de système, sans avoir recours à des mécanismes cryptographiques ou stéganographiques adaptés.
Lorsque l'adversaire est un algorithme déterministe s'exécutant en temps polynômial, le système furtif est dit non-sûr.

Remarquons que cette définition ne s'applique qu'au contexte d'attaques passives, c'est-à-dire au cas où le système d'extraction d'information et plus généralement l'ensemble des composants du moteur de détection ne modifient pas le système analysé (l'analyse ne doit en particulier pas modifier les paramètres caractérisant la distribution $DSys$). Si l'on considère les méthodes d'analyse actives (l'analyse modifie la distribution $DSys$ et éventuellement la distribution $DStealth$), nous devons modifier en conséquence la définition d'un système furtif (en remplaçant les mesures de probabilités $P_X$ par des mesures de probabilités $P_{X|Y}$, où $X$ désigne la v.a. régie par la loi mesurée sur le système et $Y$ désigne la v.a. correspondant aux modifications induites par la présence de l'outil d'analyse sur le système).

Les méthodes de contrôle de modèle telles que celles proposées dans PatchFinder [Rut05a] pour détecter les hooks d'API ou dans [Lau08] pour détecter la présence d'un hyperviseur de machine virtuelle s'appuient sur un moteur de détection actif. Ces méthodes de détection peuvent donc être contournées en manipulant l'environnement d'exécution du moteur de détection et en simulant les tests statistiques. Pour éviter cet ecueil, la fonction de détection doit s'exécuter depuis l'extérieur du système, de manière à avoir une distribution $DSys$ qui ne peut pas être simulée par le programme furtif, puisqu'il ne peut pas y accéder.

J'étudie dans la suite de cette section une approche générique pour spécifier une méthode de détection qui tente de décider si un système donné, observé depuis l'extérieur (par utilisation de méthodes forensiques) est corrompu par une infection informatique furtive. J'illustre cette approche par la transposition des techniques de détection décrites dans [Rut05a] et [Lau08] dans le contexte d'une analyse passive.

PatchFinder [Rut05a] tente de détecter les hooks d'API en comparant le nombre d'instructions exécutées durant l'invocation de plusieurs fonctions d'API avec une référence saine. L'estimateur choisi par ce test statistique de détection est donc le nombre moyen $m$ d'instructions exécutées durant l'invocation d'une fonction d'API donnée. Cet estimateur décrit $D_1$.

Notons $x_s(t)$ le nombre d'instructions CPU exécutées durant l'appel système $s$ sur le système sain et $x'_s(t)$ le nombre d'instructions CPU exécutées durant le même appel système $s$ sur un système inconnu.

Le principe de PatchFinder repose sur la comparaison de ces deux valeurs. Si la différence est trop élevée, le système est considéré comme étant compromis par un rootkit. Voyons comment cette approche peut être transposée dans un contexte extérieur à la matrice (analyse passive) : nous définissons les hypothèses nulles et alternatives :

$$\begin{cases} \mathcal{H}_0 : & \text{l'appel système } s \text{ n'est pas hooké} \\ \mathcal{H}_1 : & \text{l'appel système } s \text{ est hooké} \end{cases}$$

Pour $t$ allant de 1 à $n$, nous exécutons un code en mémoire virtuelle qui appelle $s$. Ensuite, nous calculons le nombre moyen d'instructions CPU $m$.

Nous notons $m_0$ le nombre moyen d'instructions exécutées durant l'invocation de la fonction d'API $s$ sur le système sain et $m$ le même calcul sur le système inconnu.

En choisissant l'estimateur $m$, nous pouvons définir un test bilatéral comme suit :

$$\begin{cases} \mathcal{H}_0 : & m = m_0 \\ \mathcal{H}_1 : & m \neq m_0 \end{cases}$$

En effet, si un hook est installé sur le système pour la fonction $s$, le nombre moyen d'instructions CPU $m$ est soit plus élevé que $m_0$ (si le hook instrumente la fonction $s$ en ajoutant des fonctionnalités) ou moins élevé que $m_0$ (dans le cas où le hook inhibe la fonction $s$).

Étant donné un taux de faux positifs acceptable [Fil07c, FJ07], si $|m - m_0|$ est supérieur à un seuil de décision fixé, nous considérons que la fonction $s$ est hookée (et donc que le système cible est corrompu).

En analysant le système depuis l'extérieur, nos mesures sont effectuées de manière complètement furtive. La fonction de détection est donc plus difficile à contourner. Notre échantillonnage peut être effectué en exécutant le code qui appelle $s$ depuis une zone mémoire aléatoire. En prenant $n$ mesures, nous sommes capables d'estimer la distribution de la loi de probabilité sous l'hypothèse nulle $\mathcal{H}_0$ plus précisément, et donc de détecter (avec un taux de faux positifs contrôlé) toute variation suspecte vis-à-vis de cette loi.

C. Lauradoux propose dans [Lau08] une méthode de détection des rootkits VMBR (i.e. fondés sur l'utilisation d'un hyperviseur de machine virtuelle). Il propose un estimateur permettant de capturer une dépendance d'ordre 2 (covariance) en mesurant les écarts à la régularité du temps d'exécution d'un morceau de code. Il mène ses expérimentations en modifiant l'algorithme d'ordonnancement sous Linux. Je propose de reproduire ces expérimentations sous Windows. Considérons le morceau de code suivant :

```
int i=sampling_size;
_asm {
            rdtsc
            mov dword ptr [start], eax
            mov dword ptr [start+4], edx
};

while(i>0)
{
      i--;
      _asm {
                  rdtsc
                  mov dword ptr [end], eax
                  mov dword ptr [end+4], edx
      };
      data[i]=end-start;
      start=end;
}
```

Notons $N$ la taille de notre échantillon (`sampling_size` dans le morceau de code). Ce programme récupère une séquence de mesures $X_i = t_i - t_{i-1}$, $i = 1, \ldots, N$ calculées à partir des valeurs $t_i$, $i = 0, \ldots, N$ du registre TSC (*Time Stamp Counter*). Il a observé que le spectre d'autocorrelation :

$$\{cov(k) = \frac{1}{N} \sum_{i=1}^{N-k} (X_i - \overline{X})(X_{i+k} - \overline{X}) \,|\, k = 1, \ldots, K\}.$$

permet de détecter l'activité d'une machine. La figure 6.2 représente le spectre d'auto-corrélation obtenu. Les

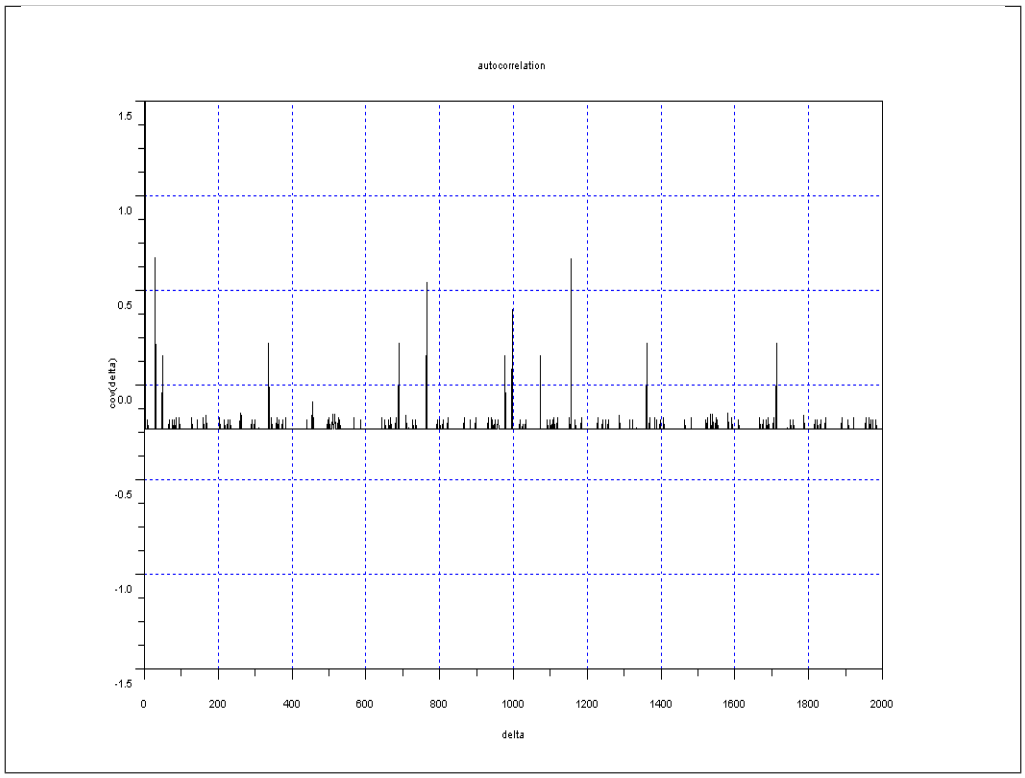



FIG. 6.2 – Spectre d'auto-corrélation

grandes variations sont provoquées par les interruptions. Effectuons à présent la mesure en masquant les interruptions (le morceau de code précédent est balisé par les instructions `_asm cli` et `_asm sti`). Nous obtenons le spectre d'autocorrélation représenté sur la figure 6.3. Remarquons que (en mode protégé) le code doit s'exécuter en mode noyau (les instructions `_asm cli` et `_asm sti` sont privilégiées. Windows NT positionne l'IOPL à 0 donc seules les applications s'exécutant en mode noyau peuvent exécuter les instructions `cli/sti`). Nous pouvons observer que lorsque nous ignorons les interruptions, l'autocorrélogramme montre des phénomènes quasi-périodiques. À l'inverse, lorsque les interruptions sont prises en compte, l'autocorrélogramme n'indique pas de corrélations particulières

Si nous effectuons les mêmes mesures dans un hyperviseur de VM (ici VMWare), nous obtenons les spectres d'autocorrélation représentés sur les figures 6.4 et 6.5 respectivement (la figure 6.5 représente l'exécution du morceau de code lorsque les interruptions sont masquées). Nous pouvons observer, en comparant les figures 6.3 et 6.5, que nous ne retrouvons pas les mêmes corrélations. Lorsque nous modifions le mode d'exécution sur la machine virtuelle (changement d'algorithme d'ordonnancement ou désactivation des interruptions), cela n'affecte pas l'exécution des processus sur le système hôte. Le spectre d'autocorrélation nous permet donc de détecter une activité anormale.
Nous pouvons observer en outre que lorsque les interruptions sont masquées dans une VM, le processus stochastique $(X_i)$ apparaît comme un bruit blanc : l'autocorrelation est (presque) nulle partout sauf à l'origine. Ce n'est pas du tout le cas lorsque les interruptions sont masquées sur le système hôte.

**Définition 6.3.2** *Un processus stochastique $(X_i)$ est qualifié de bruit blanc si :*
- $E[X_i] = 0$
- $E[X_i^2] = \sigma^2$
- $\forall i \neq j$, $E[X_i X_j] = 0$

Nous pouvons donc conjecturer que cette méthode de détection peut être modélisée par un test statistique de blancheur, tel que le test du Portemanteau.

La transposition de cette méthode de détection depuis l'extérieur de la matrice est immédiate. La seule diffé-

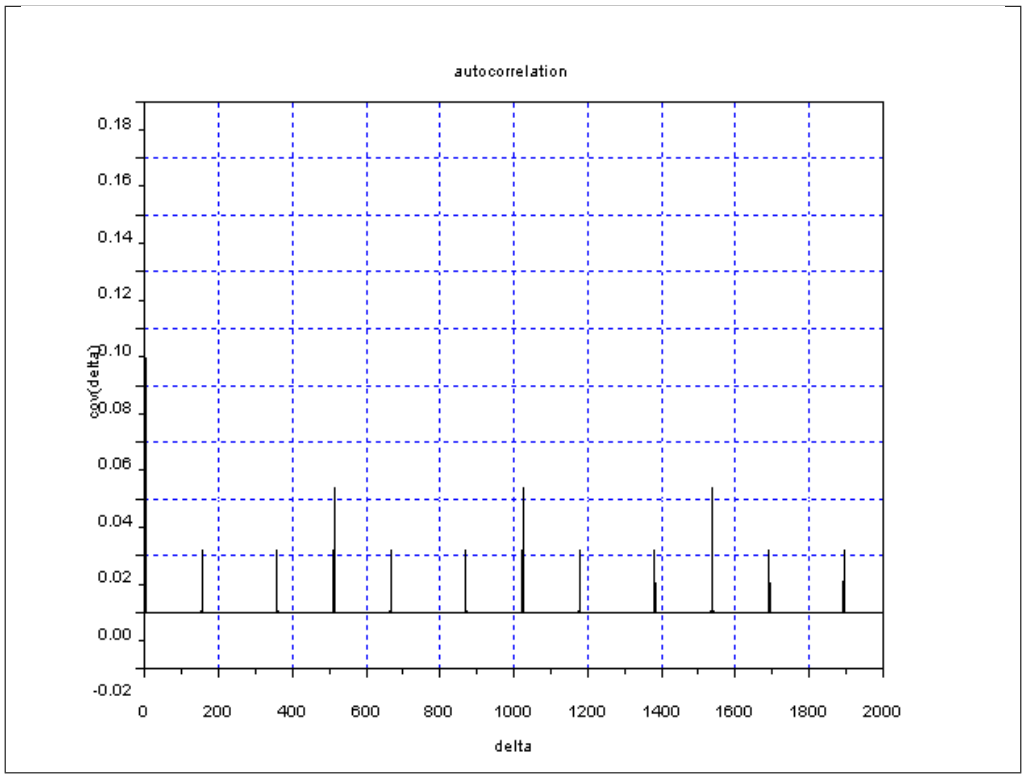



FIG. 6.3 – Spectre d'auto-corrélation avec masquage des interruptions

rence réside dans l'accès au registre TSC. Si ce registre de 64 bits ne peut pas être modifié par un rootkit, les registres `eax` et `edx` contenant le résultat de l'exécution de l'instruction `rdtsc` peuvent être modifiés/filtrés par un hyperviseur. La lecture directe du regsitre CPU depuis l'extérieur de la matrice permet d'éluder ce problème.
Comme pour l'estimateur de PatchFinder, il est possible de randomiser les contrôles et de les itérer pour améliorer la précision de l'estimation.

Je pense que l'utilisation d'un CPU logiciel permet d'appliquer de manière sûre les méthodes de détection de rootkit par contrôle de modèle. Les algorithmes de détection connus se transposent sans difficulté dans ce contexte. Une telle approche pourrait être mise en oeuvre par un composant matériel spécialisé. Des systèmes de détection matériels, comme Copilot [PJFMA04], ont été proposés mais ils sont aujourd'hui fondés exclusivement sur le contrôle d'intégrité et n'utilisent pas suffisamment l'omniscience du point de vue matériel. L'analyse des canaux cachés (temps, stockage) induis par la présence d'un rootkit sur le système peut pourtant être menée de manière fiable depuis l'extérieur de la matrice et fournit des méthodes de détection plus génériques que l'exploitation d'un problème structurel du jeu d'instruction CPU, susceptible d'être corrigé (masquage du temps, par exemple, dans les CPU Intel IVT et AMD Pacifica).

## 6.4 Evaluation

### 6.4.1 Donc qu'est ce qui est nouveau ?

Nous proposons un système d'analyse fiable, furtif et sécurisé [Jos06b, Jos07a] pour extraire automatiquement (ou interactivement lorsque c'est nécessaire) une information précise sur un rootkit. Nous récupérons une information précise relative avec ses interactions avec l'exécutif du système d'exploitation, plus particulièrement concernant la manière utilisée pour installer et maintenir des hooks dans le système cible.

La conception de notre outil est orientée isolation et furtivité. En fournissant un environnement complètement

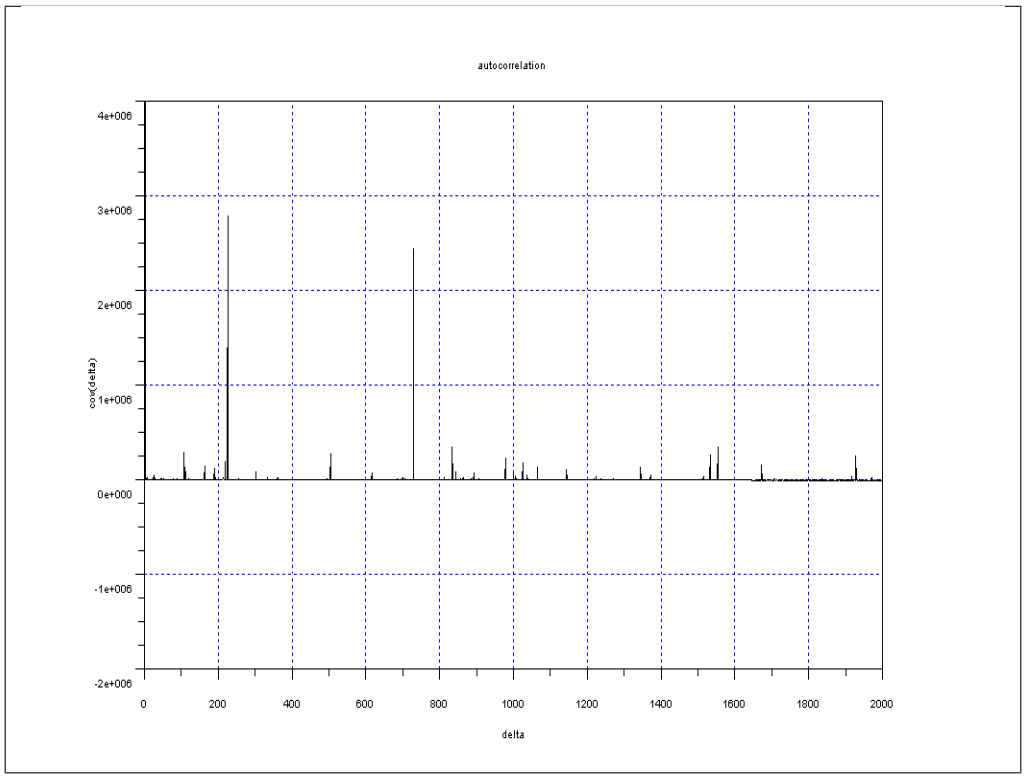



Fig. 6.4 – Spectre d'auto-corrélation (VMware)

contrôlé, nous pouvons analyser la plupart des rootkit sans aucun risque pour le système réel.

Le processus d'analyse tout entier est conduit au sein d'un environnement virtuel (la matrice), qui peut être très difficile à distinguer d'un environnement réel, donc augmentant les compétences techniques requises chez un concepteur de rootkits pour pouvoir fabriquer un rootkit susceptible de s'évader ou de se propager.

Notre approche permet de prendre un instantané fiable et furtif des objets du noyau ou matériels qui sont habituellement modifiés par les infections informatiques implémentant la plupart des technologies rootkit. Cette fonction de surveillance est aussi furtive que possible car nous utilisons une CSIM complète et précise (incluant un CPU virtuel et les composants matériels les plus courants), sur laquelle s'exécute une version non modifiée du système d'exploitation.

Ces contrôles peuvent survenir à tout moment durant le cycle de vie du rootkit sous contrôle : au moment du démarrage de l'OS, du chargement de l'exécutable ou de son exécution. En effet, nous pouvons appliquer des contrôles à chaque cycle du CPU virtuel. Nous avons donc de nombreux degrés de liberté et une fine granularité.

En observant l'ensemble du système hôte corrompu depuis l'extérieur, nous offrons un nouveau point de vue qui permet l'étude des algorithmes de détection de rootkits actuels (à l'intérieur de la matrice) et de valider leur pertinence dans ce nouveau contexte (à l'extérieur de la matrice).

En comparaison des solutions matérielles de détection des rootkits, et parce que la totalité de la machine peut être inspectée à tout instant durant le processus d'analyse, notre solution ne souffre pas des mêmes limitations et vulnérabilités (*timing attacks*, attaques sur le cache CPU, situations de concurrence).

En comparaison des outils de détection de rootkit actuels (à l'intérieur de la matrice), les même algorithmes de détection sont plus difficiles à contourner :

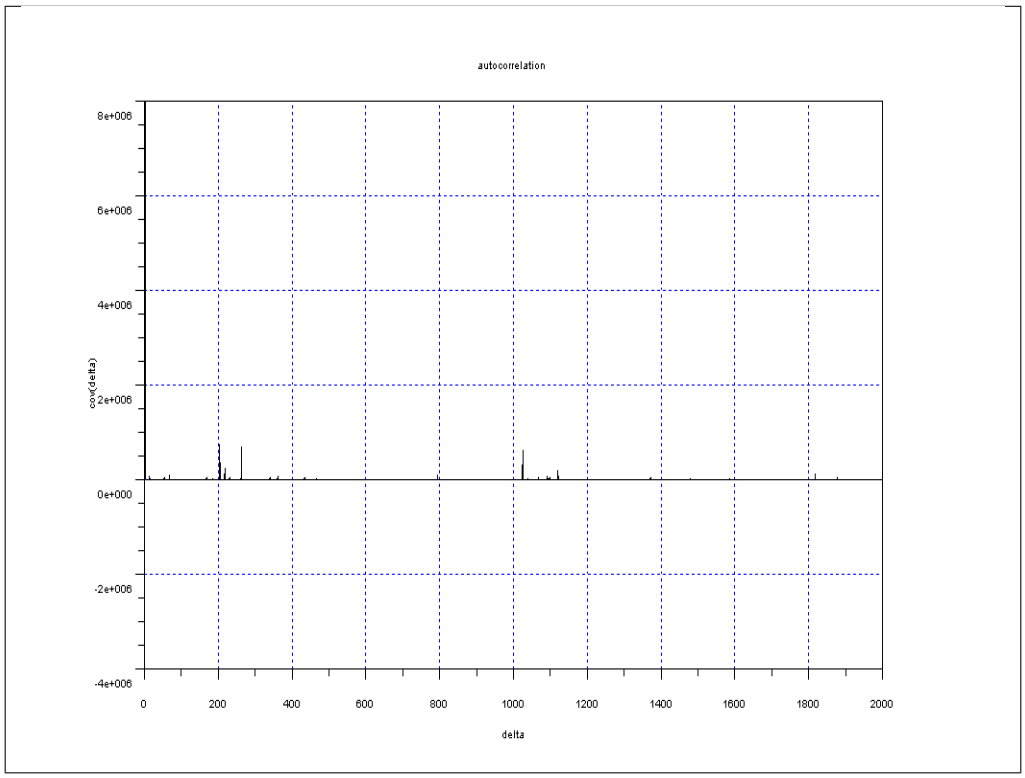



Fig. 6.5 – Spectre d'auto-corrélation (VMware) avec masquage des interruptions

- les fonctions de contrôle d'intégrité ne se heurtent pas aux contraintes et règles imposées par le système d'exploitation ou par le matériel. Des mesures furtives peuvent être pratiquées sans perturber le système d'exploitation et sans être repérables. Parce que les rootkits ne s'exécutent pas au même niveau que la fonction de détection, ils ne peuvent plus perturber son diagnostic.
- pour la même raison, les algorithmes de contrôle de modèle sont plus robustes et difficiles à déjouer. En outre, parce que nous pouvons échantillonner et effectuer des mesures avec un granularité plus fine et une fréquence plus élevée (à chaque cycle du CPU virtuel si nécessaire), il est possible d'obtenir des implémentations plus efficaces de ce type d'algorithme d'apprentissage.

Les algorithmes classiques s'appliquent mais sans les mêmes contraintes opérationnelles. En conséquence, il est plus facile d'appliquer la plupart d'entre eux conjointement et éventuellement d'augmenter la portée de l'analyse.

Durant le processus d'analyse, nous pouvons diriger des transformations complexes sur le programme cible, comme le dépacketage, et donc potentiellement analyser des rootkits blindés.

La plupart des rootkits actuels n'implémentent pas de protection logicielle comme le packetage. Notre moteur de dépacketage, qui est un plug-in de VxStripper, fournit une fonctionnalité de dépacketage générique qui pourrait être utile dans le futur lors de l'analyse de rootkits.

### 6.4.2 Limitations

La technologie rootkit évolue très rapidement et exploite volontiers la complexité structurelle croissante des systèmes d'exploitation et les nouvelles fonctionnalités des composants matériel. Dans ce contexte, spécifier un outil d'analyse fondé sur l'émulation logicielle de l'ensemble des composants matériel d'un PC relève du défi. L'émulateur QEMU, sur lequel se fonde VxStripper, ne propose pas encore de CPU logiciel correspondant aux spécifications des CPU AMD Pacifica ou Intel VT. Nous ne sommes donc pas en mesure de spécifier des méthodes de récupération d'in-

formation pour les rootkits exploitant les extensions de virtualisation proposées par le jeu d'instruction de ces CPUs.

Dans le contexte de l'analyse de rootkits, deux exigences de sécurité doivent être couvertes : l'isolation (confinement, capacité à endiguer la propagation) et la furtivité (si l'émulation est détecté, le rootkit cible ne fournira pas l'information escomptée ou pire, manipulera l'analyse). Nous avons vu au chapitre 5 (section 5.3.4) que plusieurs fonctions de sécurité sont obligatoires pour accroître la sécurité de notre solution. Pour être isolée et sécurisée, un émulateur doit implémenter une protection robuste des périphériques, du filtrage et des mécanismes de quarantaine réseau. Au minimum, un pare-feu doit être installé sur système hôte de façon à surveiller les interfaces de communication réseau entre systèmes d'exploitation hôte et invité. Nous étudions actuellement la possibilité d'inhiber toute interface entre les systèmes hôte et invité, fournissant ainsi une VM hautement sécurisée.

Le problème de détection d'un environnement émulé par un rootkit est également crucial dans le contexte de la détection de rootkit par émulation. Nous avons vu à la section 5.3.4 du chapitre 5 que les VMM souffrent de limitations structurelles qui peuvent être détectée. Les CSIM ne sont pas sensibles aux mêmes attaques, mais sont trop souvent détectables en raison de l'imperfection de l'émulation des composants matériels.

Nous avons présenté à la section 6.3.11 des méthodes de détection de VMM fondées sur le contrôle de modèle. Les CSIM ne sont *a priori* pas sensibles à ces attaques, dans la mesure où les timers et les caches du CPU logiciels sont entièrement émulés et ne sont pas partagés. Cependant, un algorithme de détection peut se fonder sur le contrôle de modèle en comparant l'empreinte du CPU logiciel constituée par le spectre d'autocorrélation avec l'empreinte du CPU réel correspondant, en terme de spécifications.

De manière à être aussi furtif que possible, les erreurs et limitations au coeur du moteur d'émulation doivent être corrigées. Il y a donc du travail à réaliser pour accroître le niveau de sécurité d'un détecteur de rootkits fondé sur un moteur d'émulation.

## 6.5 Conclusion

Nous avons présenté dans ce chapitre le module de VxStripper dédié à l'analyse des programmes furtifs. Il propose un système fiable pour récupérer automatiquement de l'information sur un rootkit et ses interactions avec le système. Cette information peut être utilisée pour définir un schéma de détection adapté.

Parmi les limites de l'outil VxStripper, nous avons identifié les imperfections relatives à l'émulation du CPU logiciel, susceptibles de trahir la présence d'un environnement d'analyse. Nous pouvons observer par ailleurs que l'émulateur QEMU, sur lequel se fonde VxStipper, ne propose pas encore de CPU logiciel supportant les extensions de virtualisation. Il ne permet donc pas l'analyse de certains rootkits VMBR, comme Blue Pill ou Vitriol.

Nous avons étudié la transposition dans VxStripper de certains des algorithmes représentatifs de l'état de l'art dans ce domaine :
- les méthodes de contrôle d'intégrité des rouages internes du système d'exploitation,
- les méthodes de contrôle de modèle.

Nous avons observé qu'il est possible de les appliquer conjointement de manière furtive et éventuellement d'augmenter la portée de l'analyse.

D'autres méthodes de détection intéressantes, actuellement implémentées par des outils forensiques spécialisés, pourraient être transposées dans notre module d'analyse des rootkits :
- les méthodes de comparaison croisée pourraient être implémentées en injectant du code dans la machine virtuelle de manière à générer des appels systèmes de plus ou moins bas niveau ;
- les contrôles de cohérence pourraient être implémentés sur plusieurs structures du noyau pour fournir des bornes intrinsèques caractérisant un état sain ;
- les méthodes à base de signatures pourraient être implémentées en contrôlant plusieurs endroits du système (par exemple, le module noyau d'un rootkit, son dropper, la zone mémoire dans laquelle il se charge, etc.) à des étapes cruciales durant l'exécution du programme cible ou durant le cycle de vie du système virtuel tout

entier (au moment du démarrage, par exemple).

J'envisage comme tâche future l'implémentation de ces méthodes de récupération d'information d'interaction bas niveau avec l'OS. Je souhaite également étudier les possibilités d'extension des CPU logiciels supportés par VxStripper aux jeux d'instructions supportant la virtualisation, afin de pouvoir étudier les rootkits les utilisant.

Nous avons signalé dans le résumé de ce chapitre que notre outil d'analyse s'applique aux rootkits mais également à tout produit de sécurité qui interagit à bas niveau avec le système d'exploitation dans le but d'asseoir sa robustesse ou de cacher des informations en installant des hooks.

En effet, nous pouvons utiliser cet outil pour vérifier la robustesse des mécanismes de sécurité et recouvrer une information sur leurs procédures et fonctionnements internes : comme nous l'avons vu dans la section 6.4, nous sommes capables de collecter de nombreuses données qui révèlent des zones possibles de dissimulation.

Nous pourrions implémenter des modules de diagnostiques plus spécialisés de manière à évaluer la robustesse des fonctions de sécurité et mécanismes au travers d'une approche boîte noire ou boîte grise : la possibilité d'appliquer de la réécriture de code binaire au niveau des instructions ou des *basic blocks* durant l'exécution d'un produit de sécurité et de modifier n'importe quel endroit de la plate-forme toute entière est un composant critique de tout système d'extraction d'information ou d'injection de fautes.

Parmi les méthodes de détection des rootkits, les méthodes de contrôle de modèle, présentées à la section 6.3.11, sont sans doute les plus génériques. Elles s'appliquent en particulier à la détection des rootkits VMBR[7]. Nous avons observé que ces méthodes peuvent être formalisées en terme de tests statistique. Nous proposons dans le chapitre suivant une généralisation de cette approche aux autres méthodes de détection virale.

[7] Observons cependant que ces méthodes s'appliquent aujourd'hui plus à la détection de la présence d'un hyperviseur sous le système d'exploitation qu'à détection d'un rootkit se fondant sur la virtualisation. C'est la raison pour laquelle il faut sans doute redéfinir les bons estimateurs [Fil07c].

Chapitre 7

# Modélisation statistique du problème de la détection virale

Nous avons vu au chapitre précédent que les méthodes de détection des virus furtifs fondées sur le contrôle de modèle peuvent être formalisées en terme de tests statistiques. Nous proposons dans ce chapitre une généralisation de cette approche, en modélisant dans ce cadre théorique le problème de la détection virale. Nous illustrons les concepts introduits par l'étude du cas de l'analyse spectrale.

## 7.1 Introduction

Un résultat fondamental de la virologie informatique est le théorème d'indécidabilité de la détection virale de Cohen [Coh86]. Ce théorème établit qu'il n'y a pas d'algorithme permettant de détecter tous les virus possibles. En 2001, Chess et White [CW00] ont démontré qu'il y a des virus qu'aucun algorithme ne peut détecter, même avec une définition plus générale et assouplie de la notion de détection. Ils ont donc étendu le résultat de Cohen. De manière plus précise, supposons disposer d'un algorithme de détection $D$ pour un virus $v$. Nous pouvons tolérer de ce détecteur qu'il affirme avoir trouvé $v$ dans un programme $P$, à condition que $P$ soit effectivement infecté par un virus (mais qui ne soit pas nécessairement $v$).

On dit donc d'un algorithme $D$ qu'il *détecte généralement* le virus $v$ si et seulement si pour tout programme $P$, $D(P)$ se termine,
- retourne *vrai* si $P$ est effectivement infecté par $v$ et
- retourne quelque chose d'autre que *vrai* si $P$ n'est infecté par aucun virus.

L'algorithme $D$ peut cependant ne retourner aucun résultat pour les programmes infectés par un virus différent de $v$, même si on lui impose de s'arrêter. Une conséquence pratique de ce résultat est qu'il met fin à l'idée selon laquelle il est toujours possible de créer une procédure de détection pour un virus donné, ce même lorsque l'on dispose d'une copie du virus. La conclusion de leur papier identifie le problème de la caractérisation formelle de leur concept de détection générale ainsi que celui des faux positifs et des cas de non-détection comme des problèmes ouverts.

L'objectif de cette partie est de résoudre ce problème et de fournir la caractérisation formelle attendue par Chess et White. En considérant un cadre statistique adapté, nous modélisons d'abord la détection antivirale d'un point de vue statistique et définissons précisemment les limitations et aspects inhérents à la détection en pratique. En outre, là où Chess et White ont seulement donné des résultats d'existence de virus indétectables, nous définissons le concept de simulabilité des tests statistiques et donc caractérisons les différentes méthodes pratiques pour effectivement construire de tels virus indétectables. Nous donnons des applications pratiques relatives à ce dernier point.

Le principal intérêt de notre travail n'est pas d'identifier les techniques les plus adaptées pour construire des virus indétectables, mais de disposer d'un cadre plus puissant pour évaluer et tester les produits antivirus. En effet,

comme il a été établit dans [Fil06, Fil07c], les schémas de détection peuvent être très complexes et une analyse exhaustive des motifs de détection n'est pas faisable. Par conséquent, l'approche statistique développée dans cette partie permet de considérer un produit antivirus d'un point de vue probabiliste ou sur la base d'un échantillonnage. Notre motivation première est l'utilisation des informations statistiques extraites d'un programme viral pour former un test statistique de détection virale.
Nous ne disposons pas de modélisation statistique du problème de la détection virale nous permettant par exemple de mesurer la difficulté à contourner une fonction de détection fondant son verdict sur des données statistiques - ou nous permettant de qualifier ces algorithmes, en terme de résistance aux transformations d'obfuscation.
Nous présentons une modélisation statistique du problème de la détection virale. Nous donnons des résultats constructifs sur les virus indétectables. Nous montrons que toute technique de détection existante peut être modélisée par un ou plusieurs tests statistiques. Les concepts de faux positif et de non détection sont définis précisément. Le concept de simulabilité des tests statistiques est ensuite présenté et nous permet de donner des résultats constructifs sur la manière dont un virus indétectable peut être développé par un attaquant. Des applications pratiques de ce modèle statistique sont proposées, ainsi qu'une reformalisation en terme statistique du résultat d'indécidabilité de la détection virale de Cohen.
Nous nous intéressons aux possibilités de contournement de la fonction de détection par simulabilité statistique en considérant des techniques de détection plus sophistiquées. Nous présentons en premier lieu une nouvelle approche de détection virale par analyse spectrale fondée sur des modèles bayésiens.
Le principal intérêt de cette approche réside dans le fait que nous pouvons définir un tel modèle pour une famille virale quelconque. Tous ces modèles sont dérivés d'un unique modèle initial avec une paramétrisation adaptée.
Nous illustrons cette discussion avec la présentation de techniques de détection virale fondées sur des techniques spectrales et les modèles de Markov cachés. Nous identifions trois problèmes relatifs à la détection virale et les présentons en détails. Nous présentons une solution algorithmique à ces problèmes. Cette discussion nous permet alors de présenter une méthode pratique pour contourner toute méthode de détection de ce type par simulabilité statistique.

Une partie du contenu de ce chapitre, co-écrite avec E. Filiol, a été publiée dans les actes de la conférence EICAR 2007 [FJ07].

Le reste de ce chapitre est organisé de la manière suivante :
- la section 7.2 propose un modélisation statistique du problème de détection virale ;
- la section 7.3 présente une première application de ce formalisme au cas de l'analyse spectrale ;
- les sections 7.4 et 7.5 présentent deux modèles employés par certains moteurs de détection virale, mais également par d'autres applications de protection logicielle dédiée à la détection d'intrusion, et étudie leur instanciation dans le cas de l'analyse spectrale. Ces deux modèles correspondent à une approche bayésienne (méthode de classification naïve de Bayes dans le premier cas, réseau bayésien dynamique par utilisation du modèle de Markov caché dans le second cas).

## 7.2 Modélisation statistique de la détection antivirale

### 7.2.1 Définition du modèle

**Définition 7.2.1** *(Modèle statistique). On appelle modèle statistique tout couple $(\mathcal{X}, (P_\theta)_{\theta\in\Theta})$ où $\mathcal{X}$ est un ensemble dit espace des observations, $\Theta$ est un ensemble dit espace des paramètres et $(P_\theta)_{\theta\in\Theta}$ est une famille de probabilités définies sur une tribu fixée de parties de $\mathcal{X}$.*

**Définition 7.2.2** *(Statistique). Soit $(\mathcal{X}, (P_\theta)_{\theta\in\Theta})$ un modèle statistique, on appelle statistique sur ce modèle toute application $E$, indépendante de $\theta$, mesurable relativement à la tribu $\mathcal{A}$ considérée sur $\mathcal{X}$ et à valeur dans un certain espace $\mathcal{Y}$ muni d'une tribu $\mathcal{B}$. Le modèle $(\mathcal{Y}, (E(P_\theta))_{\theta\in\Theta})$ où pour tout $\theta \in \Theta$, la loi $E(P_\theta)$ est la loi image par $E$ de $P_\theta$ est dit modèle image par $E$ du modèle $(\mathcal{X}, (P_\theta)_{\theta\in\Theta})$.*

Nous nous restreignons au cadre des tests paramétriques : considérant une variable aléatoire $X$, le but est d'estimer les principales caractéristiques de la loi de probabilité $\mathcal{P}$ gouvernant $X$. La loi observée est supposée appartenir à une famille de lois décrites par des densités de probabilités $\{f(x, \theta)\,;\, \theta \in \Theta\}$ qui ne dépendent que d'un seul

paramètre inconnu $\theta$. Nous supposons l'espace des paramètre $\Theta \subseteq \mathbb{N}^k$, où $k$ est la dimension du paramètre $\theta$ (pour une moyenne, par exemple, $k = 1$ alors que pour une variance, $k = 2$). La forme de la fonction de densité est supposée connue.

L'objectif d'un test statistique est de prendre une décision concernant le rejet ou l'acceptation d'une hypothèse $\mathcal{H}$ sur la base de la réalisation d'un échantillon $(X_1, \ldots, X_n)$ permettant d'estimer le paramètre $\theta$ de la loi $\mathcal{P}$. Nous considérons les deux hypothèses suivantes :

- l'hypothèse nulle $\mathcal{H}_0 : \theta \in \Theta_0$, correspondant au cas où la distribution observée de la variable aléatoire $X$ est proche de celle observée pour un programme sain ;
- l'hypothèse alternative $\mathcal{H}_1 : \theta \in \Theta_1$, correspondant au cas où la distribution observée de la variable aléatoire $X$ trahit la présence d'une infection virale.

La construction d'un test repose sur le calcul d'un estimateur ou statistique de test $E \in \mathbb{R}$. Nous noterons $e$ la valeur réalisée sur un échantillon pour l'estimateur $E$. Selon que l'on est sous l'une ou l'autre des hypothèses du test, cet estimateur suit une loi différente. Un test d'hypothèse consiste donc à décider, sur la base de la valeur observée de l'estimateur sur un échantillon, quelle est la loi gouvernant la population que l'on étudie.

La figure 7.1 illustre le principe d'utilisation d'un test statistique pour fonder la détection virale. Elle représente les fonctions de densité gouvernant respectivement l'hypothèse nulle $\mathcal{H}_0$ et l'hypothèse alternative $\mathcal{H}_1$. La zone de recouvrement des deux distributions définit, un seuil de décision ayant été choisi, les risques de première et seconde espèce, $\alpha$ et $\beta$.

Un détecteur antiviral $D$ va conduire un ou plusieurs tests pour décider si un fichier $F$ est infecté infecté ou

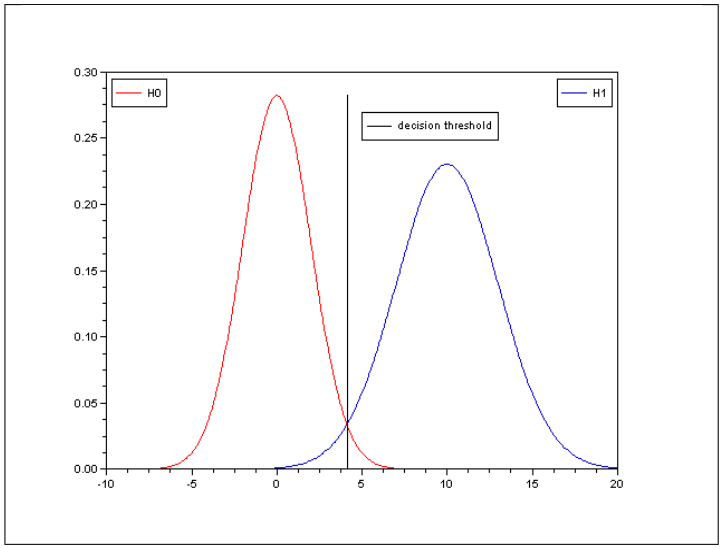



FIG. 7.1 – Modélisation statistique de la détection virale

non. Supposons que le détecteur conduise $n$ tests de détection $\mathcal{T}_1, \ldots, \mathcal{T}_n$, chaque test $\mathcal{T}_i$ étant caractérisé par un risque de première espèce $\alpha_i$ ou probabilité de fausse alarme et par un risque de seconde espèce $\beta_i$ ou probabilité de non-détection.
Une stratégie de détection est caractérisée par la fonction booléenne de détection globale utilisée pour prendre en compte les événements. Par exemple, une première stratégie va consister à prendre en compte la réalisation conjointe des événements pour fonder une décision, tandis qu'une autre stratégie va consister à appliquer la fonction majorité. Les tests peuvent être menés en parallèle, c'est-à-dire que chaque test est appliqué sur la totalité des données, ou en série, c'est-à-dire que chacun d'entre eux s'exerce sur le résultat du précédent. Dans le cas où les tests sont appliqués en série et qu'ils sont supposés indépendants, c'est-à-dire que le résultat de tout test $\mathcal{T}_i$ n'influe pas sur le résultat de tout autre test $\mathcal{T}_j$ pour $i \neq j$, il est très facile de calculer les risques de première et seconde espèce résiduels du

détecteur $D$ :

Lorsque les $n$ tests sont indépendants et appliqués en série, la probabilité de non-détection résiduelle du détecteur $D$ est égale à :

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

En effet, pour $n = 2$,

$$\begin{array}{rcl} \beta & = & p[\text{non-détection} \,|\, \mathcal{H}_1^1 \text{ et } \mathcal{H}_1^2 \text{ vraie}] \\ & = & p[\text{non-détection} \,|\, \mathcal{H}_1^1 \text{ vraie}] \times p[\text{non-détection} \,|\, \mathcal{H}_1^2 \text{ vraie}] \\ & = & \beta_1 \times \beta_2 \end{array}$$

puisque les hypothèses alternatives $\mathcal{H}_1^1$ et $\mathcal{H}_1^2$ sont indépendantes. Le résultat s'en déduit en itérant le même raisonnement pour $n > 2$. $\square$

Sous les mêmes hypothèses, la probabilité de fausse alarme résiduelle du détecteur $D$ est égale à :

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

En effet, pour $n = 2$, le test $\mathcal{T}_1$ décide en moyenne et à tort que $\alpha_1 \times N$ fichiers sont sont infectés parmis $N$. Sur ces $\alpha_1 \times N$ fichiers, le test $\mathcal{T}_2$ est ensuite appliqué avec un nombre de fichiers moyen considérés à tort comme infecté égal à $\alpha_2 \times \alpha_1 \times N$. Soit une probabilité de fausse alarme résiduelle égale à $\alpha_2 \times \alpha_1$. Le résultat s'en déduit en itérant le même raisonnement pour $n > 2$. $\square$

Dans le cas où une dépendance existe entre les tests, alors la probabilité de non-détection $\beta_2$ dépends de $\beta_1$. S'il on note $A_i$ l'événement [non-détection $|\, \mathcal{H}_1^i$ vraie], on a alors :

$$p[A_1 \wedge \ldots \wedge A_n] = p[A_1]p[A_2|A_1] \ldots p[A_n|A_1 \wedge \ldots \wedge A_{n-1}].$$

Plus la dépendance des tests est forte, moins vite la probabilité de non-détection résiduelle $\beta$ converge vers 0 lorsque le nombre de tests augmente.

### 7.2.2 La simulabilité des tests statistiques

Nous allons étudier les possibilités offertes par un attaquant pour contourner un détecteur probabiliste. La notion de simulabilité des tests statistiques [Fil05b, Fil07c] permet de formaliser la situation où l'attaquant, connaissant les outils statistiques de détection, les utilise contre le défenseur. Donnons une définition générale de la simulabilité des tests :

**Définition 7.2.3** *Simuler un test consiste, pour un adversaire, à introduire, dans une population donnée, un biais statistique non détectable pour un évaluateur, relativement à un ensemble de tests donnés.*

Il existe deux formes de simulabilité :
- la première ne dépend pas des paramètres des tests considérés par le défenseur (*simulabilité forte*) ;
- la seconde dépend des paramètres des tests que l'on souhaite simuler (*simulabilité faible*).

Donnons une définition plus précise de ces deux concepts :

**Définition 7.2.4** *(Simulabilité forte d'un test statistique [Fil07c]). Soit une propriété $P$ et un test $T$ destiné à vérifier si $P$ est valide pour une population donnée. Simuler fortement ce test, c'est concevoir ou modifier la population de manière à ce que $T$ décide systématiquement, au risque d'erreur près, que $P$ est vérifiée sur la population, mais qu'il existe un test $T'$ tel que ce dernier décide du contraire.*
*De la même manière, on dira que l'on simule fortement $n$ tests $(T_1, \ldots, T_n)$ si leur application conjointe conduit à décider que la propriété $P$ est vraie sur la population mais ne l'est plus si on considère un $(n+1)$-ième test $T_{n+1}$.*

La notion de simulabilité formalise à la fois la recherche de vulnérabilités dans un produit de détection virale et la situation où une trappe a été volontairement placée dans un tel produit par leur concepteur :
- dans le premier cas, un évaluateur ou un auteur de virus cherche à déterminer dans quelle mesure il peut contourner l'ensemble des tests implémentés par le détecteur probabiliste ;
- dans le second cas, le concepteur du produit a volontairement supprimé un test, gardé secret.

**Définition 7.2.5** *(Simulabilité faible d'un test statistique). Soit une propriété $P$ et un test $T$ destiné à vérifier si $P$ est valide pour une population donnée. Simuler faiblement ce test, c'est introduire dans la population une nouvelle propriété $P'$ modifiant partiellement la propriété $P$, de manière à ce que $T$ décide systématiquement, au risque d'erreur près, que $P$ est vérifiée sur la population.*

La propriété $P'$ est telle que le ou les estimateurs utilisés restent dans la région d'acceptation du test, en exploitant la marge offerte par les paramètres du test.

### 7.2.3 Une version statistique du résultat d'indécidabilité de Cohen

Nous avons vu que lorsque $n$ tests indépendants $\mathcal{T}_1, \ldots, \mathcal{T}_n$ sont appliqués en série par un détecteur $D$, la probabilité de non-détection résiduelle du détecteur $D$ est égale à :

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

S'il on peut faire tendre la probabilité de non-détection vers 0, celle-ci ne sera jamais nulle. En effet, si une série de tests suffisamment grande produisait une probabilité de non détection exactement égale à 0, on aboutirait alors à la conclusion contradictoire [Coh86] que le problème de la détection est décidable.
Dans sa thèse, Cohen a démontré de manière informelle l'indécidabilité du problème de la détection en utilisant comme support un algorithme viral appelé virus contradictoire, et noté `CV`, dont nous donnons le pseudo-code ci-dessous :

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

Nous voyons que quelle que soit la procédure de détection $D$, il est toujours possible de construire un virus qui ne sera pas détecté par $D$.
La notion de simulabilité des tests statistiques nous permet de donner une illustration de ce résultat dans le cas où le détecteur $D$ est modélisé par un ensemble de tests statistiques $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Le pseudo-code peut alors être réécrit comme suit :

```
CV(){
   .....
   simul(D,V)

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

   main(){
      si non D(CV) alors {
          infection();
```

```
            si condition vraie alors charge_finale();
        } sinon {
            V' = simul(D,CV)
            exécuter V'
        }
        aller au programme suivant
    }
}
```

L'algorithme viral obtenu est tout aussi contradictoire : si $D$ décide que `CV` est un virus, alors `CV` mute en simulant les tests $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Dans le cas contraire, l'infection survient et `CV` se révèle bien être un virus.

## 7.3 Application à l'analyse spectrale

Le principe de l'analyse spectrale repose sur l'analyse de la distribution statistique des instructions d'un programme. Considérons, pour un compilateur donné $\mathcal{C}$, un spectre constitué d'une liste d'instructions $(I_i)_{1 \leq i \leq c}$, chacune d'entre elles étant caractérisée par un nombre théorique moyen d'occurence $\mu_i$ dans les programmes non infectés générés par ce compilateur. Les spectres de référence du compilateur $\mathcal{C}$ sont notés :

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

### 7.3.1 Modèle de détection élémentaire avec loi alternative connue

Considérons le cas de la détection de l'utilisation anormale d'une instruction assembleur $I$ dans un programme. Nous supposons que dans le cas d'un fichier sain, l'instruction $I$ a une probabilité d'apparition $p = p_0$ et que pour les fichiers infectés par certains virus, utilisant un moteur de mutation donné, l'instruction $I$ a une probabilité d'occurence $p_1$. Si l'on considère un programme comportant $N$ instructions, le nombre moyen d'occurence de l'instruction $I$ vaut $\mu = \mu_0 = N \times p_0$ dans le cas où le fichier est sain et $\mu = \mu_1 = N \times p_1$ sous l'hypothèse alternative. En considérant chaque instruction comme la réalisation d'une variable aléatoire de Bernouilli $X$, et que les épreuves sont indépendantes, nous pouvons donc approximer par la loi normale (lorsque $N > 30$) la distribution de la somme :

$$E_N = \sum_{i=0}^{N} \mathbf{1}_{X_i = I}.$$

Nous pouvons à présent formuler les hypothèses du test :

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

Sous l'hypothèse nulle, $E_N$ suit asymptotiquement une loi normale $\mathcal{N}(\mu_0, \sigma_0)$ et sous l'hypothèse alternative, $E_N$ suit asymptotiquement une loi normale $\mathcal{N}(\mu_1, \sigma_1)$, où pour $i = 0, 1$, l'écart type $\sigma_i$ vaut $\sqrt{N \times p_i \times (1 - p_i)}$. Ayant fixé le seuil de signification $\alpha$ et la loi de probabilité décrivant le comportement moyen de l'estimateur choisi, d'un échantillon à l'autre, selon que l'hypothèse nulle est valide ou non, nous pouvons calculer le seuil de décision $S$. On a :

$$\alpha = p[E > S \,|\, \mathcal{H}_0 \text{ vraie}] = 1 - F_{\mathcal{N}(0,1)}\left(\frac{S - Np_0}{\sigma_0}\right),$$

où $F_{\mathcal{N}(0,1)}$ désigne la fonction de répartition de la loi normale centrée réduite, d'où :

$$S = \sigma_0 F^{-1}_{\mathcal{N}(0,1)}(1 - \alpha) + Np_0.$$

Si l'on fixe également le risque de non détection $\beta$, il nous reste à calculer le nombre minimal d'instructions $N_{\min}$ pour lequel le test est valide (pour le couple $(\alpha, \beta)$ choisi). On a :

$$\beta = p[E < S \,|\, \mathcal{H}_1 \text{ vraie}] = F_{\mathcal{N}(0,1)}\left(\frac{S - Np_1}{\sigma_1}\right).$$

On en déduit la valeur de $N_{\min}$ (indépendante de la valeur du seuil de décision $S$) :

$$N_{\min} = \left( \frac{F^{-1}_{\mathcal{N}(0,1)}(\beta)\sqrt{p_1(1-p_1)} - F^{-1}_{\mathcal{N}(0,1)}(1-\alpha)\sqrt{p_0(1-p_0)}}{p_0 - p_1} \right)^2 .$$

### 7.3.2 Modèle de détection élémentaire avec loi alternative inconnue

Considérons à présent le cas où l'on ne connaît pas la loi alternative. La seule connaissance que nous avons concerne la loi gouvernant l'hypothèse nulle : $\mathcal{H}_0 : \mu = \mu_0$. Nous supposons comme précédemment que la loi asymptotique de $E_N$ est la loi normale $\mathcal{N}(\mu_0, \sigma_0)$. Nous allons donc considérer comme viral tout programme pour lequel le nombre d'apparition de l'instruction $I$ s'écarte trop en moyenne de cette loi de référence. De nombreuses techniques de polymorphisme/métamorphisme utilisant des techniques de réécriture, cette instruction $I$ peut voir sa fréquence d'apparition diminuer (si elle est par exemple remplacée par une autre instruction) ou augmenter (dans le cas par exemple où des instructions de bourrage ou de code mort sont insérées). Un programme infecté par des codes mettant en jeu ce type de technique sera gouverné par l'hypothèse alternative $\mathcal{H}_1 : \mu \neq \mu_0$. Ayant fixé le seuil de signification $\alpha$ et la loi de probabilité décrivant le comportement moyen de l'estimateur choisi, nous pouvons calculer le seuil de décision $S$. On a :

$$\frac{\alpha}{2} = p\left[ \frac{|E-\mu_0|}{\sigma_0} > \frac{S-\mu_0}{\sigma_0} \,|\, \mathcal{H}_0 \text{ vraie} \right] = 1 - F_{\mathcal{N}(0,1)}\left( \frac{S - Np_0}{\sigma_0} \right).$$

D'où :

$$S = \sigma_0 F^{-1}_{\mathcal{N}(0,1)}(\frac{\alpha}{2}) + Np_0.$$

La règle de décision est alors la suivante, pour $E = e$ sur le fichier en cours d'analyse : si le rapport critique $\frac{|e-\mu_0|}{\sigma_0} \geq F^{-1}_{\mathcal{N}(0,1)}(\frac{\alpha}{2})$, alors on rejette l'hypothèse nulle : le fichier est probablement infecté. Dans le cas contraire, on ne peut pas rejetter l'hypothèse nulle : le fichier est probablement sain.

### 7.3.3 Simulabilité faible de l'analyse spectrale

Considérons à présent la totalité du spectre $spec_j(\mathcal{C}) = (I_i, \mu_i)_{1 \leq i \leq c}$. Lors de l'analyse d'un programme, pour chacune de ces instructions, l'effectif observé $\widehat{\mu}$ est calculé après $n$ observations. Nous pouvons à présent formuler les hypothèses nulle et alternative du test :

$$\begin{cases} \mathcal{H}_0 : \forall i,\, 1 \leq i \leq c,\, \widehat{\mu}_i = \mu_i \\ \mathcal{H}_1 : \exists i,\, 1 \leq i \leq c,\, \widehat{\mu}_i \neq \mu_i \end{cases}$$

Puisque $\sum_{i=1}^{n} \widehat{\mu}_i = n$, l'hypothèse alternative traduit le fait que deux instructions au moins ont une distribution qui s'écarte de la distribution de référence. L'estimateur suivant :

$$D^2 = \sum_{i=1}^{c} \frac{(\widehat{\mu}_i - \mu_i)^2}{\mu_i}$$

admet sous l'hypothèse nulle une loi asymptotique du $\chi^2$ à $c-1$ degrés de liberté. Un seuil de signification $\alpha$ ayant été choisi, la procédure de test consiste à comparer l'estimateur $D^2$ avec le seuil de décision $\chi^2_{\alpha,c-1}$ et à décider comme suit :

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

Nous allons à présent expliquer comment simuler faiblement ce test statistique. Plus précisemment, nous décrivons comment un moteur de mutation peut être paramétré de telle sorte que les variantes polymorphes ou métamorphes du virus aient un spectre d'instruction qui contourne le test d'analyse spectrale. Il s'agit de simulabilité faible car la méthode de contournement dépend des paramètres du test considéré.
Supposons que deux instructions $I_{i_1}$ et $I_{i_2}$ du spectre soient affectées par le processus de mutation du code. Notons

$D_0^2$ la valeur de l'estimateur du $\chi^2$ pour la souche initiale. Nous supposons que le concepteur du virus l'a conçu de sorte que :

$$D_0^2 = \sum_{i=1}^{c} \frac{(\widehat{\mu}_i - \mu_i)^2}{\mu_i} \leq \chi^2_{\alpha,c-1}.$$

Supposons que le processus de mutation induise les modifications suivantes :

$$\begin{aligned} \widehat{\mu}'_{i_1} &= \widehat{\mu}_{i_1} + \delta_1 \\ \widehat{\mu}'_{i_2} &= \widehat{\mu}_{i_2} + \delta_2 \end{aligned}$$

La valeur de l'estimateur sur la version modifiée du virus étant notée $D_1^2$, on a :

$$D_1^2 - D_0^2 = \frac{\varepsilon}{\mu_{i_1}\mu_{i_2}}$$

avec :

$$\varepsilon = 2\delta_1(\widehat{\mu}_{i_1} - \mu_{i_1})\mu_{i_2} + \delta_1^2\mu_{i_2} + 2\delta_2(\widehat{\mu}_{i_2} - \mu_{i_2})\mu_{i_1} + \delta_2^2\mu_{i_1}.$$

Le moteur de mutation devra donc aléatoirement modifier les instructions $I_{i_1}$ et $I_{i_2}$ en respectant les contraites suivantes sur $\delta_1$ et $\delta_2$ :

- l'espérance $E[\varepsilon]$ doit être proche de 0 ;
- la variance $\sigma(\varepsilon)$ doit être faible.

Remarquons qu'il est intéressant d'impliquer un plus grand nombre d'instructions, dans la mesure où cela permet de créer des modifications artificielles visant à compenser d'autres modifications, elles indispensables pour rester dans la zone d'acceptation du test.

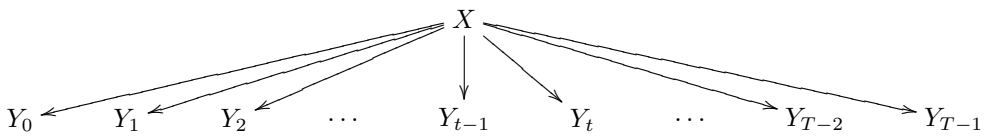
### 7.3.4 Analyse spectrale par utilisation des réseaux bayésiens

Nous avons présenté dans la section précédente des modèles statistiques élémentaires et illustré leur application à la détection virale par analyse spectrale. Nous avons étudié la simulabilité statistique de ces modèles. L'objet des sections 7.4 et 7.5 est de présenter des modèles statistiques plus évolués et d'étudier la simulabilité statistique de ces modèles. Notre choix s'est naturellement porté sur les réseaux bayésiens, en raison de leur utilisation fréquente dans les systèmes de filtrage et de détection d'intrusion. Les réseaux bayésiens correspondent à un type particulier de modèle graphique, dont nous rappelons ici le principe.
Les modèles graphiques sont très pratiques pour décrire l'indépendance conditionnelle et ses conséquences. Cette abstraction permet de représenter un grands nombre d'idées statistiques. Plus précisément, un modèle graphique est un graphe $\mathcal{G} = (V, E)$ où $V$ est un ensemble de noeuds et l'ensemble d'arcs $E$ est un sous-ensemble de $V \times V$.
Un modèle graphique donné est associé à une collection de variables aléatoires et à une famille de distributions de probabilité sur cette collection. L'ensemble des noeuds $V$ correspond exactement aux variables aléatoires, tandis que les arcs représentent les propriétés d'indépendance conditionnelles des variables aléatoires qui sont vraies pour tous les éléments de la famille de distributions de probabilités associée.
Il existe différents types de modèles graphiques, dont dépend l'ensemble des hypothèses d'indépendance conditionnelle spécifiés par ce modèle, ainsi que la famille de distributions de probabilités constituant ces modèles. Nous nous intéressons dans cette section et la suivante à un type particulier de modèle graphique orienté (c'est-à-dire où les arcs sont orientés) appelé réseau bayésien.
Nous obtenons une représentation graphique des relations d'indépendance conditionnelle en appliquant la propriété suivante, appelée propriété de Markov locale orientée : une variable est conditionnellement indépendante de ses non-descendants, connaissant ses parents immédiats.
À titre d'exemple, le réseau bayésien suivant :

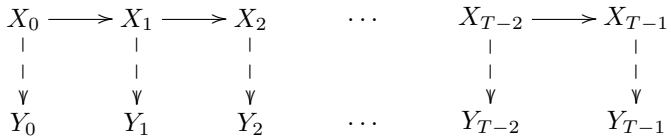
$$X_0 \longrightarrow X_1 \longrightarrow X_2 \qquad \cdots \qquad X_{t-1} \longrightarrow X_t \qquad \cdots \qquad X_{T-2} \longrightarrow X_{T-1}$$

représente une chaîne de Markov d'ordre 1 : la variable aléatoire $X_t$ est conditionnellement indépendante des variables $(X_s)_{s<t-1}$, connaissant $X_{t-1}$.

Nous étudions dans la section 7.4 le modèle de Bayes naïf, qui correspond au réseau bayésien suivant :




Conditionnellement à la variable aléatoire $X$, l'observation $Y_t$ est supposées indépendante des variables aléatoires $Y_{\neg t} = (Y_0, \ldots, Y_{t-1}, Y_{t+1}, \ldots, Y_{T-1})$.
Nous étudions dans la section 7.5 le modèle de Markov caché, qui correspond au réseau bayésien suivant :




Dans ce modèle, conditionnellement à la variable aléatoire $X_t$, la variable aléatoire $Y_t$ est indépendante de $\{X_{\neg t}, Y_{\neg t}\}$.

Dans un modèle graphique, les variables aléatoires peuvent être soit cachées, soit observées. Dans le premier cas, leurs valeurs sont inconnues. Elles sont supposées être des variables vraiment aléatoires. Dans le second cas, leur valeur est connue. Nous notons généralement $X$ les variables aléatoires cachées et $Y$ les observations.
Ces deux modèles sont les exemples les plus simples de réseau bayésien. Ils sont en outre largement utilisés dans les systèmes de filtrage et de détection d'intrusion. Nous étudions dans un premier temps leur application à l'analyse spectrale, puis discutons des possibilités de simulabilité statistique de ces modèles.

## 7.4 Analyse spectrale par approche bayésienne

Nous présentons dans cette section une méthode classique pour générer une règles de décision sur la base de données d'apprentissage. Cette méthode, appelée classification "naïve" de Bayes, en raison de l'hypothèse d'indépendance conditionnelle très forte (naïve) sur laquelle elle repose, est implémentée dans de nombreux logiciels dédiés au filtrage et à la détection d'intrusion. Elle est très simple à implémenter et la complexité calculatoire de l'algorithme est optimale au regard des autres méthodes de classification [Elk97]. Nous présentons son application dans le contexte de l'analyse spectrale. Un test statistique bayésien peut être formé sur la base de cette méthode de classification. Nous étudions la simulabilité faible d'un tel test.

Dans l'approche de la statistique inférentielle que nous avons présenté dans la section précédente, qui est l'approche classique, le paramètre $\theta$, s'il peut être inconnu, reste néanmoins fixé dans $\Theta$. L'approche bayésienne suppose toujours $\theta$ inconnu mais considère que ce paramètre est aléatoire : il est donc régi par une certaine loi de probabilité $\eta$, dite loi *a priori* et supposée connue. L'analyse bayésienne tire parti de l'observation $x$ pour réactualiser la loi *a priori* $\eta$ : on construit, sur la base de la loi *a priori* $\eta$ et de l'observation $x$, une loi $P^x$ dite loi *a posteriori* car déterminée après avoir observée $x$. Le principe de l'inférence bayésienne consiste à corriger l'*a priori* que l'on suppose sur $\theta$ au travers de la loi $\eta$, par l'information apportée par $x$ en utilisant la loi $P^x$. Ce principe d'inférence constitue le principe bayésien.

**Définition 7.4.1** *(Modèle statistique bayésien [Fou02]). On appelle modèle statistique bayésien tout modèle statistique $(\mathcal{X}, (P_\theta)_{\theta\in\Theta})$ tel que l'espace des paramètres $\Theta$ soit muni d'une loi de probabilité $\eta$, dite loi a priori, une tribu $\mathcal{C}$ étant fixée sur $\Theta$.*

Nous allons décrire à présent une méthode de classification fondée sur l'approche bayésienne. L'algorithme de Bayes naïf part du principe qu'il existe un modèle génératif des exécutables : ils sont générés par un modèle de mélange, dont les composantes sont les catégories d'exécutables :

$$c_j \in \mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}.$$

Ces dernières sont des variables cachées, dans la mesure où elles ne sont pas observées. On note $X$ la variable aléatoire représentant la catégorie d'exécutable. La génération d'un exécutable par ce modèle se fait de la manière suivante :

- choix d'une classe $X = c_j$ d'exécutable ;
- génération de l'exécutable $Y$, sur la base d'un ensemble de mots (typiquement le jeu d'instructions $\mathcal{W}$ du compilateur), avec des paramètres qui varient en fonction de la classe d'exécutable qui a été choisie.

L'algorithme est dit « naïf » car il s'appuie sur une hypothèse forte : la présence d'une instruction $w_i$ dans le programme est supposée indépendante de la présence des autres instructions. En outre, la positions des instructions dans le programme et leur agencement les unes par rapport aux autres ne sont pas pris en compte.

Nous pouvons identifier deux variantes classiques de l'algorithme de Bayes naïf, qui correspondent à deux modèles génératifs différents :
- le modèle de Bernouilli multivarié ;
- le modèle multinomial.

Ces deux variantes sont exposées et comparées dans [MN98], où elles sont appliquées à la classification de documents. Je présente leur utilisation dans le contexte de l'analyse spectrale.

Un exécutable généré par le modèle de Bernouilli multivarié est caractérisé par la présence ou l'absence des mots $w_i$ du jeu d'instruction $\mathcal{W}$ : il peut être caractérisé par un vecteur binaire $b = (b_1, \ldots, b_{|\mathcal{W}|})$, avec $b_i = 1$ si l'instruction $w_i$ est présente dans l'exécutable et $b_i = 0$ sinon.

Un exécutable généré par le modèle multinomial est caractérisé par les instructions du programme et intègre également leur nombre d'occurence.

La classification par utilisation de l'algorithme de Bayes naïf se fonde sur le calcul de vraisemblance suivant :

$$p(X = c_i|Y = d) = \frac{p(X = c_i)p(Y = d|X = c_i)}{p(Y = d)} = \frac{p(X = c_i)p(Y = d|X = c_i)}{\sum_{j=1}^{|\mathcal{C}|} p(X = c_j)p(Y = d|X = c_j)}.$$

Les $p(X = c_i)$ sont estimés lors de la phase d'apprentissage en calculant la fréquence d'apparition de chaque classe. Reste donc à estimer les membres $p(Y = d|X = c_i)$, correspondant à la probabilité de générer l'exécutable $d$ lorsque l'on est dans la classe $c_i$.

Dans le cas du modèle de Bernouilli multivarié, où seule compte la présence ou l'absence dans le programme de chaque instruction du jeu d'instruction $\mathcal{W}$, la probabilité de générer un programme $d$ correspond à la probabilité de générer le vecteur binaire associé $b = (b_1, \ldots, b_{|\mathcal{W}|})$. Si on note $B_j = \mathbf{1}_{b_j=1}$ qui vaut 1 si $w_j \in d$ et 0 sinon, on a :

$$\begin{aligned}
p(Y = d|X = c_i) &= \prod_{j=1}^{|\mathcal{W}|} p(b_j|X = c_i) \\
&= \prod_{j=1}^{|\mathcal{W}|} \left(\mathbf{1}_{b_j=1} p(W = w_j|X = c_i) + \mathbf{1}_{b_j=0}(1 - p(W = w_j|X = c_i))\right) \\
&= \prod_{j=1}^{|\mathcal{W}|} \left(B_j.p(W = w_j|X = c_i) + (1 - B_j)(1 - p(W = w_j|X = c_i))\right)
\end{aligned}$$

La probabilité $p(W = w_j|X = c_i)$ qu'une instruction donnée $w_j$ apparaisse dans le programme d'une catégorie donnée $c_i$ est estimé durant la phase d'apprentissage en calculant la proportion de programmes de la classe $c_i$ qui contiennent l'instruction $w_j$. Notons $B_{j,k} = \mathbf{1}_{b_j=1,Y=d_k}$ qui vaut 1 si $w_j \in d_k$ et 0 sinon. On évite d'avoir des probabilités nulles en utilisant la formule suivante :

$$p(W = w_j|X = c_i) = \frac{1 + \sum_{k=1}^{|\mathcal{D}|} B_{j,k} p(X = c_i|Y = d_k)}{2 + \sum_{k=1}^{|\mathcal{D}|} p(X = c_i|Y = d_k)},$$

où $\mathcal{D} = \{d_1, \ldots, d_{|\mathcal{D}|}\}$ désigne l'ensembles des données d'apprentissage.

Dans le cas du modèle multinomial, un programme $d$ est généré en tirant aléatoirement une taille $|d|$, puis en tirant de façon indépendante $|d|$ instructions dans $\mathcal{W}$. Le tirage est effectué avec remise, de façon a prendre en compte le nombre d'occurence d'une instruction de $\mathcal{W}$. Ce tirage se fait selon une loi multinomiale. Si $p(|d|)$ désigne la probabilité de générer un programme de longueur $|d|$ et $N_j$ désigne le nombre d'occurences de l'instruction $w_j$ dans le programme $d$, on a :

$$p(Y = d|X = c_i) = p(|d|)|d|! \prod_{j=1}^{|\mathcal{W}|} \frac{p(W = w_j|X = c_i)^{N_j}}{N_j!}.$$

La probabilité $p(W = w_j|X = c_i)$ qu'une instruction donnée $w_j$ apparaisse dans le programme d'une catégorie donnée $c_i$ est estimé durant la phase d'apprentissage en calculant la proportion du mot $w_j$ parmi tous les mots du jeu d'instruction $\mathcal{W}$ dans les programmes de la classe $c_i$.
Notons $N_{j,k}$ le nombre d'occurences de l'instruction $w_j$ dans le programme $d_k$. Comme précédemment, on évite d'avoir des probabilités nulles en utilisant la formule suivante :

$$p(W = w_j|X = c_i) = \frac{1 + \sum_{k=1}^{|\mathcal{D}|} N_{j,k} p(X = c_i|Y = d_k)}{2 + \sum_{j=1}^{|\mathcal{W}|} \sum_{k=1}^{|\mathcal{D}|} N_{j,k} p(X = c_i|Y = d_k)}.$$

Dans les deux cas, nous utilisons les données d'apprentissage pour former une règle de décision permettant, à partir des caractéristiques spectrales extraites d'un programme, de décider sa classe d'appartenance. Étant donné un programme $d$, on détermine la classe $C$ la plus probable en calculant :

$$g(w_1, \ldots, w_{|\mathcal{W}|}) = \operatorname*{argmax}_c p(X = c) \prod_{i=j}^{|\mathcal{W}|} p(W = w_j \mid X = c).$$

Considérons un problème de test de modèle statistique $(\mathcal{X}, (P_\theta)_{\theta \in \Theta})$ et d'hypothèse à tester $\Theta_0 \subset \Theta$ dans un contexte bayésien, $\eta$ étant la loi *a priori*. Au regard du principe bayésien, toute inférence se fait, après avoir observé $x$ dans $\mathcal{X}$, au travers de la loi *a posteriori* $P^x$ qui traduit une combinaison de l'information contenue dans $\eta$ et de celle apportée par $x$. Dans ce contexte, les probabilités $P^x(\Theta_0)$ et $P^x(\Theta_0^c)$ expriment respectivement les probabilités de l'hypothèse nulle et de l'hypothèse alternative. Leur signification induit à fonder des tests, dits tests bayésiens, sur ces quantités. Ainsi, si l'on souhaite se prémunir contre un rejet à tort de l'hypothèse nulle $\mathcal{H}_0$, on choisit de ne rejeter cette hypothèse que pour des valeurs de $x$ telles que la probabilité $P^x(\Theta_0^c)$ soit élevée. On se fixe donc un nombre $\gamma$, tel que $\frac{1}{2} \leq \gamma < 1$, qui minore cette probabilité, soit tel que :

$$\gamma \leq P^x(\Theta_0^c)$$

ou de manière équivalente :

$$P^x(\Theta_0) \leq 1 - \gamma \leq \frac{1}{2} \leq \gamma \leq P^x(\Theta_0^c).$$

Un cas particulier courant est celui celui où $\gamma = \frac{1}{2}$ et où donc $P^x(\Theta_0) \leq \frac{1}{2} \leq P^x(\Theta_0^c)$. Ce test rejette l'hypothèse nulle $\mathcal{H}_0$ dès que la probabilité de l'alternative dépasse celle de l'hypothèse nulle.

Nous pouvons construire un test bayésien à partir de l'algorithme de classification correspondant au modèle génératif multinomial, dans le cas où $|\mathcal{C}| = 2$. Notons $c$ et $\neg c$ les deux classes de programmes correspondantes. Typiquement, dans le contexte de l'analyse spectrale, la classe $X = c$ correspond aux programmes générés à partir d'un kit de génération de virus ou à une famille de codes viraux utilisant un même moteur de mutation. On a :

$$p(X = c|Y = d) = \frac{p(X = c)p(Y = d|X = c)}{p(Y = d)}$$

et

$$p(X = \neg c|Y = d) = \frac{p(X = \neg c)p(Y = d|X = \neg c)}{p(Y = d)}$$

donc :

$$\frac{p(X = c|Y = d)}{p(X = \neg c|Y = d)} = \frac{p(X = c)}{p(X = \neg c)} \frac{p(Y = d|X = c)}{p(Y = d|X = \neg c)} = \frac{p(X = c)}{p(X = \neg c)} \prod_{j=1}^{|\mathcal{W}|} \left( \frac{p(W = w_j|X = c)}{p(W = w_j|X = \neg c)} \right)^{N_j}$$

En prenant le logarithme népérien, on obtient :

$$\ln \left( \frac{p(X = c|Y = d)}{p(X = \neg c|Y = d)} \right) = \ln \left( \frac{p(X = c)}{p(X = \neg c)} \right) + \sum_{j=1}^{|\mathcal{W}|} N_j \ln \left( \frac{p(W = w_j|X = c)}{p(W = w_j|X = \neg c)} \right).$$

La règle de décision est alors la suivante pour le fichier $d$ en cours d'analyse : si le ratio des log-vraisemblance :

$$\ln \left( \frac{p(X = c|Y = d)}{p(X = \neg c|Y = d)} \right) > 0\,,$$

alors on rejette l'hypothèse nulle $\mathcal{H}_0$ : le fichier est probablement infecté par un virus de la famille $c$. Dans le cas contraire, on ne peut pas rejetter l'hypothèse nulle : le fichier est soit sain, soit infecté par un virus d'une autre famille virale.

Cette méthodologie de test est très simple, mais induit en pratique un taux de fausse alarme trop élevé. Pour y remédier, une première étape a pour objectif d'étalonner le détecteur sur des données de test. Soit une population virale $\mathcal{V}$ donnée, et un ensemble de programmes sains $\mathcal{B}$. Nous formons plusieurs sous-ensembles à partir de $(\mathcal{V}, \mathcal{B})$ :

- les sous-ensembles $(\mathcal{V}_1, \mathcal{B}_1)$, avec $\mathcal{V}_1 \subset \mathcal{V}$ et $\mathcal{B}_1 \subset \mathcal{B}$ constituent les données d'apprentissage ;
- les sous-ensembles $(\mathcal{V}_2, \mathcal{B}_2)$, avec $\mathcal{V}_2 \subset \mathcal{V}$ et $\mathcal{B}_2 \subset \mathcal{B}$ constituent les données de test.

Plaçons nous dans le cas où notre modèle est formé uniquement sur des données d'apprentissage virales. Nous utilisons le sous-ensemble $\mathcal{V}_1$ pour estimer les probabilités $p(X = c)$ et $p(W_i = w_i \,|\, X = c)$. Nous utilisons ensuite notre modèle pour calculer la probabilité qu'un programme de l'ensemble $\mathcal{V}_2 \cup \mathcal{B}_2$ appartienne à la même famille de programmes que l'ensemble d'apprentissage. Ayant calculé ces vraisemblances, nous pouvons déterminer empiriquement un seuil $S$ à partir duquel la classification opère sans erreur.
Nous pouvons enfin former deux sous-ensembles supplémentaires $\mathcal{V}_3 \subset \mathcal{V}$ et $\mathcal{B}_3 \subset \mathcal{B}$ qui vont nous permettre de calculer les erreurs de type I et II qui caractérisent notre détecteur probabiliste.

L'intérêt de cette approche est que chaque modèle peut être associé à une famille virale, résultant par exemple de l'utilisation d'un kit générateur de virus. À la différence d'une approche par recherche de signatures, un unique modèle convenablement étalonné peut s'appliquer à toutes les variantes d'une famille.

Le choix d'un bon dictionnaire est primordial dans tout problème de classification. Dans le cas de l'analyse spectrale, un jeu d'instruction adapté doit être construit. En effet, selon le nombre d'instructions qui le composent, les resultats peuvent varier très significativement pour une même méthode de classification et des performances différentes peuvent être observés entre différentes méthodes. La réduction de la taille du dictionnaire a en outre un impact sur les performances. La théorie de l'information fournit des outils qui sont traditionnellement utilisés pour réduire la taille d'un dictionnaire en ne conservant que les mots ayant un pouvoir discriminant suffisant. Nous verrons que la théorie de l'information, à travers les concepts d'entropie et d'information mutuelle, nous permet de qualifier un modèle de classification, en terme de précision.
L'entropie et l'entropie conditionnelle d'une variable aléatoire $X$ sont définies de la manière suivante :

**Définition 7.4.2 (Entropie [SW49])** *L'entropie $H(X)$ est une mesure de l'incertitude portant sur la variable aléatoire $X$ :*

$$H(X) = -\sum_x P_X(x) \log P_X(x).$$

*L'entropie conditionnelle $H(X|Y)$ représente l'incertitude sur la variable $X$ après observation de la variable aléatoire $Y$ :*

$$H(X|Y) = E_{P_Y}\left[-E_{P_{X|Y}}[\log P_{X|Y}]\right].$$

L'information mutuelle de deux variables aléatoires $X$ et $Y$ est définie de la manière suivante :

**Définition 7.4.3 (Information mutuelle [SW49])** *L'information mutuelle $I(X,Y)$ représente la réduction d'incertitude sur la variable aléatoire $X$ après observation de la variable aléatoire $X$ :*

$$\begin{aligned} I(X,Y) &= E_{P_{X,Y}}\left[\log \frac{P_{X,Y}}{P_X P_Y}\right] \\ &= \sum_{x,y} P_{X,Y}(x,y) \log \frac{P_{X,Y}(x,y)}{P_X(x)P_Y(y)} \\ &= H(X) - H(X|Y). \end{aligned}$$

*L'information mutuelle peut être vue comme un taux de transmission à travers un canal bruité :*

$$X \longrightarrow \boxed{Channel} \longrightarrow Y$$

L'information mutuelle $I(X,Y)$ est aussi la distance KL [CT91] entre la distribution jointe, $P_{X,Y}$, et le produit indépendant des distributions, $p_X p_Y$. Par conséquent, une autre façon de voir l'information mutuelle est qu'il s'agit de la distance entre les distributions corrélées et non-corrélées de $X$ et $Y$ :

$$I(X,Y) = d_{KL}(p_{X,Y}||p_X p_Y) = \sum_{x,y} p_X(x)p_Y(y) \log \frac{p_X(x)p_Y(y)}{p_{X,Y}(x,y)}.$$

Parceque la distance KL vaut zéro si et seulement si $p_{X,Y} = p_X p_Y$, il s'ensuit que l'information mutuelle capture toutes les dépendances entre variables aléatoires, pas seulement par exemple une dépendance du second ordre, telle que capturée par la covariance.

L'information mutuelle permet de sélectionner les instructions porteuses d'un pouvoir discriminant en choisissant les mots les plus caractéristiques d'une catégorie de programme. Pour chaque instruction, on regarde s'il est corrélé avec les classes. Si c'est le cas, cela signifie qu'il est porteur de sens et donc qu'il faut le conserver. Le calcul suivant permet, pour chaque instruction $W = w_i$, de vérifier si l'entropie de la distribution des classes diminue bien suffisamment lorsque l'on sait que l'instruction $w_i$ est présente :

$$\begin{aligned} I(X,W_i) &= H(X) - H(X|W_i) \\ &= -\sum_{j=1}^{|\mathcal{C}|} p(X=c_j)\log p(X=c_j) + \sum_{B_i\in\{0,1\}} p(B_i) \sum_{j=1}^{|\mathcal{C}|} p(X=c_j|B_i)\log p(X=c_j|B_i) \\ &= \sum_{B_i\in\{0,1\}} \sum_{j=1}^{|\mathcal{C}|} p(X=c_j,B_i)\log \frac{p(X=c_j,B_i)}{p(X=c_j)p(B_i)} \end{aligned}$$

Nous pouvons définir, dans le contexte de l'analyse spectrale, la précision d'une fonction de détection fondée sur un modèle NB de la manière suivante [RHJ01] :

**Définition 7.4.4 (Précision d'un modèle NB)** *Notons $p_Y = p(Y_0,\ldots,Y_{T-1})$ et $p_Y^{NB} = p(Y_0,\ldots,Y_{T-1})$ la vraie distribution et la distribution sous un modèle NB des variables observées $Y$. La précision d'un modèle NB peut être définie en utilisant la distance de Kullback-Leibler $d_{KL}$ entre les deux distributions $p_Y$ et $p_Y^{NB}$ :*

$$d_{KL}(p_Y||p_Y^{NB}) = \sum_y p_Y(y)\log\frac{p_Y(y)}{p_Y^{NB}(y)}.$$

*Plus cette dernière est proche de 0, plus le modèle NB est précis.*

*Dans le cas général, nous considérons des modèles NB qui ont une distance KL différente de zéro :*

$$d_{KL}(p_Y||p_Y^{NB}) = \varepsilon > 0.$$

La précision d'un modèle NB rend compte de l'entropie relative entre la distribution réelle d'une famille virale et le modèle de détection construit à partir des données d'apprentissage.

Nous pouvons donc proposer la définition suivante de la simulabilité d'une fonction de détection fondée sur un modèle NB :

**Proposition 7 (Simulabilité faible de l'analyse spectrale fondée sur des modèles NB)** *Pour simuler faiblement l'analyse spectrale fondée sur des modèles NB, le moteur de mutation doit modifier aléatoirement les instructions $Y$ de telle sorte que la distance de Kullback-Leibler $d_{KL}$ entre les distributions $p_Y$ et $p_Y^{\lambda_i^{NB}}$ est plus grande que $\varepsilon$, pour chacun des modèles $\lambda_i^{NB}$ utilisés par la fonction de détection.*

## 7.5 Analyse spectrale par modèle Markovien

Nous avons vu dans la section précédente une approche bayésienne élémentaire pour générer une règle de décision sur la base de données d'apprentissage. Nous présentons à présent une approche fondée sur le modèle de Markov caché, qui peut être considéré comme le plus simple des réseaux bayésiens dynamiques [Gha98]. Nous présentons son application dans le contexte de l'analyse spectrale. Un test statistique bayésien peut être formulé sur la base de cette méthode de classification. Nous étudions la simulabilité faible d'un tel test.

### 7.5.1 Autres applications du modèle de Markov caché

Les modèles de Markov cachées peuvent être utilisés en génétique ([Mur97], [Pla01]), en reconnaissance de la parole, texte, langue ([Sta04], [Rab89], [Tsa04], [LC96], [Bil04], [Bil06]), pour la désobfuscation de SPAM ([LN05a], [LJC07]), la désobfuscation de code viral ([WS06], [Won06]), ou la cryptanalyse d'algorithmes de chiffrement ([KW03], [SWT01]).

Avant de présenter l'utilisation qui peut être faite des modèles Markoviens cachés dans le contexte de l'analyse spectrale de code viral, nous présentons quelques-unes des utilisations réussies en biologie, en cryptographie, et en traitement de textes.

#### Décodage des molécules d'ADN

Le génome d'un organisme est formé de molécules d'ADN. Une molécule d'ADN est constituée de deux brins appariés en une double hélice. Chaque brin est constitué d'une séquence des bases `a`, `c`, `t`, `g` (adénine, cytosine, thymine et guanin). Il existe un lien entre les bases portées par les deux brins : `a` et `c` sont toujours appariées. Les deux brins sont orientés. Seule une partie de cette importante quantité d'information code effectivement pour des protéines. On appelle ces séquences codantes, des gènes. Nous nous intéressons ici à la localisation de ces séquences codantes. Les bases sont lues par groupes de trois bases appelés codons. Ces codons sont traduits en acides aminés, les constituants de base des protéines. Il existe 20 acides aminés différents.

Les procaryotes sont des organismes assez élémentaires (bactéries) partageant des caractéristiques communes au niveau de leur génome. Chez ces organismes, un gène commence par un des codons "start" (`atg`, `gtg`, `ttg`) et se termine par un des codons "stop" (`taa`, `tag`, `tga`).

Les eucaryotes sont des organismes généralement plus évolués. Chez ces organismes, la structure d'un gène est plus complexe. Un gène est toujours délimité par un codon "start" et un codon "stop", mais comporte des sous-séquences non codantes, appelées introns, délimitées à l'intérieur d'un gène par les couples de bases `ag` et `gc`. Cette particularité complique la tâche, dans la mesure où un intron peut contenir des codons "stop".

Les modèles classiquement utilisés par les biologistes sont les suivants :

- M1M0 : le processus caché est une chaîne de Markov d'ordre 1. Conditionnellement au processus caché, le processus observé est une chaîne de Markov d'ordre 0, i.e., conditionnellement au processus caché, les bases sont générées de manière indépendante.

- M1M1 : le processus caché est une chaîne de Markov d'ordre 1. Conditionnellement au processus caché, le processus observé est une chaîne de Markov d'ordre 1.
- M1M2 : le processus caché est une chaîne de Markov d'ordre 1. Conditionnellement au processus caché, le processus observé est une chaîne de Markov d'ordre 2. Ce modèle suppose une dépendance markovienne d'ordre 2 entre les observations conditionnellement aux états cachés.

L'utilisation de modèles de Markov caché d'ordre 0 (modèle M1M0), d'ordre 1 (modèle M1M1) ou d'ordre 2 (modèle M1M2) permettent de localiser les zones dans le génome qui ont des propriété remarquables, relativement au modèle choisi (comme par exemple de longues séquences sans codon "stop" dans le cas d'un modèle M1M0), puis d'extraire les séquences codantes [Mur97, Pla01].

### Attaques par canaux auxiliaires

L'idée fondamentale des attaques par canaux auxiliaires (également appelées attaques par canaux cachés), en cryptologie, est de chercher à obtenir de l'information sur des données secrètes, telles que des clés de chiffrement ou de signature, en observant les variations de consommation (nombre de cycles CPU par exemple) d'une application cryptographique. Après une indispensable étape d'analyse théorique du produit, une telle attaque se décompose fondamentalement en deux étapes, la première consistant à acquérir des courbes de consommation lors du fonctionnement de l'application et la seconde visant à extraire de l'information par des moyens algorithmiques.

Pour se prémunir de telles attaques, une idée simple consiste à introduire des données aléatoires dans le calcul. Considérons par exemple l'algorithme suivant, de multiplication scalaire d'une courbe elliptique (`m` et `p` sont des points de la courbe elliptique, `k=(k[N-1]...k[0])` est la clé) :

```
int mult(int *k, m)
{
       q=m;
       p=0;
       for(int i=0; i<N; i++){
          if (k[i]==1)
               p+=q;
          q*=2;
       }
       return(p);
}
```

Afin de rendre difficile les attaques par canaux auxiliaires fondées sur une différence de consommation entre les opérations

- addition ;
- addition et multiplication ;

nous considérons la version suivante du même algorithme :

```
int mult(int *k, m)
{
       q=m;
       p=0;
       for(int i=0; i<N; i++){
          r=p;
          b=rand_bit();
          if (k[i]==0){
               if (b==1)
                    r+=q;
          } else {
               p+=q;
          }
          q*=2;
       }
```

```
        return(p);
}
```

Nous voyons que de manière aléatoire, des calculs inutiles sont ajoutés. Supposons maintenant que l'attaquant observe le canal auxiliaire et est en mesure de distinguer l'opération `p+=q ;` de l'opération `p+=q ; q*=2 ;`. L'introduction de l'opération supplémentaire `r+=q ;` avec probabilié 1/2 va rendre l'attaque plus difficile à mettre en oeuvre. Le principe de l'utilisation de code supplémentaire aléatoire est de rendre aléatoires les données observées sur le canal auxiliaire, de manière à dissocier les informations fournies par le canal auxiliaire de la suite des états internes de l'algorithme à protéger.

L'utilisation d'un modèle de Markov caché adapté [KW03] permet de retrouver la séquence d'états la plus probable compte-tenu des observations, puis de retrouver la clé.

### 7.5.2 Rappels sur les chaînes de Markov

**Définition 7.5.1** *(Chaîne de Markov à temps discret [Cin75]). Par définition, une chaîne de Markov $\{X_n, n \in E\}$ possède la propriété suivante :*

$$P(X_n = i_n \,|X_0 = i_0, \ldots, X_{n-1} = i_{n-1}) = P(X_n = i_n \,|X_{n-1} = i_{n-1})$$

Cette propriété peut s'interpréter de la façon suivante : sachant l'état présent $i_{n-1}$, l'état futur $i_n$ est indépendant des états passés $i_j$, $j < n-1$.

**Définition 7.5.2** *(Chaine de Markov homogène [Cin75]). Soit respectivement : $p_j(k) = P(X_k = j), \forall j \in E$, et $p_{ij}(m,k) = P(X_k = j|X_m = i), \forall m, k : 0 \leq m \leq k$, la distribution de probabilité de la v.a. (variable aléatoire) $X_k$ et la distribution conditionnelle de $X_k$ sachant $X_m$. On dit qu'une chaîne est homogène si l'on a*

$$p_{ij}(m,k) = p_{ij}(k-l, m-l), \forall l = 0, \ldots, m.$$

Dans ce cas, $p_{ij}(m,k)$ ne dépend que de $i$, $j$ et $(k-m)$. On peut donc poser :

$$p_{ij}(n) = p_{ij}(m+n, m), \forall m.$$

**Définition 7.5.3** *(Probabilité de transition d'une chaîne de Markov homogène [Cin75]). $p_{ij}(n)$ s'appelle la probabilité de transition à $n$ étapes. La probabilité de transition à 1 étape se note simplement $p_{ij} = p_{ij}(1)$.*

Par convention, définissons :

$$p_{ij}(0) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{sinon.} \end{cases}$$

A partir de maintenant, nous ne considérons que des chaines de Markov homogènes. Notons que la distribution de probabilité conjointe $P(X_0 = i_0, \ldots, X_n = i_n)$ peut s'écrire à l'aide des probabilités de transition à une étape :

$$\begin{aligned} & P(X_0 = i_0, \ldots, X_n = i_n) \\ = \;& P(X_0 = i_0, \ldots, X_{n-1} = i_{n-1}) P(X_n = i_n \,|X_0 = i_0, \ldots, X_{n-1} = i_{n-1}) \\ = \;& P(X_0 = i_0, \ldots, X_{n-1} = i_{n-1}) P(X_n = i_n \,|X_{n-1} = i_{n-1}) \\ = \;& P(X_0 = i_0, \ldots, X_{n-1} = i_{n-1}) p_{i_{n-1} i_n} \end{aligned}$$

et donc par induction

$$P(X_0 = i_0, \ldots, X_n = i_n) = p_{i_0}(0) p_{i_0 i_1} \ldots p_{i_{n-1} i_n}$$

La distribution initiale de $X_0$ est notée à l'aide du vecteur ligne :

$$\mathbf{p}(0) = (p_0(0), p_1(0), \ldots)$$

Notons $P$ la matrice formée des probabilités de transition à une étape $p_{ij}$ :

$$P = (p_{ij})_{i\geq 0, j\geq 0}$$

Comme $\forall i, j$, $0 \leq p_{ij} \leq 1$ et comme $\sum_{j\geq 0} p_{ij} = 1$, $i \geq 0$, la matrice $P$ est une matrice stochastique. Ainsi, le processus est complètement déterminé si on connaît la distribution initiale $\mathbf{p}(0)$ et la matrice de transition $P$.

Notons $f_{jk}(n)$, la probabilité que la première entrée dans l'état $k$ se produise à l'étape $n$ sachant que la chaîne se trouve initialement dans l'état $j$ (i.e. sachant que $X_0 = j$). Autement dit, si la v.a. (variable aléatoire) $T_k$ désigne la date de la première entrée dans l'état $k$, $f_{jk}(.)$ est la distribution conditionnelle donnée par :

$$f_{jk}(n) = P(T_k = n \,|X_0 = j).$$

Posons :

$$f_{jk} = \sum_{n\geq 1} f_{jk}(n),$$

$f_{jk}$ est donc la probabilité conditionnelle que le processus passe par l'état $k$ sachant qu'il est initialement dans l'état $j$ ; c'est encore la probabilité que $T_k$ prenne une valeur finie :

$$f_{jk} = P(T_k < \infty \,|X_0 = j).$$

Compte tenu de la définition de $f_{jk}(n)$, on a la relation suivante :

$$f_{jk}(n) = \begin{cases} p_{jk} & \text{si } n = 1 \\ \sum_{l\neq k} p_{jl} f_{lk}(n-1) & \text{si } n > 1 \end{cases}$$

Définissons maintenant la v.a. $N_j$ comme le nombre de visites à l'état $j$ ; i.e. pour une trajectoire $\omega$, on a :

$$N_j(\omega) = \sum_{n\geq 0} \mathbf{1}_{[X_n(\omega)=j]}$$

Partant de $j$, la probabilité que $N_j = n$ est donnée par :

$$P_j(N_j = n) = P(N_j = n \,|X_0 = j) = f_{jj}^{n-1}(1 - f_{jj}),\ n \geq 1.$$

Notons que dans ce cas, $N_j$ ne peut pas être nulle puisque $X_0 = j$.
Partant de $i$, la distribution de $N_j$ est donnée par :

$$P_i(N_j = n) = \begin{cases} 1 - f_{ij} & \text{si } n = 0 \\ f_{ij} f_{jj}^{n-1}(1 - f_{jj}) & \text{si } n > 0 \end{cases}$$

Considérons la probabilité que $N_j$ prenne une valeur finie :

$$P(N_j < \infty) = \sum_{n\geq 0} P_j(N_j = n)$$

On a le théorème suivant :

**Théorème 7.5.1** *(Probabilité du nombre de visites d'un état [Cin75]).*

$$P_j(N_j < \infty) = \begin{cases} 1 & si\ f_{jj} < 1 \\ 0 & si\ f_{jj} = 1 \end{cases}$$

**Preuve** : Si $f_{jj} = 1$, alors $P_j(N_j = n) = f_{jj}^{n-1}(1-f_{jj}) = 0$, $\forall n \geq 1$. Donc $P_j(N_j < \infty) = \sum_{n\geq 0} P_j(N_j = n) = 0$. □

Partant d'un état $j$, le nombre de visite à cet état est infini si et seulement si $f_{jj} = 1$. Puisque dans ce cas le retour à l'état $j$ est certain, on peut définir :

$$m_j = \sum_{n\geq 1} n f_{jj}(n)$$

Si $m_j < \infty$, $m_j$ est l'espérance du temps de retour. Les définitions suivantes introduisent les différents types des états de $E$.

**Définition 7.5.4** *(Etat accessible [Cin75]). Un état $j$ est accessible à partir d'un état $i$ si*

$$\exists n_{n\in\mathbb{N}}[p_{ij}(n) > 0].$$

*Cette définition est équivalente à : partant de $i$, il est possible que le processus prenne ultérieurement l'état $j$. Notons que l'état $i$ est accessible à partir de l'état $i$ car $p_{ii}(0) = P(X_0 = i|X_0 = i) = 1$.*

**Définition 7.5.5** *(Etats communiquants [Cin75]). Les états $i$ et $j$ communiquent si $i$ et $j$ sont mutuellement accessibles.*

Cette relation de communication, que nous noterons $\sim$ est une relation d'équivalence :
- $i \sim i$
- $(i \sim j) \Longrightarrow (j \sim i)$
- $(i \sim j)$ et $(j \sim k) \Longrightarrow (i \sim k)$

**Définition 7.5.6** *(Etat récurrent, transitoire [Cin75]). L'état $j$ est dit récurrent si $f_{jj} = 1$ ; Il est dit transitoire si $f_{jj} < 1$. Un état récurrent $j$ est dit récurrent non nul (ou positif) si $m_j < \infty$ ; Il est dit récurrent nul si $m_j = \infty$.*

Compte tenu du théorème 7.5.1, on voit que s'il est atteint, un état récurrent est visité une infinité de fois alors qu'un état transitoire ne peut être visité qu'un nombre fini de fois. On voit aussi qu'une chaine de Markov à nombre fini d'états ne peut avoir tous ses états transitoires.

**Définition 7.5.7** *(Etat absorbant). L'état $i$ est dit absorbant si $p_{ii} = 1$.*

Cela signifie qu'une fois que le système est dans un état absorbant, il le reste.

**Définition 7.5.8** *(Etat périodique). L'état $i$ est de périodicité $d_i$ si $d_i$ est le pgcd des entiers*

$$\{n \in \mathbb{N}\,|p_{ii}(n) > 0\}$$

**Définition 7.5.9** *(Etat apériodique). Si $d_i = 1$, l'état $i$ est dit apériodique.*

**Théorème 7.5.2** *(Classe des états récurrent). Si $j$ est un état récurrent et si $i \sim j$, alors $i$ est aussi un état récurrent.*

Ainsi la récurrence est une propriété de classe. Il en est de même pour les états transitoires.

**Corrolaire 1** *(Classe des états transitoires). Si $j$ est un état transitoire et si $i \sim j$, alors $i$ est aussi un état transitoire.*

On peut encore préciser les résultats précédents à l'aide du théorème suivant.

**Théorème 7.5.3** *[Cin75] Soit $\mathcal{E}_k$ une classe d'équivalence de l'ensemble $\mathcal{E}$ des états d'une chaîne de Markov. Alors tous les états de $\mathcal{E}_k$ sont tous soit transitoires, soit récurrents nuls, soit récurrents non nuls. Ils sont tous soit apériodiques, soit périodique de même période $d$.*

La récurrence positive est donc une propriété de classe, de même que la périodicité.

**Définition 7.5.10** *(Chaîne de Markov irréductible [Cin75]). Une chaîne de Markov est dite irréductible si ses états constituent une classe unique, i.e., si tous ses états communiquent entre eux.*

Comme les états d'une chaîne de Markov irréductible constituent une classe unique, ils sont tous du même type et on a le corollaire suivant :

**Corrolaire 2** *(Etats d'une chaîne de Markov irréductible [Cin75]). Etant donnée une chaîne de Markov irréductible $X$, tous ses états sont tous soit transitoires, soit récurrents nuls, soit récurrents non nuls. Ils sont tous soit apériodiques, soit périodiques de même période $d$.*

Ainsi, même si la chaîne est irréductible, les états peuvent être récurrents nuls, voire transitoires, si le nombre d'états est infini. Mais si le nombre d'état est fini, nous allons voir que les états ne peuvent être que récurrents non nuls.

**Théorème 7.5.4** *(Chaîne de Markov irréductible à nombre fini d'états [Cin75]). Une chaîne de Markov irréductible à nombre fini d'états n'a que des états récurrents non nuls.*

**Définition 7.5.11** *(Etat ergodique [Cin75]). Un état récurrent non nul et apériodique dont la probabilité asymptotique est indépendante des conditions initiales, est dit ergodique.*

**Définition 7.5.12** *(Chaîne de Markov ergodique). Une chaîne de Markov irréductible dont tous les états sont récurrents non nuls et apériodiques (et admettent donc des probabilités asymptotiques indépendantes des conditions initiales) est dite ergodique.*

### 7.5.3 Modèle de Markov caché

**Définition 7.5.13** *(Modèle de Markov caché [Cin75]). Une modèle de Markov caché est défini par $\lambda = (A, B, \mathbf{a}(0))$ où $A = (a_{ij})_{0\leq i\leq N-1,\, 0\leq j\leq N-1}$ est la matrice de transition d'une chaîne de Markov (d'ordre 1) avec $a_{ij} = P(X_{n+1} = x_j \,|X_n = x_i)$ ; $B = (b_{jk})_{0\leq j\leq N-1,\, 0\leq k\leq M-1}$ est une matrice $N \times M$ avec $b_{jk} = b_j(k) = P(Y_n = k \,|X_n = x_j)$ et $\mathbf{a}(0)$ est la distribution initiale de $X_0$.*

Considérons la chaîne cachée de longueur $T$ : $X = (X_0, \ldots, X_{T-1})$ et les observations correspondantes $Y = (Y_0, \ldots, Y_{T-1})$. On a :

$$P(X = (x_0, \ldots, x_{T-1})) = a_{x_0}(0)b_{x_0}(0)a_{x_0x_1}b_{x_1}(1)\ldots a_{x_{T-2}x_{T-1}}b_{x_{T-1}}(T-1)$$

Nous pouvons identifier trois problèmes fondamentaux qui doivent être résolus de telle manière que le modèle de Markov caché puisse avoir des applications concrètes. Cette caractérisation est due à J. Ferguson, qui l'a introduit lors de lectures dans les laboratoires Bell. Ces trois problèmes sont les suivants [Rab89] :

Problème a) Etant donné un tel modèle $\lambda = (A, B, \mathbf{a}(0))$, et une séquence d'observations $Y$, nous pouvons chercher à déterminer la vraisemblance de la séquence observée, étant donné le modèle.

Problème b) Etant donné un tel modèle $\lambda = (A, B, \mathbf{a}(0))$, et une séquence d'observations $Y$, nous pouvons chercher à retrouver la partie cachée du modèle. Il s'agit donc de trouver une séquence d'états $X$ qui soit optimale pour le modèle sous-jacent.

Problème c) Etant donné une séquence d'observations $Y$ et les dimensions $N$ et $M$ (donnant le nombre d'états de la chaîne de Markov $X$ et la taille de la séquence observée), nous pouvons chercher à trouver le modèle $\lambda = (A, B, \mathbf{a}(0))$ qui maximise la probabilité de $Y$. Il s'agit donc de fournir au modèle des données d'apprentissage, de manière à estimer les paramètres du modèle.

**Problème a** : Etant donné un modèle $\lambda = (A, B, \mathbf{a}(0))$, et une séquence d'observations $Y$, nous cherchons à déterminer la vraisemblance de la séquence observée, étant donné le modèle.

Il s'agit donc de déterminer $P(Y\,|\lambda)$. Par définition de la matrice $B$, on a :

$$P(Y\,|X, \lambda) = b_{x_0}(0)b_{x_1}(1)\ldots b_{x_{T-1}}(T-1)$$

et par définition de la distribution initiale $\mathbf{a}(0)$ et de la matrice de transition de la chaîne de Markov, on a :

$$P(X\,|\lambda) = a_{x_0}(0)a_{x_0x_1}\ldots a_{x_{T-2}x_{T-1}}$$

Puisque

$$P(Y, X\,|\lambda) = \frac{P(Y \cap X \cap \lambda)}{P(\lambda)}$$

et

$$P(Y\,|X, \lambda) = \frac{P(Y \cap X \cap \lambda)}{P(X \cap \lambda)}$$

et

$$P(X\,|\lambda) = \frac{P(X \cap \lambda)}{P(\lambda)}$$

on a

$$P(Y, X\,|\lambda) = \frac{P(Y \cap X \cap \lambda)}{P(X \cap \lambda)} \frac{P(X \cap \lambda)}{P(\lambda)} = P(Y\,|X, \lambda) P(X\,|\lambda).$$

En sommant sur l'ensemble des séquences d'états possibles, on obtient :

$$\begin{aligned} P(Y\,|\lambda) &= \sum_X P(Y, X\,|\lambda) \\ &= \sum_X P(Y\,|X, \lambda) P(X\,|\lambda) \\ &= \sum_X a_{x_0}(0) b_{x_0}(0) a_{x_0 x_1} b_{x_1}(1) \ldots a_{x_{T-2} x_{T-1}} b_{x_{T-1}}(T-1) \end{aligned}$$

Cependant, le calcul direct de $P(Y\,|\lambda)$ est très coûteux, puisqu'il requiert environ $2TN^T$ multiplications. L'algorithme 3 permet d'effectuer ce même calcul avec $N^2T$ multiplications. Il se fonde sur l'observation suivante :

$$P(Y\,|\lambda) = \sum_{i=0}^{N-1} P(Y_0, \ldots, Y_{T-1}, X_{T-1} = x_i\,|\lambda).$$

Définissons $\alpha_t(i) = P(Y_0, \ldots, Y_{T-1}, X_t = x_i\,|\lambda)$, i.e., la probabilité de l'observation partielle jusqu'à l'instant $t$, où la chaîne de Markov sous-jacente est dans l'état $x_i$. Le calcul de $\alpha_t(i)$ peut être fait récursivement.

**Pour** $i$ **de** $0$ **à** $N-1$ **faire**
  $\alpha_0(i) = a_i(0) b_i(0)$
**Fin Pour**
**Pour** $t$ **de** $1$ **à** $T-1$ **faire**
  **Pour** $i$ **de** $0$ **à** $N-1$ **faire**
    $\alpha_t(i) = \left[ \sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} \right] b_i(t)$
  **Fin Pour**
**Fin Pour**
**Retourner** $P(Y\,|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$

Algorithme 3: Forward [Cin75]

**Problème b** : Etant donné un modèle de Markov caché $\lambda = (A, B, \mathbf{a}(0))$, et une séquence d'observations $Y$, nous cherchons à retrouver la partie cachée du modèle. Il s'agit donc de trouver une séquence d'états $X$ qui soit optimale pour le modèle sous-jacent.

Pour $0 \leq t \leq T-2$, $0 \leq i \leq N-1$, on définit

$$\gamma_t(i) = P(X_t = x_i\,|Y, \lambda)$$

L'état optimal est donné par

$$x_t = \underset{i=0,\ldots,N-1}{\operatorname{argmax}} \{\gamma_t(i)\}$$

Définissons la probabilité $\beta_t(i)$ de l'observation partielle à partir de l'instant $t$, où la chaîne de Markov sous-jacente est dans l'état $x_i$ :

$$\beta_t(i) = P(Y_{t+1}, \ldots, Y_{T-1} \,|X_t = x_i, \lambda),\ 0 \leq t \leq T-1,\ 0 \leq i \leq N-1$$

$\alpha_t(i)$ donne la probabilité de l'observation partielle jusqu'à l'intant $t$. Or

$$P(Y_0, \ldots, Y_t, X_t = x_i \,|\lambda) P(Y_{t+1}, \ldots, Y_{T-1} \,|X_t = x_i, \lambda) = P(X_t = x_i \,|Y, \lambda) P(Y \,|\lambda)$$

Donc

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(Y \,|\lambda)}$$

L'algorithme 4 permet de calculer efficacement $\beta_t(i)$ et l'algorithme 3 permet de calculer efficacement $\alpha_t(i)$ et $P(Y \,|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$.

**Pour** $i$ **de** $0$ **à** $N-1$ **faire**
  $\beta_{T-1}(i) = 1$
**Fin Pour**
**Pour** $t$ **de** $T-2$ **à** $0$ **faire**
  **Pour** $i$ **de** $0$ **à** $N-1$ **faire**
    $\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(t+1) \beta_{t+1}(j)$
  **Fin Pour**
**Fin Pour**

Algorithme 4: Backward [Cin75]

**Problème c** : Etant donné une séquence d'observations $Y$ et les dimensions $N$ et $M$ (donnant le nombre d'états de la chaîne de Markov $X$ et la taille de la séquence observée), nous cherchons à trouver le modèle $\lambda = (A, B, \mathbf{a}(0))$ qui maximise la probabilité de $Y$. Il s'agit donc de fournir au modèle des données d'apprentissage, de manière à estimer les paramètres du modèle.

Définissons la probabilité $\gamma_t(i,j)$ d'être dans l'état $x_i$ à l'instant $t$ et d'effectuer une transition vers l'état $x_j$ à l'instant $t+1$

$$\gamma_t(i,j) = P(X_t = x_i, X_{t+1} = x_j \,|Y, \lambda)$$

On démontre facilement que

$$\gamma_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(t+1) \beta_{t+1}(j)}{P(Y \,|\lambda)}$$

On a

$$P(X_t = x_i \,|Y, \lambda) = \sum_{j=0}^{N-1} P(X_t = x_i, X_{t+1} = x_j \,|Y, \lambda)$$

i.e.,

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i,j)$$

L'algorithme 5 permet de ré-estimer itérativement le modèle $\lambda = (A, B, \mathbf{a}(0))$ de manière à maximiser la probabilité de $Y$.

**En entrée** : $\lambda = (A, B, \mathbf{a}(0))$
**Pour** $i$ **de** $0$ **à** $N-1$ **faire**
  $a_i(0) = \gamma_0(i)$
**Fin Pour**
**Pour** $i$ **de** $0$ **à** $N-1$ **faire**
  **Pour** $j$ **de** $0$ **à** $N-1$ **faire**
    $a_{ij} = \sum_{t=0}^{T-2} \gamma_t(i,j) / \sum_{t=0}^{T-2} \gamma_t(i)$
  **Fin Pour**
**Fin Pour**
**Pour** $j$ **de** $0$ **à** $N-1$ **faire**
  **Pour** $k$ **de** $0$ **à** $M-1$ **faire**
    $b_j(k) = \sum_{0 \leq t \leq T-2 \,|\, X_t = k} \gamma_t(j) / \sum_{t=0}^{T-2} \gamma_t(j)$
  **Fin Pour**
**Fin Pour**
**Retourner** $\lambda = (A, B, \mathbf{a}(0))$

Algorithme 5: Baum-Welch : ré-estimation du modèle $\lambda = (A, B, \mathbf{a}(0))$ [Cin75]

La procédure de ré-ajustement itératif du modèle $\lambda = (A, B, \mathbf{a}(0))$ est la suivante :

1. Initialiser le modèle $\lambda = (A, B, \mathbf{a}(0))$ de manière aléatoire, en prenant $a_i(0) \simeq 1/N$, $a_{ij} \simeq 1/N$ et $b_j(k) \simeq 1/M$.
2. Calculer $\alpha_t(i)$ (en utilisant l'algorithme 3), $\beta_t(i)$ (en utilisant l'algorithme 4), $\gamma_t(i,j)$ et $\gamma_t(i)$.
3. Ré-estimer le modèle $\lambda = (A, B, \mathbf{a}(0))$.
4. Si $P(Y|\lambda)$ croît suffisamment (c'est à dire augmente de $\delta \geq \Delta$, où $\Delta$ est un seuil prédéterminé) ou si le nombre d'itérations dépasse un certain seuil, recommencer à l'étape 2.

Nous voyons donc qu'il est possible de former un modèle de Markov caché $\lambda = (A, B, \mathbf{a}(0))$ sur les données d'apprentissage (constituées des spectres d'instructions d'une famille de programmes viraux) de manière à estimer les paramètres du modèle, en utilisant l'algorithme 5 de Baum-Welsh [Cin75].
Ensuite, étant donné un tel modèle, et une séquence d'instructions extraites d'un fichier $F$, nous pouvons calculer la vraisemblance de la séquence observée, étant donné le modèle, en utilisant l'algorithme Forward 3.
Dans [Rab89], un HMM est comparé à un ensemble d'urnes contenant chacune un certain nombre de balles de couleur. L'échantillonnage d'un tel modèle se fait alors en choisissant à chaque étape une nouvelle urne, ce choix ne dépendant que de l'urne précédemment choisie, puis en choisissant en remplacement une balle de cette urne. La séquence des choix d'urne n'est pas rendu public (cette séquence est cachée). En revanche, les choix de balles sont connus (ils sont observés).
Dans le contexte de la détection virale, nous observons une séquence d'octets ou d'instructions assembleur, et nous nous intéressons non seulement à la manière dont le HMM génère ces deux séquences, mais également à la distribution des séquences produites par un HMM et sur la manière de les comparer. Lors de la phase d'apprentissage du modèle, l'idée est d'utiliser un ensemble de programmes viraux générés à partir d'un même moteur de mutation : typiquement, un ensemble de variantes produites par un virus polymorphe ou un ensembles de programmes viraux produits à partir d'un kit de génération de virus. Chaque HMM est donc destinée à stocker une information permettant de résumer ou de compresser les caractéristique d'un moteur de mutation. On espère en fait que lors de la phase d'apprentissage d'un modèle, les informations parasites, résultant par exemple de l'application de transformations d'obfuscation, ne sont pas prise en compte dans la caractérisation d'une famille virale ou qu'elles sont prises en compte d'une manière qui ne perturbe pas la détection. Une partie de l'information stockée dans un HMM concerne la chaîne de Markov caché, qui on l'espère contiendra, à l'issue de la phase d'apprentissage, une information synthétique sur le moteur de mutation virale permettant de reconnaître tous les programmes qui en sont issus.

L'un des intérêt de ce type de modèle, comparé à une signature classique (schéma de signature), est qu'il rend intrinsèquement possible, et calculatoirement de manière efficace, de retrouver la séquence d'état $X$ et une information sur la structure du modèle, à partir de la donnée du modèle $\lambda$ et de la séquence observée $Y$. Il est donc possible de caractériser un modèle sur la base d'informations structurelles et qualitatives. Nous allons voir qu'une des propriétés importantes d'un HMM est sa précision et que l'un des facteurs nuisant à sa précision est un nombre trop petit d'états cachés. Cet objectif correspond à la résolution du problème d'optimisation (*tuning*) (problème b)) que nous avons déjà identifié. Le fait de pouvoir retrouver la partie cachée d'un modèle nous permet de comparer entre eux deux modèles, sur la base de critères précis. Remarquons que dans le cas de la recherche de signatures, il est également possible d'extraire une information qualitative sur la précision d'une signature, par le biais d'une analyse boîte noire du moteur de détection. Cette approche est décrite dans [Fil06] et répond également à cette volonté de caractériser un schéma de détection sur la base de critères précis.

Une des limites de ce modèle est qu'il requiert une quantité suffisante de données d'apprentissage. Si cette contrainte ne semble pas poser de problème lorsque l'on dispose du moteur de mutation, comme c'est le cas lors de l'utilisation d'un kit de génération de virus, elle s'avère plus gênante dans le cas de virus possédant un nombre trop faible de variantes ou dont la capture pose problème. C'est en particulier le cas de virus utilisant des sources d'entropie relatives à leur environnement système ou réseau pour muter. Cette approche apparaît donc, à ce titre, complémentaire de l'approche traditionnelle par signature.

La procédure de détection fondée sur l'utilisation de HMMs est très comparable à celle fondée sur une recherche de signature : disposant d'un ensemble de HMMs $\lambda_1, \ldots, \lambda_n$, et d'un code $Y$, nous calculons pour chaque modèle $\lambda_i$ la vraisemblance de la séquence observée $Y$, étant donné le modèle. Dans le domaine de la détection d'intrusion, une telle approche est dite basée sur la connaissance. Si celle-ci dépasse un certain seuil $T$, le programme $Y$ est considéré comme appartenant à la famille virale $\lambda_i$.

Cette méthode de détection n'est pas nouvelle, et a d'ores et déjà fait l'objet d'expérimentations sur les familles virales générées par utilisation des kits de génération de virus G2, MPCGEN, NGVCK et VCL32 [WS06, AS08]. Les auteurs utilisent un score de similarité, appelé LLPO(*Log Likelyhood Per Opcode*), correspondant à la log-vraisemblance d'une observation $Y$, étant donnée le modèle, divisée par la taille |Y| de l'observation. Ils se sont principalement intéressé au pouvoir discriminant d'un tel modèle, en le comparant au pouvoir discriminant d'un autre index de similarité, proposé par Mishra dans [Mis03].

Nous avons présenté au chapitre précédent une formalisation des techniques de furtivité au regard de la théorie de l'information, dont les fondements sont décrits dans le papier [Fil07a]. Dans ce papier, les techniques de furtivité sont comparées aux techniques stéganographiques. La sécurité d'une technique furtive vis-à-vis d'une attaque passive (on suppose que la sonde utilisée pour capturer l'information n'interagit pas avec le système) visant à détecter la présence d'une telle technique sur un système y est défini, en utilisant la distance de Kullback-Leibler entre deux distributions :

- la distribution $DSys$ des objets du système qui peuvent être utilisés comme support par un rootkit ;
- la distribution $DStealth$ de ces même objets, lorsqu'ils sont effectivement utilisés par un rootkit.

La sécurité d'un système furtif (l'homologue d'un système stéganographique dans notre contexte) est défini de la manière suivante :

**Définition 7.5.14 (Sécurité d'un système furtif [Fil07a])** *Un système furtif est dit $\varepsilon$-sûr vis-à-vis des attaques passives si et seulement si :*

$$d_{KL}(p_{DSys}||p_{DStealth}) = \sum_y p_{DSys}(y) \log \frac{p_{DSys}(y)}{p_{DStealth}(y)} \leq \varepsilon.$$

*Lorsque $\varepsilon = 0$, le système est dit parfaitement sûr.*

Remarquons que l'analyse par approche bayésienne ou modèle Markovien, que nous présentons principalement, dans ce chapitre, dans le cadre de l'analyse spectrale, s'applique également à l'analyse des virus furtifs. La procédure de détection fondée sur de tels modèles est cette fois plus comparable à l'analyse comportementale, telle que définie

dans le domaine de la détection d'intrusion. Les données d'apprentissage proviennent d'un système de référence, supposé non corrompu. La procédure de détection consiste à calculer périodiquement la vraisemblance de la séquence observée, compte tenu du modèle. Si celle-ci est en dessous d'une certaine valeur, le système est considéré comme étant corrompu par un rootkit.

De la même manière que nous avons défini la sécurité d'un système furtif, nous pouvons définir de manière duale la précision d'une fonction de détection basée sur un HMM, dans le contexte de l'analyse spectrale :

**Définition 7.5.15 (Précision d'un HMM [Bil06])** *Notons* $p_Y = p(Y_0, \ldots, Y_{T-1})$ *and* $p_Y^{HMM} = p(Y_0, \ldots, Y_{T-1})$ *la vraie distribution et la distribution sous un HMM des variables observées* $Y$*. La précision d'un HMM est d'autant plus grande que la distance de Kullback-Leibler* $d_{KL}$ *entre les distributions* $p_Y$ *et* $p_Y^{HMM}$ *est proche de zéro :*

$$d_{KL}(p_Y || p_Y^{HMM}) = \sum_y p_Y(y) \log \frac{p_Y(y)}{p_Y^{HMM}(y)}.$$

*Dans le cas général, nous considérons des HMM qui ont une distance KL différente de zéro (KL-distance error) :*

$$d_{KL}(p_Y || p_Y^{HMM}) = \varepsilon > 0$$

La précision d'un HMM rend compte de l'entropie relative entre la distribution réelle d'une famille virale et le modèle de détection construit sur la base des données d'apprentissage.

La simulabilité d'une fonction de détection fondée sur des HMM s'en déduit immédiatement :

**Définition 7.5.16 (Simulabilité faible de l'analyse spectrale fondée sur des HMM)** *Pour simuler faiblement l'analyse spectrale fondée sur des HMM, le moteur de mutation doit modifier aléatoirement les instructions* $Y$ *de telle sorte que la distance de Kullback-Leibler* $d_{KL}$ *entre les distributions* $p_Y$ *et* $p_Y^{\lambda_i}$ *est plus grande que* $\varepsilon$*, pour chacun des modèles* $\lambda_i$ *utilisés par la fonction de détection. On doit donc avoir :*

$$\min_{i=1,\ldots,n} d_{KL}(p_Y || p_Y^{\lambda_i}) \geq \varepsilon.$$

En d'autre terme, la distribution du programme viral doit rester à distance respectueuse des distributions reconnues par le détecteur. La manière la plus sûre d'y parvenir est de se rapprocher de la distribution des programmes sains du système. Une telle approche a un double avantage :

- une fonction de détection par analyse spectrale prenant en compte une telle famille virale prend le risque de voir augmenter considérablement le risque de faux positifs. En effet, la vraisemblance d'une séquence observée $Y$ compte tenu du modèle correspondant sera trop élevé, et dépassera la valeur du seuil de détection $T$, dès lors que le programme $Y$ est un programme sain utilisé par le virus comme "modèle". Remarquons qu'un tel programme viral correspond à un système de furtivité au moins $T$-sûr vis-à-vis de de la fonction de détection, même si ici une alerte est déclenchée (fausse alerte).
- une fonction de détection par analyse spectrale ne prenant pas en compte les familles virales dont la distribution statistique se rapproche trop de celle des programmes sains du système prend le risque de voir augmenter le risque de faux négatifs.

Nous voyons donc apparaître une autre limite de la détection par analyse spectrale : les virus hébergés par un programme sain sont intrinsèquement difficiles à détecter. Les virus métamorphes utilisant des techniques d'intégration de code[1] le sont encore plus.

Examinons à présent les caractéristiques de la séquence observée qui peuvent rendre difficile la construction d'un modèle HMM précis. La première caractéristique concerne naturellement la distribution de la séquence observée. Si celle-ci est choisie de manière inadéquate, la fonction de détection restera inefficace. Ce critère est lié à l'extraction des informations utilisées pour entraîner un modèle : ces sources d'information doivent être fiables. Dans le cas de

[1] Le virus se mélange dans le flot des instructions de son hôte, par utilisation d'un désassembleur et d'un moteur lui permettant de reconstruire le binaire hôte. Une telle technique est notamment utilisée par le moteur de mutation du virus Zmist, appelé Mistfall [Z0m00b].

l'analyse spectrale, les séquences observées correspondent au code assembleur des virus d'une famille. Le désassemblage doit être correct. La même contrainte s'applique lors de la phase de détection utilisant ce modèle. Ce critère, lorsqu'il s'applique à la détection des virus furtifs, est encore plus important dans la mesure où la sonde dédiée à la récupération des séquences observées ne doit pas interagir avec ces dernières. Si c'est le cas, le modèle doit être corrigé en conséquence.

Nous avons déjà évoqué l'importance du nombre des états cachés pour assurer la précision d'un modèle HMM. Pour comprendre ce critère, il est utile d'adopter une vision informationnelle d'un modèle HMM, à travers la notion de canal bruité : si l'on note $Y_t$ l'observation à l'instant $t$ (une instruction assembleur dans le cas de l'analyse spectrale) et $Y_{\neg t}$ la collection d'observations aux autres instants excepté l'instant $t$. La dépendance de $Y_t$ vis à vis de $Y_{\neg t}$ peut être vue comme le passage dans un canal bruité. Le taux de transmission d'information dans ce canal est donné par l'information mutuelle $I(Y_{\neg t}, Y_t)$ entre ces deux ensembles de variables. Par définition du modèle HMM, toute observation $Y_t$ est indépendante de $\{X_{\neg t}, Y_{\neg t}\}$, conditionnellement à l'état caché $X_t$. Par conséquent, une variable cachée $X_t$ sépare $Y_t$ de son contexte $Y_{\neg t}$ :

$$Y_{\neg t} \longrightarrow \boxed{\text{Canal I}} \xrightarrow{X_t} \boxed{\text{Canal II}} \longrightarrow Y_t$$

D'une certaine manière, un HMM compresse l'information relative à $Y_t$ contenue dans $Y_{\neg t}$ dans la variable discrète $X_t$. Pour que le modèle HMM soit précis, le canal bruité représenté ci-dessus doit avoir le même taux de transmission d'information que le canal bruité suivant :

$$Y_{\neg t} \longrightarrow \boxed{\text{Canal}} \longrightarrow Y_t$$

Sous l'hypothèse de précision du modèle HMM, on a :

$$d_{KL}(p_Y || p_Y^{HMM}) = \sum_y p_Y(y) \log \frac{p_Y(y)}{p_Y^{HMM}(y)} = 0.$$

Si cette condition est vraie, l'information mutuelle entre deux sous-ensembles quelconques $S_1$ et $S_2$ de variables de chaque distribution est égale :

$$I(Y_{S_1}, Y_{S_1}) = I^{HMM}(Y_{S_1}, Y_{S_1}).$$

En particulier, on a :

$$I(Y_{\neg t}, Y_t) = I^{HMM}(Y_{\neg t}, Y_t).$$

Donc, on a :

$$\begin{aligned} I^{HMM}(Y_t, (X_t, Y_{\neg t})) &= I^{HMM}(Y_t, Y_{\neg t}) + I^{HMM}(Y_t, X_t | Y_{\neg t}) \\ &= I(Y_t, Y_{\neg t}) + I^{HMM}(Y_t, X_t | Y_{\neg t}) \end{aligned}$$

La quantité $I^{HMM}(Y_t, (X_t, Y_{\neg t}))$ peut également s'écrire de la manière suivante :

$$\begin{aligned} I^{HMM}(Y_t, (X_t, Y_{\neg t})) &= I^{HMM}(Y_t, X_t) + I^{HMM}(Y_t, Y_{\neg t} | X_t) \\ &= I^{HMM}(Y_t, X_t) \text{ (car c'est une HMM)} \end{aligned}$$

D'où :

$$\begin{aligned} I^{HMM}(Y_t, X_t) &= I(Y_t, Y_{\neg t}) + I^{HMM}(Y_t, X_t | Y_{\neg t}) \\ &\geq I(Y_t, Y_{\neg t}) \end{aligned}$$

Considérons à présent l'entropie de l'état caché $X_t$. On a donc :

$$\log |X| \geq H^{HMM}(X_t) \geq H^{HMM}(X_t) - H^{HMM}(Y_t | X_t) = I^{HMM}(Y_t, X_t) \geq I(Y_t, Y_{\neg t}).$$

On en déduit le théorème suivant :

**Théorème 7.5.1 (Condition nécessaire à la précision d'un HMM [Bil06])** *Les conditions suivantes sont nécessaires à la précision d'un HMM :*

- *le taux de transmission $I_{HMM}(Y_{\neg t}, X_t)$ entre $Y_{\neg t}$ et $X_t$ doit être plus grand que $I(Y_t, Y_{\neg t})$ ;*
- *le taux de transmission $I_{HMM}(X_t, Y_t)$ entre $X_t$ et $Y_t$ doit être plus grand que $I(Y_t, Y_{\neg t})$ ;*

– *la variable $X_t$ doit avoir suffisamment de capacité de stockage pour coder l'information circulant entre les deux canaux bruités. Plus précisément, le nombre d'états cachés $N = |X|$ doit satisfaire la condition suivante :*

$$N \geq 2^{I(Y_t, Y_{\neg t})}.$$

Les deux premières conditions sont intuitivement assez naturelles : si l'une de ces conditions n'est pas satisfaite, l'un des deux canal (canal I ou II) va se transformer en goulot d'étranglement. Chacun des deux canaux doit avoir une capacité suffisante. L'imprécision d'un HMM peut résulter de l'utilisation d'une famille de distribution d'observation impropre, qui correspond à l'utilisation d'un canal possédant une capacité insuffisante.

La dernière condition est sans doute la plus importante car dans tous les cas, un goulot d'étranglement du canal peut provenir du nombre fixé d'états cachés. Dans le contexte de l'analyse spectrale, si les instructions assembleur produites par un moteur de mutation possèdent une information mutuelle importante, l'approximation par un modèle HMM risque d'échouer en raison du nombre trop important d'états cachés requis. Rappelons que l'information mutuelle $I(Y_1, Y_2)$ entre entre deux variables aléatoires $Y_1$ et $Y_2$ est aussi la distance $KL$ entre la distribution jointe $p_{Y_1,Y_2}$ et le produits indépendant des distributions $p_{Y_1}p_{Y_1}$. Cette condition nécessaire à la précision du modèle, et donc à sa sécurité vis à vis d'une attaque par simulation, impose un niveau de dépendance faible entre les instructions assembleur produites par le moteur de mutation que l'on cherche à modéliser par HMM.

## 7.6 Conclusion

Nous avons proposé dans ce chapitre un cadre statistique adapté à la modélisation de la détection antivirale. L'idée maîtresse de cette approche est qu'il est possible de voir tout algorithme de détection comme une combinaison de tests statistiques. Nous avons proposé une version statistique du résultat d'indécidabilité de Cohen, permettant de définir précisément les limitations et aspects inhérents à la détection en pratique. En se fondant sur le concept de simulabilité des tests statistiques, il est possible de caractériser un schéma de détection par une mesure de la difficulté à le contourner. Là où Chess et White ont seulement donné des résultats d'existence de virus indétectable, nous caractérisons par ce concept les différente techniques pratiques pour effectivement construire de tels virus indétectables.
Nous illustrons cette position par une étude des algorithmes de détection fondés sur l'analyse du spectre des instructions. Nous avons considéré en premier lieu un modèle de détection élémentaire et démontré la simulabilité faible du test statistique correspondant.
Nous nous sommes ensuite intéressé aux possibilités de contournement de schémas de détection plus sophistiqués, les réseaux bayésiens. Nous avons étudié en détail l'application de deux de ces modèles à l'analyse spectrale : le modèle de Bayes naïf et le modèle de Markov caché. Le principal intérêt de ces modèles réside dans le fait qu'il est possible de définir un tel modèle pour une famille virale quelconque. Tous ces modèles sont dérivés d'un unique modèle initial avec une paramétrisation adaptée. Il est donc possible de caractériser par un tel modèle l'ensemble des codes viraux générés par un kit de génération de virus ou issus d'un même moteur de mutation. Nous avons défini la précision de ces modèles, en se fondant sur des mesures issues de la théorie de l'information et étudié la simulabilité de ces modèles. Il nous est donc possible dans ce cadre théorique de formaliser les notions de complétude et de robustesse associées à une fonction de détection, au regard d'une classe de transformations d'obfuscation (l'ensemble des transformations implémentées par un kit de génération ou par un moteur de mutation viral).

Rappelons que l'intérêt de ce travail n'est pas d'identifier les techniques les plus adaptées pour construire un virus indétectable, mais de disposer d'un cadre plus puissant pour évaluer et tester les produits antivirus. Cet effort de modélisation a pour objectif la définition de critères précis permettant de mesurer la difficulté à contourner une fonction de détection fondant son verdict sur des données statistiques. L'étude du design de ces algorithmes nous permet de les qualifier en terme de résistance aux transformations d'obfuscation, sur la base d'informations spectrale. Nous pensons que ce cadre formel complète utilement les cadres déjà utilisé dans cet objectif (théorie de la complexité et grammaires formelles, théorie de l'interprétation abstraite).

### 7.6.1 Travaux futurs

Les méthodes bayésiennes et les HMM sont d'ores et déjà utilisées dans une large variété d'applications, incluant la reconnaissance de textes, de voix et de langages, la génétique, la cryptanalyse, le filtrage des SPAMs, et la détection

virale.
Concernant la dernière application, des travaux supplémentaires sont sans doute nécessaires pour affiner les modèles existants ou trouver des modèles plus adaptés dans l'étendue des modèles graphiques ou des réseaux bayésiens.

Ces méthodes de classification sont prometteuses, et leur utilité dans la détection par analyse spectrale des familles virales générées par un kit de génération ou un moteur de mutation viral, indéniable. En effet, les VGKs fournissent généralement une quantité suffisante de données pour l'apprentissage des modèles. La simulabilité des tests statistiques se révèle être un concept très utile pour guider l'analyse de rigidité de ces schémas de détection.

# Troisième partie

# Évaluation

Chapitre 8

# Évaluation d'un produit anti-virus

Ce chapitre présente une approche méthodologique, une plate-forme de tests et une revue des critères permettant de qualifier un algorithme de détection et plus globalement d'évaluer la robustesse des fonctions de sécurité d'un produit antivirus.

Cette étude d'efficacité se fait au regard d'une menace virale identifiée. Nous avons concentré notre analyse de la menace virale sur les mécanismes et outils déjà utilisés ou susceptibles de l'être dans un avenir proche par les concepteurs de virus :

- dans le chapitre 2, nous avons étudié les mécanismes cryptographiques adaptés au contexte boîte blanche, i.e. permettant à un virus de protéger ses données critiques en confidentialité dans un environnement complètement maîtrisé par l'attaquant ;
- dans le chapitre 3, nous avons complété ce panorama de la menace virale par l'étude des chaînes de compilation spécialisées, qui ont vocation première à protéger de code d'applications légitimes et peuvent être utilisés à des fins illégitimes par les concepteurs de virus.

Face à cette évolution de la menace virale, l'architecture des systèmes de détection doit progresser, du simple moteur de recherche de motifs vers un système expert, capable de corréler plusieurs modèles de détection. Plusieurs formalismes sont applicables à la modélisation d'un moteur de détection, et permettent de rendre compte du type et de la qualité des algorithmes implémentés. Nous avons vu que les critères de correction et de complétude sont suffisamment générique pour être exprimés dans chacun de ces cadres formels. Malheureusement, ces critères sont rarement utilisés par les concepteurs de produits anti-virus pour démontrer la pertinence (et faire état des limites) de leurs choix de conception. Ces preuves sont à la charge de l'évaluateur. En outre, une telle analyse doit être menée en boîte blanche, c'est-à-dire sur la base des spécification détaillées de l'algorithme de détection. En l'absence d'une telle information, l'évaluation doit être menée en boîte noire.

Afin d'être aussi exhaustive et impartiale que possible, cette évaluation doit être menée méthodiquement et s'appuyer sur des critères partagés par tous.

Nous présentons une approche dont l'objectif est de rendre plus aisé et pertinent le choix d'un produit antivirus. Parmi les qualités que l'on peut attendre d'un tel produit, on trouve typiquement l'utilisation optimale des ressources et la réactivité du constructeur, concernant en particulier la mise à jour de la base des signatures virales et des moteurs. Si ces exigences sont significatives, d'autres vérifications méthodiques et techniques sont requises pour appuyer le choix d'une personne ou une entreprise. Dans le schéma d'évaluation des Critères Communs [Her03], un *Profil de Protection* est proposé pour aider un constructeur de logiciel à concevoir un produit qui pourra être évalué par un laboratoire d'évaluation indépendant. Les Profils de Protection [PP] sont rédigés conformément aux standard des Critères Communs. Partant d'un Profil de Protection, nous proposons des tests qui peuvent être menés pour valider les exigences de sécurité d'un produit antivirus. L'utilisation conjointe d'un Profil de Protection et la spécification de tests nous paraît être une base saine pour mesurer la confiance que l'on peut accorder à un produit antivirus.

## 8.1 Introduction

L'utilisateur final, lorsqu'il choisit un produit antivirus, se concentre sur des qualités classiques :
- utilisation optimale des ressources ;
- réactivité du constructeur.

L'analyste d'un laboratoire d'évaluation, lorsqu'il juge du niveau de confiance qu'il peut accorder à un produit de sécurité, se focalise sur :
- la conformité à des critères standards, partagés par le constructeur et le laboratoire d'évaluation ;
- la robustesse vis-à-vis de la menace ;
- la qualité de la conception et de l'implémentation, au regard de l'état de l'art dans ce domaine.

Partant des critères partagés par l'évaluateur et le constructeur, des tests fonctionnels permettent de valider les exigences de sécurité obligatoires.

La veille technologique de la menace, par utilisation de bases de vulnérabilités comme CVE [Bak99], BugTraq [Foc06] ou Vigil@nce [Vig06i], permet de juger de la robustesse d'un produit de sécurité, par le développement de tests de pénétration adaptés. Les tests fonctionnels et de pénétration permettent de jauger la qualité de la conception et de l'implémentation.

Notre définition de l'efficacité couvre ces deux points de vue (utilisateur final et évaluateur) :
- l'efficacité d'un produit anti-virus repose sur la conformité à des critères que nous définissons et qui correspondent à l'état de l'art dans le domaine et à une expression de besoin de sécurité. Un plan de tests doit servir de support à l'analyse des fonctions et mécanismes de sécurité. L'efficacité du produit anti-virus est évaluée au regard de la robustesse de chacun des mécanismes qui le caractérisent.

Choisir un produit antivirus peut être une tâche difficile. Une des raisons est que les mécanismes internes de ces produits (en particulier les algorithmes utilisés et les particularités de conception) ne sont généralement pas bien documentés. En conséquence, comment être certain de faire le bon choix ?

Certaines sociétés accordent aux produits antivirus des récompenses ou des certificats de qualité :

Un premier exemple est la récompense VB100% Award [lab06a]. Pour obtenir ce label, un produit antivirus doit détecter tous les virus d'une liste prédéfinie (virus *In-the-Wild*[1]) et ne doit générer aucun faux positif lors de l'analyse d'un ensemble de fichiers sains.

Un second exemple est la récompense WCL Checkmark [Lab06c]. Pour obtenir la WCL Checkmark, niveau 1, un produit antivirus doit détecter tous les virus de la liste *In-the-Wild* et ne doit pas lever de fausse alarme lors de l'analyse d'un ensemble de fichiers sains. Pour obtenir la WCL Checkmark, niveau 2, le produit doit en outre être en mesure de désinfecter tous les virus de la Wild-List (lorsque c'est possible). Des *checkmarks* supplémentaires peuvent être fournies par WCL, qui contrôlent le taux de détection vis-à-vis de types d'infections informatiques spécifiques (*WCL Trojan Checkmark*, *WCL Spyware Checkmark*).

Un troisième exemple est le certificat ICSA Labs [Lab06b]. Pour obtenir ce certificat, un produit antivirus doit satisfaire toutes les exigences du module primaire de critères qui correspondent à sa catégorie de produit (par exemple, la catégorie antivirus de poste de travail ou de passerelle). Les tests relatifs au module secondaire de critères (module de désinfection) est optionnel. Une matrice appelée *Certification Test Suites Matrix* permet d'associer à un type de produit antivirus plusieurs collections de virus (*In-the-wild*, *common infectors* et *Zoo*).

Les tests des laboratoires Virus Bulletin, West Coast Labs, and ICSA Labs focalisent sur les taux de détection. Tous utilisent (parmi d'autres collections de virus) la célèbre Wild-List pour leurs tests. Examinons plus en détail l'offre de ICSA Labs.
Les critères de certification de ICSA Labs sont assez similaires aux Critères Communs : les antivirus satisfaisant les

[1] La Wild-List est une liste de virus connus pour être actifs et répandus. Les virus de cette liste sont habituellement appelés virus In-The-Wild.

exigences du module primaire doivent avoir la capacité à :

- détecter et prévenir toute réplication des virus,
- ne reporter aucun faux positif,
- journaliser les tentatives de détection virale,
- et effectuer les fonctions d'administration nécessaires.

Ces exigences peuvent être vues comme un sous-ensemble des exigences fonctionnelles des Critères Communs telles que décrites dans la suite de ce chapitre.
Cependant, l'approche de ICSA Lab ne prend pas en compte les composants d'assurance relatifs aux choix de conception ou la qualité du processus de développement du produit. Les critères de certification de ICSA Lab ne sont basés ni sur des principes fondamentaux de conception ou d'ingénierie, ni sur une expertise de la technologie sous-jacente. Leur approche est une approche boîte noire, qui est orientée résultats.
Le principal avantage de l'approche d' ICSA Lab est que les procédures de test sont aisément automatisables et reproductibles pour un produit antivirus. Son inconvénient majeur est que les tests ne sont pas assez précis pour accroître le niveau d'assurance de l'évaluation, dans la mesure où ils ne sont pas ouverts à l'interprétation et à l'analyse. Il s'agit à la fois d'une qualité et d'une limitation. À titre d'exemple, ces critères ne sont pas suffisamment précis pour rendre possible une revue du code source.
En outre, les critères de certification de ICSA Lab ne prennent pas en considération les exigences relatives à l'environnement de la cible d'évaluation. Le périmètre de leur analyse est limité au produit.
Finalement, il reste difficile de savoir quels tests ont étés menés au regard des critères choisis par cette société.

Dans ce chapitre, une démarche est décrite pour rendre le choix d'un produit antivirus plus aisé et pertinent. Sur la base d'un Profil de Protection, nous listons des tests qui peuvent être mis en oeuvre pour valider les exigences de sécurité de tout produit antivirus.

Notre approche est complémentaire de celle introduite par les laboratoires mentionnés auparavant. Nous focalisons sur la robustesse des fonctions et mécanismes de sécurité, au regard des vulnérabilités spécifiques de cette catégorie de produits. Nous élargissons le périmètre de leur évaluation en prenant en compte l'environnement système et la conception interne.

Il nous a paru important, au regard de la menace virale actuelle, de nous munir d'outils permettant de reproduire ou de simuler une attaque virale. Nous avons donc implémenté plusieurs mécanismes viraux caractéristiques. Il est à noter que ces techniques virales sont de plus en plus souvent appliquées (par les chercheurs) à la validation de nouvelles techniques de détection comportementale d'intrusion virale [CJ04, FJL07, FEG+07, FGJ+08].
Nous avons choisi d'appliquer ces techniques virales à la validation des fonctions de sécurité des produits anti-virus. Nos réalisations les plus complètes sur ce point sont : la réalisation d'un moteur d'obfuscation et la réalisation d'un outil d'injection à distance de code relogeable.
Dans le contexte de cette lutte du glaive contre le bouclier, il nous a paru important de comprendre également les contraintes liées à l'implémentation d'un produit anti-virus. Nous avons donc spécifié et développé plusieurs des composants actuels d'un produit anti-virus. Nos réalisations les plus complètes sur ce point sont : la réalisation d'un programme d'analyse à la demande de code viral et réalisation d'un moniteur temps réel de détection virale.
Nous avons enfin spécifié un ensemble de tests (benchmarks, tests fonctionnels, tests de pénétration) permettant d'éprouver l'efficacité des fonctions de sécurité d'un produit anti-virus face à la menace virale.

Nous proposons une approche et des outils permettant d'aider un utilisateur ou une entreprise à choisir un produit antivirus sur étagères avec un certain degré de confiance :

- approche méthodologique : utilisation d'un profil de protection de niveau standard des produits antivirus ;
- analyse de robustesse des fonctions et mécanismes d'un produit antivirus ;
- spécification et implémentation de tests destinés à valider la conformité des fonctions et mécanismes vis-à-vis du profil de protection (tests de conformité) et leur efficacité/robustesse (tests de pénétration).

Des outils spécialisés ont été spécifiés et développés pour permettre la mise en oeuvre des tests d'efficacité :

- transformations d'obfuscation élémentaires (fondé sur le moteur Mistfall [Z0m00b]) ;
- injection de code relogeable sur un service réseau.

En outre, certains composants d'un produit antivirus ont été spécifiés et développés, de manière à mieux comprendre les contraintes liées à l'implémentation d'un produit antivirus :
- fonction de détection à la demande, fondée sur un base de signatures et sur le produit antivirus open source ClamAV [K+07] ;
- moniteur temps-réel, permettant de contrôler l'accès au système de fichiers.

Cette approche méthodologique et le banc de tests ont été mis en oeuvre sur de nombreux produits antivirus.

Une partie du contenu de ce chapitre a été publiée dans les actes de la conférence EICAR 2006 [Jos06a], ainsi que dans le journal MISC en 2007 [FEG+07] et 2008 [FGJ+08].

La suite du présent chapitre est organisé de la manière suivante :
- la section 8.2 présente notre démarche méthodologique, qui se traduit par l'énoncé de la problématique de sécurité d'un produit antivirus au regard d'une menace virale identifiée ;
- la section 8.3 présente les outils et critères théoriques sur lesquels fonder une analyse (en boîte blanche) des spécifications et de la conception du moteur de détection d'un produit antivirus ;
- la section 8.4 présente les outils et critères techniques sur lesquels fonder une analyse (en boîte noire) de conformité et de robustesse de l'implémentation du moteur de détection viral ;
- la section 8.5 présente la plate-forme de test ;
- la section 8.6 conclue sur nos résultats et identifie les axes de recherche à investir dans le futur.

## 8.2 Énoncé de la problématique de sécurité

### 8.2.1 Outils et critères méthodologiques

La première chose à faire lorsque l'on cherche à comparer des produits consiste à définir clairement notre cible, au niveau de la catégorie de produit et des caractéristiques de son environnement. Nous avons adopté le même périmètre que celui défini par le profil de protection : *US Government Protection Profile Anti-Virus Applications for Workstations in Basic Robustness Environments* [CCE05] [2].

Un Profil de Protection est une spécification des exigences d'assurance sur l'information indépendante des spécification. Les Profils de Protection sont une combinaison complète d'objectifs de sécurité, d'exigences fonctionnelles relatives à la sécurité, d'exigences d'assurance sur l'information, d'hypothèses et d'éléments de preuve. Les Profils de Protection sont rédigés en conformité avec le standard des Critères Communs. Dans le schéma d'évaluation des Critères Communs [Her03], un Profil de Protection est prévu pour permettre à un constructeur de logiciel de concevoir un produit qui pourra être évalué par un laboratoire d'évaluation indépendant. Les analystes de sécurité effectuent des tests et évaluent la couverture des exigences de sécurité. Plusieurs acteurs peuvent prendre part au processus d'évaluation : analystes de sécurité, développeurs du produit antivirus cible, une tierce partie, chargée d'arbitrer et de contrôler le processus. Par exemple, en France, cette tierce partie est une organisation gouvernementale qui dépend des services du premier ministre. Le laboratoire d'évaluation doit en outre être agréé par cette tierce partie. Ce Profil de Protection [CCE05] spécifie les exigences de sécurité minimum pour un produit antivirus utilisé sur des stations de travail du gouvernement US pour un environnement de niveau de robustesse basique. Le niveau de robustesse de la cible d'évaluation (i.e. dans quelle mesure la cible d'évaluation peut se protéger elle-même ainsi que ses ressources) de robustesse basique est considéré comme suffisant lorsque l'environnement correspond à un faible niveau de menace ou lorsque la compromission de l'information protégée n'aura pas un impact significatif sur les objectifs de mission. Ce point est discuté en détail dans [CCE05]. Ce profil de protection est basé sur les Critères Communs, version 2.2. La figure 8.1 représente la cible d'évaluation et son environnement.

Nous avons utilisé ce profil de protection et les critères communs comme base (méthode et terminologie) pour nos analyses. Par exemple, dans cette terminologie, le terme *virus* est utilisé de manière générique pour désigner une suite d'exploits de vulnérabilités de sécurité, comme c'est le cas d'un ver ou d'un cheval de Troie. Le même terme

[2] Ce profil de protection a été mis à jour en 2006 [CCE06], puis en 2007 [CCE07], principalement pour des raisons de compatibilité avec les dernières versions des critères communs.

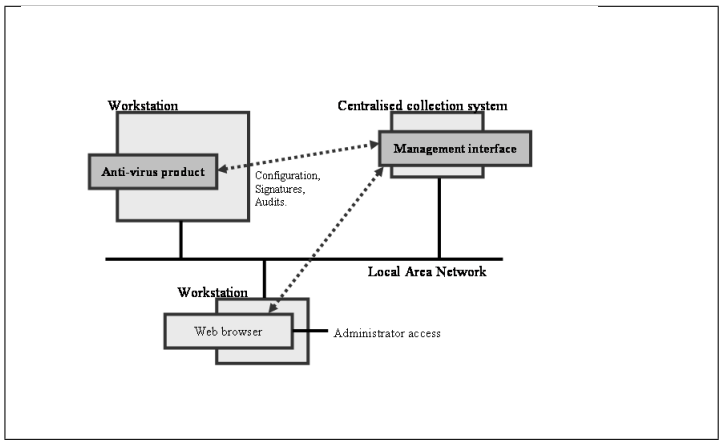



FIG. 8.1 – Cible d'évaluation et son environnement

est utilisé plus spécifiquement pour désigner les exploits qui se répliquent eux mêmes. Le terme *antivirus* désigne typiquement les mesures utilisées pour contrer la suite entière d'exploits, pas seulement la définition plus spécifique de la notion de virus.

L'idée consiste à s'appuyer sur cette cible de sécurité générique pour définir une stratégie de test permettant à l'évaluateur de mener à bien ses tâches :

- analyse des fournitures et des éléments de preuve fournis par le développeur (les fournitures comprennent notamment les différents guides et les documents de spécification et de conception de l'application, parfois le code source) ;
- validation de la démarche de test du développeur (repasser les tests et statuer sur la couverture des tests du développeur) ;
- analyse de conformité des fonctions de sécurité (c'est-à-dire vérifier que les fonctions de sécurité se comportent suivant leurs spécifications, en spécifiant des tests complémentaires de conformité) ;
- analyse indépendante de vulnérabilité (spécification de tests de pénétration, visant à mettre en oeuvre des menaces ou à enfreindre des règles de la politique de sécurité en exploitant des vulnérabilités).

Cette dernière étape est cruciale, puisqu'elle doit permettre de confirmer ou d'infirmer, par une estimation indépendante, le potentiel minimum d'attaque nécessaire pour mettre en oeuvre les attaques, tel que l'a estimé le développeur.

En résumé, l'utilisation d'un profil de protection, c'est à dire d'une spécification de besoin de sécurité générique à la catégorie des produits anti-virus (la cible de sécurité constituant le point d'entrée de toute évaluation selon les critères communs) va nous servir de base à l'élaboration d'une stratégie de test. Sur la base du résultat de ces tests, il est possible de mener les différentes tâches d'une évaluation selon les critères communs. Je pense que ce jeu de test peut permettre également à un particulier mener une analyse indépendante de vulnérabilité, la plupart des tests étant menés en boite noire.
Les autres tâches d'évaluation nécessitent des informations sur les spécifications ou la conception du produit qui ne sont généralement pas fournies à l'utilisateur final.

Remarquons que depuis la publication de ce chapitre dans les actes de la conférence EICAR [Jos06a], une seule évaluation de produit anti-virus (il s'agit du produit McAfee VirusScan [McA07, CAF07]) a été menée en conformité avec ce profil de protection (version 1.1, [CCE06]).

Le rapport de Certification Sécurité de Premier Niveau (CSPN) émis par la DCSSI fournit aux commanditaires un document leur permettant d'attester du niveau de sécurité offert par le produit dans les conditions d'utilisation ou d'exploitation définies dans ce rapport pour la version qui a été évaluée. Il est destiné également à fournir à

l'acquéreur potentiel du produit les conditions dans lesquelles il pourra exploiter ou utiliser le produit de manière à se trouver dans les conditions d'utilisation pour lesquelles le produit a été évalué et certifié.
Ce rapport de certification est destiné à être lu conjointement aux guides d'utilisation et d'administration évalués ainsi qu'à la cible de sécurité du produit qui décrit les menaces, les hypothèses sur l'environnement et les conditions d'emploi présupposées afin que l'utilisateur puisse juger de l'adéquation du produit à son besoin en termes d'objectifs de sécurité.
Il valide les travaux d'évaluation menés par un centre d'évaluation agréé, dans un temps et une charge contraints (une vingtaine d'homme/jour), comprenant notamment :

- une analyse de conformité du produit à la cible de sécurité ;
- une description de l'installation, des options d'installation retenues pour le produit et des non-conformités éventuelles vis-à-vis de la documentation ;
- une analyse de la conformité des fournitures (analyse de la documentation, éventuellement revue du code source) et des fonctionnalités du produit ;
- une analyse de la résistance des mécanismes et des fonctions ;
- une analyse des vulnérabilités (conception, construction, etc.) ;
- une analyse de la facilité d'emploi (cas où la sécurité est ambiguë) et préconisations, recommandations pour une utilisation sûre du produit ;
- une analyse de la résistance des mécanismes cryptographiques et du générateur d'aléas.

Je pense que la stratégie de test présentée dans la suite de ce chapitre permet d'accélerer sensiblement la réalisation des tâches d'une évaluation CSPN d'un produit anti-virus.

La CSPN a été mise en oeuvre à titre expérimentale sur l'année 2008. La documentation décrivant ce processus est donc suceptible d'évoluer au fur et à mesure de la prise en compte des retours d'expérience. Le premier produit certifié au titre de la CSPN n'est pas un produit anti-virus (il s'agit de l'hyperviseur de machines virtuelles *Trango Hypervisor*).

### 8.2.2 Utilisation du profil de protection

Nous rappelons dans cette section, sous forme résumée, les principales données du profil de protection. Une description plus complète est fournie dans le profil de protection.

Le périmètre logique d'un produit anti-virus est donné par la figure 8.1. La TOE comprends le moteur de détection et une interface de gestion que l'administrateur utilise pour :

- configurer et déployer une politique de sécurité ;
- consulter les fichiers journaux.

Le périmètre physique de la TOE comprend généralement le logiciel installé sur un serveur maître et les logiciels installés sur les stations de travail. La plate-forme matérielle sur laquelle ces composants logiciels sont installés ne fait pas partie de la TOE. Les systèmes d'exploitations sont également exclus du périmètre de l'évaluation.
La TOE peut distribuer les mises à jour par le biais d'un répertoire maître, mis à disposition des systèmes clients.

Les principales fonctionnalités de sécurité de la TOE sont, pour le moteur de détection :

- protection temps-réel des ports, des fichiers et des processus résidants en mémoire ;
- fonction d'analyse des mails ;
- fonction de sélection d'une profondeur d'analyse pour les fichiers archives ;
- fonction de mise à jour automatique de la base des signatures virales.

D'autres fonctionnalités de sécurité sont souvent proposées par les produits anti-virus :

- fonction de protection contre les débordements de tampon ;
- fonction de détection des virus furtifs ;
- fonction de détection des virus de macro (JavaScript, VBScripts) ;
- fonction de mise à jour du moteur de détection ;

- fonction de gestion des alertes.

L'interface de gestion inclue les fonctionnalités suivantes :
- fonction d'audit des fichiers journaux ;
- fonction de gestion centralisée des paramètres de détection et des fichiers journaux ;
- opérations cryptographiques de chiffrement et de signature des composants du logiciels ;
- fonction de protection de la TOE elle-même.

Les données utilisées par les fonctions de sécurité sont :
- les signatures virales ;
- les empreintes numériques ;
- les enregistrement d'audit.

Les attributs de sécurité de la TOE sont les rôles permettant de déterminer :
- les actions qu'un utilisateur peut prendre en cas de détection ;
- le paramétrage du moteur de détection, pour les règles d'exclusion notamment.

Les données utilisateur sont les fichiers qui sont sur les postes de travail.

L'argumentaire concernant la séparation et le non-contournement sont partagés en terme de responsabilité entre la TOE et son environnement (matériel, système d'exploitation). La TOE doit être isolée sur son propre domaine, c'est-à-dire que l'environnement système maintient et contrôle l'exécution des fonctions de sécurité séparément des autres processus.

L'environnement de sécurité est caractérisé par des hypothèses faites sur l'environnement, des menaces qui doivent être contrées par la TOE et des politiques de sécurité organisationnelle.

Les objectifs de sécurité de la TOE et de son environnement permettent d'identifier les responsabilités de la TOE et de son environnement.

Les exigences fonctionnelles de sécurité sont également partagées entre la TOE et son environnement. Les exigences de sécurité portant sur les composants d'assurance sont également documentées dans le profil de protection.

Sur la base de cette spécification de besoin, nous sommes en mesure de définir une stratégie de test permettant de couvrir les exigences de sécurité identifiées. Ce jeu de test servira de support à l'évaluation, définie rappelons le, comme le processus permettant une estimation indépendante de l'efficacité des mesures de sécurité par rapport à un ensemble donné d'exigences. Ces exigences sont formulées comme le fait d'atteindre des objectifs de sécurité et de contrer des menaces.

## 8.3 Pertinence de la conception du moteur de détection

Je présente dans cette section une revue des critères théoriques permettant d'apprécier la pertinence de la conception d'un moteur de détection. Il apparait malheureusement difficile d'identifier des critères théoriques suffisament puissants pour comparer entre eux des algorithmes fondés sur des méthodes de détection fondamentalement différentes dans leur approche, comme c'est le cas par exemple de l'analyse spectrale et de l'analyse syntaxique ou sémantique.
Cependant, parmi un ensemble de méthodes issues d'une même approche, je pense possible de comparer entre eux deux algorithmes de détection, sur la base des critères de correction et de complétude notamment.

### 8.3.1 Outils et critères théoriques

Une évaluation du moteur de détection selon des critères objectifs requiert un effort de modélisation. Nous pouvons citer notamment trois cadres théoriques dans lesquels il est possible de dégager des critères utiles à l'analyse en boîte blanche d'un moteur de détection :

- la théorie de la complexité et l'étude des grammaires et langages formels ;
- la théorie de l'interprétation abstraite (algèbre) ;
- la théorie de l'information, des probabilités et statistiques.

### Cadre des grammaires et langages formels

La modélisation par utilisation de la théorie de la complexité, permet de définir des limites théoriques et de qualifier les solutions algorithmiques approximatives proposées, en terme de performance mais également en terme de complétude et de correction. Nous avons présenté les principaux résultats associés à ce cadre théorique dans le chapitre 1.
Dans ce cadre théorique, nous pouvons considérer les critères suivant [Fil07b] :

| CRITERE | DESCRIPTION |
|---|---|
| correction | un schéma de détection $(L_{\mathcal{V}}, f_{\mathcal{V}})$[3] est dit correct vis-à-vis d'un ensemble viral $\mathcal{V}$ lorsque le langage $L_{\mathcal{V}}$ décrivant cette famille virale est suffisamment précis pour éviter le risque qu'un programme bénin soit considéré à tort comme appartenant à ce langage. La fonction de détection $f_{\mathcal{V}}$ ne reconnaît que les mots appartenant à ce langage. |
| complétude | un schéma de détection $(L_{\mathcal{V}}, f_{\mathcal{V}})$ est dit complet vis-à-vis d'un ensemble viral $\mathcal{V}$ lorsque le langage $L_{\mathcal{V}}$ décrivant cette famille virale est complet, dans le sens où toutes les variantes possibles du virus appartiennent à ce langage. Le risque résiduel est dans ce cas que des programmes bénins appartiennent également à ce langage. |

Comme nous l'avons rappelé dans le chapitre 2, dans le contexte viral, un moteur de mutation, c'est-à-dire un moteur de polymorphisme ou de métamorphisme, doit renforcer à la fois la diversité et la résistance vis-à-vis de la reconnaissance de motifs. Le polymorphisme [Qoz99] et le métamorphisme [Fil07b] peuvent être formalisés en terme de productions de grammaire. La difficulté à reconnaître un virus correspond à l'expressivité nécessaire de la machine ou de l'automate qui est capable de reconnaître le langage $L(G)$ généré par la grammaire $G$ en appliquant ses règles de production (voir table 2.1 page 31).

Le type de grammaire générative est crucial lorsque l'on conçoit un moteur de polymorphisme. Ce point a été discuté plus en détail dans la section 2.6.4 du chapitre 2. Nous avons alors observé que plus la grammaire est irrégulière, plus il est difficile de dériver un automate à partir de la grammaire qui soit capable de détecter la fonction de chiffrement d'un virus polymorphe. Nous avons également alors rappelé plusieurs résultats théoriques concernant la complexité de la détection des virus polymorphes et métamorphes.

Il en ressort que l'analyse en boîte blanche d'un moteur de détection virale peut se fonder sur ce cadre théorique, qui se révèle particulièrement pertinent lorsqu'il s'agit de qualifier un algorithme de détection en termes de :

- performance, en temps et espace mémoire nécessaires à l'exécution de l'algorithme de détection ;
- correction et complétude vis-à-vis d'une classe de transformations d'obfuscation, dès lors que l'on est capable de décrire le langage produit par un ensemble de transformations d'obfuscation et de démontrer que l'automate correspondant au schéma de détection est en mesure de reconnaître tous les mots du langage, et seulement ces derniers.

Il est à noter que cet outil théorique ne s'applique pas qu'aux moteurs de détection fondés sur une recherche de motifs (ou d'expression rationnelles), mais peut également s'appliquer naturellement aux moteurs de détection fondés sur une approche structurelle, comme par exemple [BMM06, BKM08].
À titre d'exemple, on trouve dans le papier [BKM08] une formulation du problème de détection sous-jacent en terme de reconnaissance du langage produit par une grammaire formelle par un automate à état finis. Cette présentation pourrait être complétée par une preuve de résistance du schéma de détection à certaines transformations d'obfuscation du flot de contrôle et une présentation des limites du schéma de détection (vis à vis d'autres transformations d'obfuscation).

[3] le motif de détection est représenté par le langage $L_{\mathcal{V}}$, supposé produit par une grammaire $G_{\mathcal{V}}$ et la fonction de détection $f_{\mathcal{V}}$ est un automate à états finis, capable de reconnaître le langage $L_{\mathcal{V}}$

### Cadre de l'interprétation abstraite

La modélisation par interprétation abstraite permet de capturer les aspects sémantiques de l'information extraite d'un programme, ainsi que de valider la correction et l'efficacité des transformations imposées à cette information (i.e. l'efficacité du processus d'élaboration d'un schéma de détection fondé sur des informations sémantiques). Elle permet en particulier de formaliser l'analyse statique et dans une certaine mesure la rétro-ingénierie par utilisation de méthodes d'analyse dynamique ou hybride statique/dynamique. Nous avons présenté et étudié ce formalisme dans le chapitre 4. Dans ce cadre théorique, nous pouvons considérer les critères suivant [DPCJD07] :

| CRITERE | DESCRIPTION |
|---|---|
| correction | un schéma de détection $(\alpha(S[\![V]\!]), \alpha)$[4] est dit correct vis-à-vis d'un ensemble de transformations d'obfuscation $\mathbb{O}$ si et seulement si : $$\exists \alpha_r : \alpha(S[\![V]\!]) \subseteq \alpha \circ \alpha_r(S[\![P]\!]) \Rightarrow \exists \mathfrak{o} \in \mathbb{O} : P \text{ infecté par } \mathfrak{o}(V).$$ |
| complétude | un schéma de détection $(\alpha(S[\![V]\!]), \alpha)$ est dit complet vis-à-vis d'une transformation d'obfuscation $\mathfrak{o}$ si et seulement si : $$P \text{ infecté par } \mathfrak{o}(V) \Rightarrow \exists \alpha_r : \alpha(S[\![V]\!]) \subseteq \alpha \circ \alpha_r(S[\![P]\!]).$$ |

Remarquons que des travaux récents [DPCJD07] ont permi de modéliser un moteur de détection [CJS+05] dans le cadre de l'interprétation abstraite et de prouver dans ce cadre la résistance et la complétude de leur schéma de détection, sous certaines hypothèses[5]. Ils ont complété cette démonstration par une preuve des limites de leur schéma de détection au regard de certaines transformations d'obfuscation.

### Cadre probabiliste

Une modélisation statistique du problème de détection offre une vision complémentaire et s'applique à l'analyse de comportements viraux, sur la base d'informations statistiques. Elle permet notamment de capturer les aspects liés aux interactions du programme avec son environnement. Ce formalisme a été présenté et étudié dans le chapitre 7. Dans ce cadre théorique, nous pouvons considérer les critères suivant [FJ07] :

| CRITERE | DESCRIPTION |
|---|---|
| correction | un schéma de détection $(\lambda_{\mathcal{V}}, f_{\mathcal{V}})$[6] est dit correct vis-à-vis d'un ensemble viral $\mathcal{V}$ lorsque $\beta$, le risque de fausse alarme, est suffisamment proche de zéro. |
| complétude | un schéma de détection $(\lambda_{\mathcal{V}}, f_{\mathcal{V}})$ est dit complet vis-à-vis d'un ensemble viral $\mathcal{V}$ lorsque $\alpha$, le risque de non détection, est suffisamment proche de zéro. |

Les tests statistiques et la notion de simulabilité des tests constituent un outil théorique naturellement adapté à la modélisation d'un moteur de détection, en particulier lorsque celui-ci met en oeuvre des méthodes de classification probabilistes basées sur des algorithmes d'apprentissage, comme c'est le cas des moteurs de détection fondés sur l'utilisation de modèles bayésiens (réseaux bayésiens dynamiques par exemples [SEZS01, AHH+99, WS06]). Nous

[4] le motif de détection représente une information utile $\alpha(S[\![V]\!])$ extraite de la sémantique $[\![V]\!]$ du virus $V$ et la fonction de détection est une fonction d'abstraction $\alpha$ qui extrait l'information sémantique utile des programmes

[5] De manière à pouvoir raisonner indépendamment de la résolution de certains problèmes réputés difficiles de la théorie de la complexité ou de l'optimisation des programmes, les auteurs ont introduit une définition de la complétude et de la correction d'une fonction de détection qui prend en compte l'invocation d'un ou plusieurs oracles, supposés parfaits. Ils considèrent en particulier que certaines réponses à des questions difficiles sont fournis par des oracles, comme le problème de l'extraction du graphe CFG d'un programme ou la détection de junk code

[6] le schéma de signature est représenté par la distribution d'une loi de probabilité caractérisant l'information ou par un modèle $\lambda_{\mathcal{M}}$ formé sur des données d'apprentissage et la fonction de détection est une règle de décision générée sur la base de données d'apprentissage, permettant d'évaluer la vraisemblance d'une observation, étant donné le modèle $\lambda_{\mathcal{M}}$

avons illustré l'utilisation de tels modèles dans le cas de l'analyse spectrale (cf. chapitre 7), et démontré que la notion de simulabilité des tests statistiques permet de rendre compte de la robustesse ou rigidité d'un schéma de détection, les risques de première et seconde espèce associés à la définition d'un test permettant de rendre compte naturellement des notions de correction et de complétude.
Ces méthodes de détection s'appliquent naturellement à la reconnaissance de programmes poly/métamorphes, par le biais de l'utilisation d'algorithmes de recherche de similarité et de classification par souches virales.
Nous avons constaté que la simulabilité des tests statistiques revient à exploiter l'imprécision du modèle de détection.
Nous pouvons définir de la manière suivante l'imprécision d'un schéma de détection :

| CRITERE | DESCRIPTION |
|---|---|
| imprécision | l'imprécision d'un schéma de détection $(\lambda_{\mathcal{V}}, f_{\mathcal{V}})$ est donnée par l'entropie relative entre la vraie distribution $p_Y$ de la variable $Y$ et la distribution $p_Y^{\lambda_{\mathcal{V}}}$ de la variable $Y$ sous le modèle de détection : $$d_{KL}(p_Y \| p_Y^{\lambda_{\mathcal{V}}}) = \sum_y p_Y(y) \log \frac{p_Y(y)}{p_Y^{\lambda_{\mathcal{V}}}(y)} = \varepsilon > 0.$$ |

L'imprécision d'un schéma de détection rend compte de l'entropie relative entre la distribution réelle d'une famille virale et le modèle de détection construit sur la base des données d'apprentissage.
La rigidité d'un schéma de détection sera d'autant plus grande que l'imprécision du modèle est proche de zéro.
Nous pouvons observer que la rigidité (ou l'imprécision) d'un schéma de détection sont des critères intrinsèques aux modèles de détection considérés, c'est-à-dire qu'ils ne sont pas relatifs à la population virale considérée.

Ce formalisme s'applique également (quoique de manière duale) à l'analyse des virus furtifs par contrôle de modèle (cf. chapitre 6). Dans ce cas, nous pouvons également caractériser la méthode de détection (et de manière duale le système furtif) en terme de simulabilité des tests statistiques sous-jacents.

Remarquons que cette approche est également appliquable dans d'autres domaines de la détection d'intrusion, comme le filtrage réseau ou les systèmes de détection/protection face à une intrusion, où cette fois l'approche est un peu différente, dans la mesure où elle s'attache à détecter tout comportement suspect, définit comme s'écartant d'un comportement moyen jugé a priori conforme à une politique de sécurité.
Si la fonction de sécurité est différente, les méthodes d'extraction d'information et d'élaboration des schémas de détection sont très similaires, la principale différence résidant dans le type et le nombre de bases de connaissances utilisées par la fonction de détection pour fonder sa décision.
Nous espérons donc que notre méthode d'évaluation et partie des outils et critères théoriques et techniques peuvent être adaptés sans trop d'effort à l'analyse de ce type de produits de sécurité.

### 8.3.2 Analyse boîte blanche du moteur de détection

La conception d'un moteur de détection comprend plusieurs parties, qui peuvent être qualifiées sur la base de ces critères théoriques :

| ALGORITHME | CRITERE |
|---|---|
| Unpacking | Critères cryptographiques WBAC : diversité et ambiguïté. Cryptanalyse d'une génération environnementale de clé. Complétude de l'analyse dynamique. |
| Représentation intermédiaire | Correction de l'analyse statique, Complétude de l'analyse dynamique. |
| Réécriture | Confluence, terminaison d'une stratégie de réécriture |
| Schéma de détection | Simulabilité, rigidité, correction, complétude |

Remarquons que ces parties ne sont pas toutes nécessairement présentes. À titre d'exemple, un moteur de détection fondé sur une simple recherche de motifs ne comporte pas d'algorithme dédié à l'extraction et à la normalisation d'une représentation intermédiaire.
Dans le cas des moteurs d'analyse spectrale étudiés au chapitre précédent, il est possible d'identifier une étape d'extraction d'une représentation intermédiaire (un modèle représentatif de la distribution des instructions assembleur) puis une réécriture de cette représentation, lors de la phase d'entrainement du modèle.

Le moteur de détection doit permettre la suppression du loader sécurisé, lorsque le virus a été protégé par utilisation d'un packer du marché. Nous avons vu que plusieurs algorithmes dynamiques ont été proposés pour opérer la suppression automatique du loader de protection d'un exécutable packé. Ces algorithmes souffrent tous de limitations importantes, et doivent se fonder sur des heuristiques pour retrouver le point d'entrée original de l'exécutable packé. Une méthode classique consiste à identifier le packer utilisé, en se fondant sur une base d'information. Le processus de détection virale commence alors par une phase de détection des packers ! Dans ces conditions, il est évident de définir un critère de complétude de cette base d'information : celle-ci doit permettre d'identifier tous les packers du marché susceptibles d'être utilisé par un concepteur de virus.

La représentation intermédiaire du programme peut être le code assembleur, ou la représentation du programme sous forme de graphe CFG. Il peut s'agir d'une représentation plus complète du flot de contrôle permettant par exemple de raisonner sur le flot de données d'un programme comportant du code auto-modifiable. Une telle représentation a été proposée dans [AMD06] et appelée *State-Enhanced* CFG (SE-CFG). Constatant que la représentation sous forme de graphe CFG traditionnelle ne permet pas de mener une analyse statique suffisament précise du flot de données, les auteurs ont proposé une représentation étendue du graphe de flot, permettant de s'affranchir de l'hypothèse d'un code constant. Ce modèle est présenté de manière plus détaillée au chapitre 3, section 3.3.3. Le graphe SE-CFG peut être linéarisé, c'est-à-dire que partant de la seule information du graphe SE-CFG d'un programme comportant du code auto-modifiable, il est possible de reconstruire la section de code d'origine. Le graphe SE-CFG contient une information supplémentaire permettant d'améliorer la précision de l'analyse du flot de données, et donc d'améliorer le processus de normalisation de cette représentation intermédiaire.
Ces exemples illustrent l'importance des critères de correction et de complétude associés à la représentation intermédiaire du code choisie (le choix d'un graphe CFG traditionnel comme représentation intermédiaire du programme est aussi conservatif que celui d'un graphe SE-CFG, mais l'abstraction correspondante est moins précise). Elle conditionne l'efficacité de l'étape suivante, visant à obtenir une représentation intermédiaire normalisée par application d'une statégie de réécriture. L'application de ces transformations de réécritures est généralement gouvernée par le résultat d'une analyse du flot de données. Nous pouvons observer que le choix d'une représentation intermédiare influe notamment sur la complétude de l'analyse du flot de données.

La stratégie de réécriture peut se dispenser complétement des résultats d'une analyse du flot de données. C'est le cas des transformations visant à supprimer les sauts inutiles d'un graphe CFG, que l'on trouve implémentées dans plusieurs moteurs de détection récents [CJS+05, BMM06] proposant une méthode de détection dit structurelle ou morphologique, fondée sur la recherche d'un sous-graphe viral. Une telle stratégie de réécriture est efficace au regard d'une classe de mécanismes d'obfuscation du flot de contrôle. Elle s'avère plus limitée face à des transformations d'obfuscation plus évoluées, permettant de fusionner les flots de contrôle du virus et du programme hôte [Z0m00b] ou de rendre les sauts inutiles plus difficiles à supprimer, par utilisation de prédicats opaques par exemple. Remarquons que les moteurs viraux n'implémentent pas encore cette technologie.
L'utilisation de sélecteurs de similarité [DR05] pour construire un isomorphisme de graphe CFG paraît à ce dernier égard correspondre à une stratégie de réécriture plus complète, bien que fondée sur l'application d'heuristiques pouvant entâcher sa correction.

Lorsque la stratégie de réécriture se fonde sur une analyse du flot de donnée, la correction et la complétude des algorithmes correspondant jouent naturellement un rôle primordial. Nous avons vu au chapitre 4 que le cadre formel de l'interprétation abstraite se prête naturellement à la caractérisation des méthodes d'analyse du flot de donnée en terme de correction et de complétude.

La stratégie de réécriture elle-même peut être qualifiée au regard de critère tels que la confluence et la terminaison. Les auteurs de [BKM08] proposent une représentation sous forme de termes d'un graphe CFG mais ne définissent pas leur statégie de réécriture de graphe CFG dans ce cadre formel. Ils évoquent cependant la possibilité d'utiliser les résultats d'une théorie de la réécriture de termes commutative et associative pour améliorer la correction de leur stratégie de réécriture et l'algorithme d'isomorphisme de graphe CFG en prenant en compte des aspects commutatifs, comme par exemple la permutation des arcs d'un saut conditionnel.

## 8.4 Conformité et robustesse de l'implémentation

Nous avons présenté dans la section précédente des outils théoriques permettant de qualifier la robustesse d'un moteur de détection, concernant principalement la définition des schémas de détection, leur stockage en base et la définition d'une fonction de détection. Le parti-pris est le plus souvent d'extraire du périmètre de cette modélisation le processus d'extraction d'information, supposé fiable. Au regard de la menace virale actuelle, concernant en particulier l'utilisation de méthodes de blindage logiciel et de furtivité de plus en plus sophistiquées, ce parti pris peut paraître discutable. Remarquons qu'il est cependant utile d'effectuer cette séparation logique, notamment à des fins de simplification.

Le processus d'extraction d'information est pourtant primordial dans la définition des schémas de détection ou parfois l'apprentissage des modèles définissant la fonction de détection (c'est en particulier le cas des fonctions de détection conçues comme des automates à états finis ou des modèles statistiques, et formés sur des données d'apprentissage).
Nous avons choisi de ne pas éluder ce point technique, en se dotant d'outils et de critères techniques permettant de valider la conformité et la robustesse de cette fonction indispensable au bon fonctionnement d'un moteur de détection.

### 8.4.1 Outils et critères techniques

Les processus de rétro-ingénierie et de détection de code viral correspondent à des problématiques différentes, même si une grande partie des tâches automatisables est commune. Nous avons traité dans la seconde partie de cette thèse la double problématique de l'analyse et de la détection de code viral. Nous avons pris le parti de :

- focaliser sur les méthodes dynamiques d'extraction d'information fondées sur une émulation du matériel supportant le système d'exploitation.
- ne pas négliger les problèmes de correction et de complétude des opérations d'unpacking, désassemblage et extraction du graphe de flot, dont la bonne réalisation est un pré-requis indispensable lorsque l'on souhaite valider par le test l'efficacité de l'implémentation d'un algorithme de détection.

Les problèmes du désassemblage et de l'extraction du graphe CFG d'un exécutable ont été traités dans le chapitre 3. Le problème de l'unpacking d'un exécutable a été présenté dans le chapitre 5.

Nous avons abordé dans le chapitre 6 le problème de récupération d'informations d'interactions bas niveau avec l'OS. Nous avons défini un ensemble de techniques de récupération d'information d'intégrité et de cohérence de certaines parties du système d'exploitation par utilisation d'un émulateur de la plate-forme matérielle supportant le système d'exploitation. L'outil VxStripper, qui implémente ces heuristiques et techniques forensiques, permet de surveiller les composants critiques du système d'exploitation et de récupérer automatiquement de l'information sur un virus furtif ou tout autre programme interagissant à un bas niveau avec l'exécutif du système d'exploitation.
Le processus de récupération d'informations est fiable et sécurisé. Cette information est précieuse tant pour l'élaboration de schéma de détection caractérisant ce type d'infection informatique que pour l'analyse en boîte noire d'un produit antivirus.

Nous pouvons remarquer qu'il est difficile de définir des critères précis concernant la fiabilité du processus d'extraction d'un moteur de détection virale, en particulier lorsque cette analyse ne peut pas être menée en boîte blanche (sur la base des spécifications ou des informations de conception). Nous espérons pouvoir mettre en oeuvre un tel processus de manière optimale, sans être entravé par les contraintes d'efficacité en terme de consommation de ressources inhérentes à un produit antivirus commercial. Sur la base de cette fonction d'extraction d'information de référence, nous pouvons définir un jeu de tests permettant de qualifier en boîte noire la robustesse de la fonction d'extraction d'information du moteur de détection cible, par simple comparaison.

## 8.5 Plate-forme de tests

### 8.5.1 Tests retenus

Nous avons procédé à l'analyse de plusieurs produits anti-virus sur étagère, choisis en raison de leur notoriété ou du poids de l'éditeur sur le marché du logiciel. Nous avons progressivement dégagé un ensemble de tests suffisamment discriminants pour permettre une analyse comparative des produits.

Les tests mis au point sont destinés à valider les fonctions de sécurité d'un produit anti-virus au regard des exigences de sécurité définies dans le profil de protection. Ces exigences portent soit sur les fonctions de sécurité du produit anti-virus, soit sur son environnement. Des exigences supplémentaires, relatives aux composants d'assurance, permettent de valider la qualité du développement logiciel et de la veille technologique de l'éditeur de produit anti-virus.

### 8.5.2 Description de la plate-forme de test

Le tableau suivant décrit la plate-forme de test en termes d'architecture matérielle, de système d'exploitation et de logiciels :

| | |
|---|---|
| Matériel | PC Intel ou AMD, comportant au moins un port USB[7]. |
| Système d'exploitation hôte | Windows 2K/XP/Vista |
| Logiciels de virtualisation | VxStripper, VMware [VMw06] |
| Système d'exploitation invité | Windows XP SP2 |
| Sondes réseaux | module VxStripper, ethereal |
| Instrumentation des APIs | module VxStripper, MS. `detours` [BH99] |
| Analyse statique de code | IDA Pro, HT Editor |
| Analyse dynamique | VxStripper, IDA Pro, Ollydbg |
| Analyse forensique | module VxStripper, Runtime's DiskExplorer for NTFS [Run08] |
| Outils de diagnostic système | outils Windows Driver Devel Kit (notamment `fltmc` et `devcon` [BB97, Mic06a]), outils MS. SysInternals |
| Autres outils | Outils spécifiques de test : alternate data stream NTFS, chemins longs NTFS, client/serveur d'injection à distance de shellcodes viraux, LSD [LSD02], Metasploit, conformité du moteur de filtrage des scripts (eSafe [eSa06]), rootkits (`fu`, `klog`, hook GLLK, etc.) |

La préparation de la base virale de test nécessite les outils suivants :

| | |
|---|---|
| Packers | Armadillo, Asprotect, Petite, Shrinker, UPX, yC, etc. |
| Outils de compression et d'archivage | outils GNU Win32 (formats ARC, ARJ, BASE64, BZIP2, GZIP, LHA, LZO, MIME, MS-CABINET, MS-COMPRESS, RAR, TAR, UPX, UUE et ZIP) |
| Outils d'unpacking | module VxStripper, Import Reconstructor, Relox |
| Autres outils | Outils spécifiques de test : moniteur temps-réel pour la maintenance de la base virale, module VxStripper de gestion de la collection de virus, corruption des formats ZIP et UPX (`upxredir`), système de ré-écriture de binaires/obfuscateur, VGKs |

Nous avons positionné nos tests anti-virus relativement à un contexte d'attaque fixé : les techniques ont été étudiées sur une plate-forme PC Intel/MS. Windows, en raison de la large diffusion de cette plate-forme. Concernant le paramétrage de l'environnement, le système d'exploitation doit être installé depuis peu et mis à jour. Aucun autre antivirus ne doit être installé.

[7] Les périphériques : lecteur de disquette, lecteur CDROM, carte réseau peuvent être émulés dans la machine virtuelle sans être présents sur la machine hôte. Ce n'est pas le cas du périphérique USB. Une carte réseau est cependant nécessaire si l'on souhaite évaluer la fonction de mise à jour du produit et/ou de ses signatures virales.

### 8.5.3 Tests

Nous décrivons dans cette section le plan de test permettant de valider les exigences de sécurité portant sur les fonctions de sécurité d'un produit anti-virus, son environnement et les composants d'assurance relatifs à sa conception. Nous présentons de façon résumée les tests, qui sont par ailleurs décrits de manière plus précise dans l'annexe A.

#### Installation et paramétrage du produit (environnement d'exploitation)

La première tâche de l'évaluateur consiste naturellement à « prendre en main » le produit cible après l'avoir installé et démarré. Cette tâche est effectuée en suivant les instructions de la documentation des procédures de mise en service de la cible d'évaluation et de la documentation d'exploitation, pour chaque rôle défini.
Cette première étape est l'occasion de récupérer de nombreuses informations sur les choix techniques en termes d'architecture, de conception et d'implémentation. Le jeu de tests TST_INSTALL décrit la démarche que je propose pour automatiser le plus possible la collecte des informations lors de l'installation et du démarrage du produit cible. Je décrit également dans ce jeu de tests quelques méthodes permettant d'éprouver la robustesse de l'installation des pilotes filtres ou de périphériques, dont le bon fonctionnement et la stabilité sont cruciaux pour la réalisation des fonctions de sécurité du produit.
À l'issue de cette tâche, il est possible de mettre en regard l'information collectée sur l'implémentation du produit et les documents de conception et d'architecture du produit cible. Rappelons qu'un document d'architecture doit décrire le produit en terme de sous-systèmes et de modules (identification, interactions, interfaces). L'évaluateur doit être en mesure de se convaincre de toutes les correspondances entre la représentation de l'implémentation et les spécifications fonctionnelles.

La procédure de mise en service d'un produit antivirus comprend notamment une phase de paramétrage, lors de laquelle les caractéristiques des fonctions de sécurité (ou des sujets/objets qui leur sont soumis) sont modifiées. Cette phase offre une vue d'ensemble des fonctionnalités de sécurité, de leurs interfaces et de leurs paramétrages. Elle permet également de valider le fonctionnement correct de la fonction d'audit, qui doit journaliser les événements relevants de la sécurité. Le test TST_AUDIT décrit la démarche que je propose pour éprouver la robustesse de la fonction d'audit vis-à-vis de la modification ou de la destruction des données d'audit stockées.

La procédure de mise en service d'un produit antivirus comprend une mise à jour immédiate, puis ordonnancée, de la base des signatures virales et des moteurs de détection. Le test TST_UPDATE décrit la démarche que je propose pour automatiser la collecte des informations lors de la mise à jour du produit. Le test TST_INTEG est destiné à éprouver la robustesse de la base des signatures virales, vis-à-vis d'attaques ayant pour but d'en modifier l'intégrité.

| CATÉGORIE | TESTS | RÉSULTATS ATTENDUS |
|---|---|---|
| Installation et mise en service du produit | TST_INSTALL | Un instantané de l'installation du produit antivirus et une trace de ses interactions avec plusieurs des composants du système d'exploitation. Une idée de la robustesse du produit antivirus, concernant sa pile de pilotes. |
| | TST_AUDIT | La fonction d'audit doit être conforme au profil de protection, concernant le type d'événements et l'information qui doit être enregistrée. Les mécanismes de protection en disponibilité et en intégrité des données d'audit stockées doivent être efficaces. |
| | TST_UPDATE | La fonction de mise à jour à la demande ou ordonnancé du produit antivirus doit s'appuyer sur des mécanismes et protocoles cryptographiques permettant d'assurer sa sécurité. |
| | TST_INTEG | Les mécanismes assurant la protection de la base des signatures virales doivent être robustes vis-à-vis d'attaques visant à atteindre à son intégrité. |

Ces tests sont décrits en détail dans l'annexe A.1.

#### Tests de conformité des fonctions de sécurité

La seconde tâche de l'évaluateur consiste à tester les fonctions de sécurité sur la base des spécifications fonctionnelles (et à documenter les résultats). L'objectif est de fournir des éléments de preuve montrant la couverture des spécifications fonctionnelles par les tests.
Remarquons qu'un certain nombre de tests de conformité doivent être effectués lors de la mise en service du produit. L'application de notre plan de test permet de ne pas avoir à ré-effectuer les procédures de mise en service.
Le test des interfaces de tous les sous-systèmes et de tous les comportements décrits dans la documentation de développement du produit ne doit pas être obligatoirement exhaustif. L'objectif est seulement que l'évaluateur soit en mesure de comprendre précisément les aspects des fonctions de sécurité qui ont été testés. L'évaluateur va repasser un échantillon des tests passés par le développeur, puis concevoir et passer des tests indépendants de conformité des fonctions de sécurité sur la base des spécifications fonctionnelles.
J'ai spécifié un ensemble de tests permettant de valider la conformité et la fiabilité de la fonction de détection concernant :

– l'analyse des systèmes de fichiers (TST_FS) :

| CATÉGORIE | TESTS | RÉSULTATS ATTENDUS |
|---|---|---|
| Efficacité de la fonction d'analyse du système de fichiers | *Alternate Data Streams* NTFS (TST_FS.1) | Le produit antivirus est capable de détecter un code viral caché dans un flux NTFS |
| | Chemins longs NTFS (TST_FS.2) | Le produit antivirus est capable de détecter un fichier dont le nom dépasse les 260 caractères |
| | Analyse sur plusieurs systèmes de fichiers (TST_FS.3) | Le produit antivirus détecte l'infection sur tous les supports (FAT, NTFS, CIFS, CDFS) et notifie l'utilisateur |
| | Analyse des *clusters* défectueux (TST_FS.4) | Le produit antivirus détecte un code viral dissimulé dans un *bad cluster* du système de fichier NTFS |

– l'analyse des mails entrants et sortants (TST_MAIL) :

| CATÉGORIE | TESTS | RÉSULTATS ATTENDUS |
|---|---|---|
| Analyse des mails | Envoi et réception d'un mail comportant un fichier viral en pièce jointe | Le produit antivirus détecte (et désinfecte ou supprime) la pièce jointe et notifie l'utilisateur |

– l'analyse des formats de documents et d'archives (TST_FMT) :

| CATÉGORIE | TESTS | RÉSULTATS ATTENDUS |
|---|---|---|
| Support et gestion des formats | Format d'archive, format d'archive corrompu (TST_FMT.1) | Le produit antivirus détecte le fichier sur la majorité des archives et notifie l'utilisateur |
| | Virus de macro (TST_FMT.2) | Le produit antivirus détecte l'exécution d'un script seul, l'exécution d'un script intégré à un document MS Office et notifie l'utilisateur |

– l'analyse de la mémoire (TST_MEM) :

| CATÉGORIE | TESTS | RÉSULTATS ATTENDUS |
|---|---|---|
| Analyse de la mémoire, exploitation de vulnérabilité mémoire | Exploitation d'un débordement de tampon, en local et via le réseau (TST_MEM.1) | Le produit antivirus détecte le débordement de tampon |
| | Analyse de la mémoire (TST_MEM.1) | Le produit antivirus détecte la présence d'une infection informatique en mémoire |

À ces tests s'ajoute un jeu de tests (TST_EFF) permettant de valider
– l'efficacité de la fonction de détection au regard de transformations d'obfuscation,
– et la complétude de la base des signatures virales :

| CATÉGORIE | TESTS | RÉSULTATS ATTENDUS |
|---|---|---|
| Efficacité de la fonction de détection | Base virale (TST_EFF.1) | Taux de détection global (virus toutes plates-formes) |
| | Exécutables packés (TST_EFF.2) | Le produit antivirus détecte la totalité des exécutables packés |
| | Exécutables polymorphes, *Entry Point Obscuring* (TST_EFF.3) | Le produit antivirus détecte toutes les variantes des virus modifiés (par application de techniques élémentaires d'EPO en particulier) |

La méthode d'extraction de schéma de détection [Fil06], dont le principe est rappelé dans la section **??**, permet de compléter la qualification de la fonction de détection et de la base des signatures virales.

La méthode de validation de l'efficacité de la fonction de détection [FJL07, JFD08], fondée sur un moteur de polymorphisme fonctionnel, permet de qualifier la robustesse de la fonction de détection vis-à-vis de ce type de transformations.

Constatant que la plupart des éditeurs de produits antivirus proposent un module spécialisé de détection des *rootkits*, j'ai spécifié un jeu de tests permettant de valider l'efficacité de cette fonction de détection (TST_RK) :

| CATÉGORIE | TESTS | RÉSULTATS ATTENDUS |
|---|---|---|
| Détection des rootkits | Méthodes conventionnelles (*global hook*) et non conventionnelles (*in-line patching*, IAT/EAT, etc.) d'instrumentation des fonctions d'API en espace utilisateur (TST_RK.1.1 et TST_RK.1.2) | Le produit antivirus détecte l'installation et la présence de hooks sur les fonctions de l'API |
| | Méthodes conventionnelles et non conventionnelles pour exécuter du code viral en mode superviseur (TST_RK.2.1 et TST_RK.2.2) | Le produit antivirus détecte l'exécution de code viral en mode superviseur |
| | Détection de la corruption des objets du noyau - liste doublement chaînée des DRIVER_OBJECT et/ou EPROCESS, IDT, SSDT, Sysenter, DRIVER_OBJECT Major Function IRP Hook (TST_RK.3.1 et TST_RK.3.2) | Le produit antivirus détecte la manipulation directe des objets du noyau et notifie l'utilisateur |

Ces tests sont décrits en détail dans l'annexe A.2.

#### Analyse de vulnérabilité

La troisième tâche de l'évaluateur consiste à rechercher de manière méthodique, répétable et systématique les vulnérabilités potentielles du produit antivirus pouvant être exploitées par un attaquant.

## 8.6 Conclusion

Nous avons proposé dans ce chapitre une démarche et des outils permettant à un particulier ou à une entreprise d'obtenir un certain niveau d'assurance lorsqu'il choisit un produit anti-virus sur étagère.

L'utilisation conjointe d'un profil de protection (validé par la démarche méthodologique des critères communs) et la spécification de tests permettant de rendre compte de la couverture des exigences de sécurité nous semble être une base saine pour mesurer la confiance que l'on peut accorder à un produit anti-virus.
La principale limitation induite par cette approche est l'interprétation objective des tests. Même si les tests de notre plate-forme sont automatisés, les résultats sont parfois d'abord prévus pour permettre une étude plus approfondie des rouages internes du produit. Le verdict n'est pas un verdict binaire réussite/échec, mais plutôt un commentaire élaboré de l'analyste de sécurité sur la manière dont des caractéristiques internes du produit atteignent l'état de l'art dans le domaine de la détection d'intrusion basée sur l'hôte et les exigences de sécurité de cette catégorie de produit.
Notre démarche peut naturellement être optimisée. Le jeu de tests doit évoluer constamment, afin de prendre en compte les évolutions techniques des deux adversaires en présence. Une veille académique et technologique constante doit donc être maintenue, concernant en particulier :

- les évolutions de la menace virale, au niveau technique et algorithmique ;

- les nouvelles techniques et architectures logicielles antivirales ;
- les vulnérabilités liées à la conception et à l'implémentation du produit anti-virus et de son environnement.

Le profil de protection (utilisé comme base méthodologique dans ce chapitre) est également amené à évoluer pour renforcer le niveau des exigences de sécurité et donc permettre une augmentation du niveau de confiance correspondant. Un jeu de tests pertinent devra permette sa mise en oeuvre.

# Conclusion et perspectives

Nous avons développé dans la troisième partie de cette thèse une approche méthodologique et une plate-forme de tests permettant d'évaluer un produit antivirus sur la base de critères objectifs et partagés par une large communauté de professionnels de la sécurité informatique. Fondée sur une cible de sécurité générique, permettant de définir la problématique de sécurité et de spécifier le besoin de sécurité de cette catégorie de produit, notre approche permet de définir le niveau de confiance dont on peut créditer un produit antivirus et de comparer les offres de différents constructeurs sur la base des Critères Communs.

L'analyse de sécurité d'un produit antivirus se fait au regard d'une menace virale identifiée. Dans la première partie de cette thèse, nous avons centré notre étude de la menace sur les mécanismes cryptographiques et sur les aspects techniques liés au blindage logiciel et à la protection contre la rétro-ingénierie. Nous avons en particulier :

- démontré la pertinence de la cryptographie boîte blanche dans le contexte viral,
- défini des axes d'amélioration des implémentations boîtes blanche de l'AES et du DES,
- donné une première contribution dans l'élaboration d'un schéma de chiffrement symétrique fondé sur la génération aléatoire de S-Boxes et adapté au contexte viral.

Utilisés seuls, les mécanismes cryptographiques adaptés au contexte WBAC ne sont pas suffisants pour protéger un programme viral contre la rétro-ingénierie. Nous avons complété notre panorama de la menace virale par l'étude des mécanismes non cryptographiques (que l'on trouve typiquement dans les chaînes de compilation spécialisées dans la protection logicielle) susceptibles d'être utilisés par un virus pour asseoir la robustesse des mécanismes cryptographiques qu'il implémente et parer la rétro-ingénierie. Nous avons en particulier :

- introduit les concepts clés liés à la protection et à la rétro-ingénierie logicielle,
- effectué une revue des critères permettant l'évaluation d'une solution de protection logicielle,
- proposé une démarche pour leur évaluation.

Ayant identifié les menaces à contrer et les objectifs de sécurité à atteindre, le but d'une évaluation est de contrôler, vérifier ou confirmer la conformité des fournitures aux exigences sur le contenu et la présentation des éléments de preuve. Le suivi des vulnérabilité potentielles et la mise en oeuvre de tests de conformité et de robustesse font partie des tâches fondamentales du processus d'évaluation. La couverture des objectifs de sécurité par les fonctionnalités de sécurité est au coeur de cette analyse. Nous avons privilégié l'analyse de la fonction de détection et défini des critères techniques et théoriques appropriés à l'analyse en boîte noire et en boîte blanche (c'est à dire sur la base des spécifications et/ou des informations de conception) du moteur de détection.

Nous avons donc étudié dans la seconde partie de cette thèse les moyens dont dispose la lutte anti-virale pour parer la menace virale. Nous avons étudié en premier lieu le cadre formel de l'interprétation abstraite et proposé une formalisation de la rétro-ingénierie dans ce cadre. Nous avons posé les premières bases d'une réflexion permettant de définir des critères objectifs pour qualifier les transformations de désobfuscation dans le contexte de l'analyse statique et de les étendre au cas de l'analyse dynamique. Nous avons en particulier :

- proposé une définition alternative de l'obfuscation dans le cadre de l'interprétation abstraite et une caractérisation de la désobfuscation en terme de correction et de complétude au regard d'une classe de transformations d'obfuscation,

- étudié les relations entre :
  - la caractérisation d'une transformation de désobfuscation en terme de correction et de complétude vis-à-vis d'une classe de transformations d'obfuscation,
  - ces mêmes notions appliquées à la caractérisation d'un moteur de détection viral.

Concernant la formalisation de la rétro-ingénierie dans un cadre théorique adapté, nous avons constaté que les méthodes dynamiques d'extraction d'information d'un programme protégé sont aujourd'hui encore mal formalisées. Nous avons étudié ensuite une méthode dynamique d'extraction d'information et discuté de son efficacité d'un point de vue technique. Nous avons en particulier développé un outil, appelé VxStripper, dédié à l'analyse de code viral, fondé sur un CPU logiciel. Cet outil est adapté à l'analyse automatique de programmes aussi dangereux que les programme viraux et permet l'instrumentation furtive des API natives et Win32. En nous appuyant sur cet outil, nous avons implémenté un algorithme d'unpacking permettant de récupérer le corps d'un virus protégé et de reconstruire l'exécutable.

Nous avons complété l'exposé des fonctionnalités de VxStripper par la présentation de son module spécialisé dans l'analyse des programmes furtifs. Ce module permet de récupérer d'informations d'intégrité et de cohérence de certaines parties du système d'exploitation depuis l'extérieur de la matrice. Nous avons en particulier étudié la transposition des méthodes de détection des hooks par contrôle de modèle et proposé une modélisation statistique de ces méthodes.

De nombreux prototypes de moteurs de détection implémentent des algorithmes de classification ou d'apprentissage probabilistes utilisés plus traditionnellement dans le domaine de la détection d'intrusion et fondé sur une approche comportementale. Cette algorithmie s'applique à l'analyse de comportements viraux sur la base d'informations statistiques et permet notamment de capturer certains aspects liés aux interactions du programme avec son environnement ou à la distribution du spectre de ses instructions. Pour analyser ce type de détecteurs, nous avons développé une modélisation statistique du problème de la détection virale, permettant de qualifier en boîte blanche les schémas de détection utilisés et de mesurer leur robustesse.
Nous avons en particulier :
- établi que les techniques de détection connues peuvent être modélisées par un ou plusieurs tests statistiques,
- établi que tout test ou ensemble de test peut être contourné, en vertu du principe de simulabilité des tests statistiques,
- proposé une reformulation en termes statistiques le résultat d'indécidabilité de F. Cohen.

Nous avons étudié dans ce cadre formel le cas de l'analyse spectrale par utilisation des réseaux bayésiens au travers de deux exemples :
- la classification « naïve » de Bayes,
- le modèle de Markov caché.

Nous avons donc identifié plusieurs formalismes complémentaires applicables à la modélisation des différents composant d'un moteur de détection :
- la fonction d'extraction d'information ;
- la fonction de normalisation de l'information extraite ;
- la fonction d'élaboration d'un schéma de détection et son stockage dans une base d'information ;
- la fonction de détection.

Nous avons pris le parti de focaliser sur les méthodes dynamiques d'extraction d'information fondées sur une émulation du matériel supportant le système d'exploitation. Notre principale contribution relative au problème d'extraction d'information concerne l'opération d'unpacking. Nous avons en outre défini un ensemble de techniques de récupération d'information d'intégrité et de cohérence de certaines parties du système d'exploitation, permettant d'élaborer des schémas de détection plus élaborés et couvrant globalement ce que l'on rencontre dans les prototypes de moteurs de détection proposés par la recherche antivirale.

Concernant la fonction de normalisation de l'information extraite, nous avons proposé une formalisation de la rétro-ingénierie dans le cadre formel de l'interprétation abstraite qui s'inscrit dans les travaux existant relatifs à

l'optimisation des programmes, et permet de mesurer l'impact en terme de correction et de complétude de ces transformations de programme en terme sémantique. Sur cette base formelle, nous sommes en mesure de qualifier les transformations de désobfuscation appliquées à un programme lors du processus d'extraction d'un schéma de signature.

Nous avons enfin développé une plate-forme de tests dédiée à l'analyse de robustesse des fonctions et mécanismes d'un produit antivirus. L'outil d'analyse virale VxStripper regroupe plusieurs fonctionnalités indispensable à l'analyse de code viral. Il permet notamment de maintenir une base virale exploitable dans le cadre de nos tests et d'extraire une informations très utile à l'analyse des interactions fonctionnelles d'un moteur de détection virale avec son environnement d'exécution, en particulier lorsque cette interaction se fait à bas niveau et ne laisse pas intègre le système d'exploitation, comme c'est le cas par exemple des moteurs de détection des virus furtifs ou des vers.
Nous avons en outre développé plusieurs outils complémentaires et prototypé plusieurs des composants d'un produit antivirus de manière à pouvoir implémenter et étalonner convenablement les tests de robustesse en boîte noire.

L'utilisation conjointe d'un Profil de Protection et la spécification de tests permettant de rendre compte de la couverture des exigences de sécurité nous semble être une base saine pour mesurer la confiance que l'on peut accorder à un produit anti-virus. Nous espérons que la démarche et les outils proposés pourront permettre à un particulier ou à une entreprise d'obtenir un certain niveau d'assurance lorsqu'il choisit un produit anti-virus sur étagère.

## Limites

La principale limite de notre approche méthodologique réside dans le problème de l'interprétation objective des tests. Notre démarche peut sans doute être optimisée dans ce sens. Nous pouvons également nous interroger sur le moyen d'augmenter le faible niveau de confiance associé au Profil de Protection. La spécification de tests de robustesse fondés sur l'utilisation systématique des méthodes formelles permettrait sans doute d'augmenter le niveau d'assurance. En outre, nous avons pu constater que le jeu de tests que nous avons spécifié est fini, et ne couvre que quelques aspects liés à la sécurité d'un produit antivirus. En outre, cet ensemble de test est condamné à évoluer constamment, de manière à prendre compte les évolutions techniques et théoriques liées aux deux parties en présence. Une veille académique et technologique constante doit donc être soutenue.

Concernant la menace virale, nous avons observé, lors de l'étude des mécanismes cryptographiques susceptibles d'être utilisés par un virus pour résister à l'analyse dynamique, que les implémentations boîte blanche actuelles ne sont pas aussi résistantes que leurs homologues en boîte noire (i.e. - robustesse limitée en contexte WBAC en comparaison de la résistance des algorithmes de chiffrement itératifs par blocs en contexte BBAC). Le problème de la conception d'un algorithme de chiffrement à clé privée vérifiant la propriété de la boîte noire virtuelle reste un problème ouvert[8].

Nous avons complété notre panorama de la menace virale par l'étude des mécanismes non cryptographiques susceptibles d'être utilisés par un virus pour asseoir la robustesse des mécanismes cryptographiques qu'il implémente et parer la rétro-ingénierie. Il apparaît que les critères et preuves de sécurité des mécanismes de protection semblent mal adaptés à la mesure de résistance face à une attaque approximative, dynamique ou hybride statique/dynamique. La frontière existante entre les deux disciplines que sont l'analyse statique et l'analyse dynamique est une limitation importante pour l'établissement de critères de plus grande portée pour l'évaluation d'une solution de protection.

Concernant la lutte anti-virale, nous avons étudié le cadre formel de l'interprétation abstraite et posé les premières bases d'une réflexion permettant de définir des critères objectifs permettant de qualifier les transformations de désobfuscation dans le contexte de l'analyse statique et de l'étendre au cas de l'analyse dynamique. Cependant, ce cadre formel ne s'applique actuellement qu'à l'analyse statique. Nous avons étudié les relations entre la caractérisation d'une transformation de désobfuscation en terme de correction et de complétude vis-à-vis d'une classe de transformations d'obfuscation, et ces mêmes notions appliquées à la caractérisation d'un moteur de détection viral. Ici encore, un gros travail reste à faire pour obtenir des critères de rigidité d'un schéma de détection applicables au

[8] Une contribution significative dans ce domaine peut apparaître à certains égards comme le Graal du cryptologue, dans la mesure où cela revient à proposer un nouveau type de cryptosystème asymétrique.

cas d'un moteur de détection fondé sur une analyse dynamique ou hybride.

Nous avons étudié et implémenté dans l'outil VxStripper plusieurs méthodes dynamique d'extraction d'information. Concernant le module d'unpacking, nous observons que les deux étapes de l'analyse peuvent échouer pour les raisons suivantes :

- Détection de l'émulation,
- Conditions de déclenchement environnementale (présence d'un fichier sur le système ou sur un autre ordinateur du réseau, intervalle horaire, ancrage à un matériel spécifique, etc.). L'outils d'analyse de code viral que nous avons développé est impuissant face à des virus dirigés, les conditions environnementales devant être réunies pour que l'algorithme de déprotection automatique de virus packés soit efficace.
- Mécanisme d'altération de la section code de l'exécutable (le lien entre le loader de protection et l'exécutable protégé est alors suffisamment fort pour rendre hasardeuse la suppression automatique du loader de protection).
- Mécanisme d'émulation totale des imports : des heuristiques spécifiques doivent être mises en oeuvre pour réparer l'exécutable. Le fait de pouvoir tracer de manière presque omnisciente les interactions avec les API du système n'apporte pas de solution.

Concernant le module spécialisé dans l'analyse des virus furtifs, nous avons observé que l'émulateur utilisé par VxStripper ne propose pas encore de CPU logiciel supportant les extensions de virtualisation. Il ne permet donc pas l'analyse de certains rootkits VMBR, comme Blue Pill ou Vitriol.

De manière plus générale, l'isolation de l'outil VxStripper peut être améliorée, en inhibant la connexion réseau existant entre les systèmes hôte et invité et en la remplaçant par une fonction d'upload basée sur une manipulation directe du système de fichier. La furtivité de l'environnement émulé peut également être améliorée, en corrigeant en premier lieu les erreurs et imperfections de l'émulation les plus faciles à détecter.

## Travaux futurs

J'identifie plusieurs axes de recherche que je souhaiterais mener en priorité suite aux travaux de cette thèse :

Concernant l'étude des mécanismes cryptographiques susceptibles d'être utilisés par un virus pour résister à l'analyse dynamique, le chapitre 2 apporte une première contribution dans l'élaboration d'un schéma de chiffrement symétrique fondé sur la génération aléatoire de S-Boxes et adapté au contexte viral. Je pense qu'il est possible d'améliorer l'algorithme de génération aléatoire de S-Boxes en étudiant de manière plus approfondie les propriétés de la représentation sous forme ANF d'une fonction booléenne.
Il serait intéressant de spécifier un schéma de chiffrement symétrique spécialement pensé pour être résistant en boîte blanche, tout en assurant une résistance suffisante en boîte noire. Il est possible qu'un schéma itéré ne soit pas un bon choix de conception.
Je pense étudier les possibilités offertes par les schéma de chiffrement par flot. Je souhaite également étudier les possibilités offertes par les grammaires génératrices pour diversifier le code d'une implémentation boîte blanche.

Nous avons posé dans le chapitre 4 les premières bases d'une réflexion permettant de définir des critères objectifs permettant de qualifier les transformations de désobfuscation dans le contexte de l'analyse statique et de l'étendre au cas de l'analyse dynamique. Ce cadre formel ne s'applique actuellement qu'à l'analyse statique. Je souhaiterais étudier les possibilités d'extension des critères de correction et de complétude des transformations de désobfuscation statique aux transformations de désobfuscation dynamique ou hybride statique/dynamique dans le cadre formel de l'interprétation abstraire[9].

Concernant les méthodes dynamiques d'extraction d'information actuellement implémentées dans l'outil VxStripper, je souhaiterais étudier les stratégies de réécriture de code fondées sur le modèle de flot de contrôle SE-CFG, afin d'étudier les méthodes de normalisation d'un programme comportant du code auto-modifiable.

[9] Je pense que cela pourrait contribuer à l'assouplissement de la frontière entre les deux disciplines que sont l'analyse statique et l'analyse dynamique.

Par ailleurs, un gros travail reste à faire pour rendre cet outil d'analyse virale plus efficace et convivial, à commencer par une interface graphique et la définition d'une interface plus modulaire.

Enfin, concernant la méthodologie d'évaluation proposée dans le chapitre 8, je souhaiterais avoir l'opportunité de la mettre en oeuvre dans le cadre de l'évaluation complète d'un produit anti-virus (en temps moins contraint que celui d'une certification de premier niveau).

Annexe A

# Tests de conformité et d'efficacité des fonctions de sécurité

## A.1 Installation et paramétrage du produit

**TST_INSTALL (Installation du produit antivirus)**

**Objectif**
Ce jeu de tests comporte deux parties :
- une partie dédiée à la récupération d'informations sur l'installation du produit anti-virus (interactions avec les API et le registre NT). Ces tests sont réalisés durant l'installation ;
- une partie permettant d'éprouver la robustesse de l'installation face à des attaques virales bas niveau (pile des pilotes de filtrage des systèmes de fichiers, par exemple). Ces tests sont réalisés après l'installation.

**Principe**
Nous avons automatisé la récupération d'informations utiles lors de l'installation du produit par la mise en oeuvre de techniques d'écoute et de contournement des appels aux fonctions d'API. Il existe de nombreuses méthodes autorisant l'injection de code dans l'environnement système ou dans le contexte d'un processus, ainsi que l'interception puis la re-direction des appels de fonctions de l'API Win32 vers le code injecté [BH99, DD03]. Ces techniques de furtivité sont très souvent mises en oeuvre par les chevaux de Troie et les logiciels espions. Ces techniques sont également parfois utiles pour tracer les appels systèmes (en particulier les E/S vers le système de fichier et la base de registre) effectués par un programme viral ou pour rendre inopérantes certaines protections logicielles implémentées par exemple par les virus blindés [Fil05c]. Il est à noter que ces techniques d'instrumentation et d'interception des appels aux fonctions de l'API Win32 sont également appliquées à la détection comportementale d'intrusion virale - cf. DOME, Detection of Malicious Executable [RKLC03], et même à la réalisation de moteur de contrôle d'intégrité temps-réel [Bla04]. Nous utilisons certaines de ces techniques pour tracer automatiquement les appels systèmes effectués lors de l'installation du produit anti-virus, en particulier ceux participant à l'installation des pilotes de filtrage (utilisation de l'outil `detours` [BH99]).

L'outil VxStripper permet de récupérer une information fiable sur les caractéristiques de l'installation du produit antivirus, (concernant en particulier les pilotes de périphériques et le registre du système d'exploitation), et permet de contourner et de caractériser les mécanismes de protection éventuellement implémentés par le produit antivirus (ancrage, mécanismes de furtivité).

Nous éprouvons également la robustesse de l'installation en testant (en essayant de désactiver/décharger ou désinstaller) les pilotes filtres du produit anti-virus (utilisation des outils `fltmc` et `devcon` [BB97, Mic06a]).

*De nombreuses informations peuvent être récupérées en analysant les interactions du processus d'installation avec*

*les composants centraux du système (Kernel API, COM bus, etc.). Des test sophistiqués peuvent être menés, pour contourner le pilote filtre ou mini-filtre en mode noyau du produit antivirus (instrumentation de la pile de pilotes), et donc valider plus complètement la robustesse de l'installation. D'autres techniques peuvent être utilisées pour instrumenter les API. La contrainte principale est que le jeu de test TST_INSTALL reste automatisable et puisse s'appliquer à tout produit antivirus.*

**Analyse des résultats**
À l'issue de la première étape, nous obtenons un instantané de l'installation du produit antivirus et une trace de ses interactions avec plusieurs des composants du système durant le processus d'installation. Après la seconde étape, nous avons une bonne idée de la robustesse du produit antivirus, concernant sa pile de pilotes et ses clés de registres.

### TST_AUDIT (Fichiers d'audit)

**Objectif**
L'objectif de ce test est de valider le fonctionnement correct de la fonction d'audit d'un produit antivirus, au regard du type d'événements et de l'information qui doit être enregistrée. Ce test a pour but d'examiner la robustesse de la fonction d'audit vis-à-vis de la modification ou de la destruction des données d'audit stockées.

**Principe**
Ce test permet en particulier de valider la fonction de rotation des fichiers journaux du logiciel antivirus, lorsque celui-ci impose une taille limite aux fichiers journaux. Le scénario du test est simple : remplissage des fichiers journaux, puis vérification des rotations automatiques.

**Analyse des résultats**
Les événements suivant doivent être enregistrés :
- démarrage/arrêt de la fonction de sécurité d'audit ;
- sélection d'une action par l'utilisateur ;
- action menée en réponse à la détection d'un virus.

La fonction de sécurité d'audit doit enregistrer, pour chaque évènement, au moins les informations suivantes :
- date de l'évènement ;
- type d'évènement ;
- identité du sujet (pour les évènements d'audit résultant de l'action d'utilisateurs identifiés) ;
- résultat (succès ou échec) de l'évènement ;
- action sélectionnée par un utilisateur ou action menée en réponse à la détection d'un virus ;
- description du virus détecté, fichier ou identifiant du processus où le virus a été détecté.

La fonction de sécurité d'audit doit protéger les enregistrements d'audit stockés de toute modification ou suppression par le biais de l'interface de la fonction de sécurité (l'environnement, i.e. le système d'exploitation, est responsable de la prévention de toute modification ou suppression par le biais de l'interface d'administration de l'OS).
La fonction de sécurité d'audit doit fournir les options suivantes pour prévenir la perte des données d'audits (en particulier lorsque les fichiers journaux sont pleins) :
- ignorer les évènements auditables ;
- prévenir les évènements auditables, à l'exception de ceux pris par l'utilisateur autorisé avec certains droits ;
- écraser les anciens enregistrements d'audit sauvegardés.

### TST_UPDATE (Mise à jour du produit)

**Objectif**
Un produit anti-virus peut être mis à jour manuellement ou en utilisant la fonction de mise à jour automatique. Dans le second cas, la mise à jour se fait à la demande ou de manière ordonnancée, à une fréquence paramétrable par l'utilisateur. L'objectif de ce test est de récupérer une information sur le processus de mise à jour du produit antivirus, afin de pouvoir statuer sur son efficacité.

**Principe**
Nous utilisons un analyseur réseau (ethereal) pour tracer le déroulement des mises à jour automatique. Une analyse des trames Ethernet permet de mieux comprendre la manière dont se fait la mise à jour : requête du client, réponse du serveur indiquant la liste des fichiers de définition disponibles, leurs signatures et le nom d'un serveur pouvant les fournir, enfin téléchargement des fichiers de définition, si nécessaire. Une analyse des symboles (strings) de l'exécutable de mise à jour permet de confirmer le fait que le nom du serveur est en dur dans le programme.

L'outil VxStripper permet également de récupérer une information fiable sur la manière dont se fait la mise à jour du produit antivirus (instrumentation des fonctions d'API réseau). Il permet en outre de contourner les mécanismes éventuels de protection logicielle du processus de mise à jour du produit.

**Analyse des résultats**
Nous attendons l'utilisation d'un protocole authentifiant le client, le serveur des mises à jour, éventuellement une authentification mutuelle. Nous attendons (après téléchargement des fichiers de mise à jour de la base des signatures et éventuellement des moteurs d'analyse du produit anti-virus) un contrôle de l'intégrité des fichiers téléchargés. Il peut être nécessaire de redémarrer le système d'exploitation ou le produit anti-virus. L'interface (graphique) doit signaler les modifications apportées à la base des signatures virales. Les événements liés à la mise à jour du produit doivent être journalisés.

**TST_INTEG (Intégrité de la base des signatures virales)**

**Objectif**
L'objet de ce test est d'éprouver la résistance du fichier des signatures. Le fichier de signature du produit est presque toujours écrit dans un format propriétaire au produit. Il ne doit pas être possible de modifier le contenu du fichier.

**Principe**
Nous vérifions la résistance du produit au redémarrage lorsque l'on altère l'intégrité de la base des signatures virales. Le mode opératoire est simple : on tente de corrompre ou d'atteindre l'intégrité de la base des signatures virales. L'objectif est de valider les mécanismes garants de l'intégrité de la base des signatures virales et de contrôler le comportement du logiciel, en terme de reprise sur incident. Nous effectuons plusieurs actions :
- modification de l'attribut : date de dernière modification. Ecriture dans l'un des fichiers de la base des signatures virales ;
- suppression d'un des fichiers de la base des signatures ;
- suppression d'une des entrées de la base des signatures virales ;
- substitution d'une ancienne base à la nouvelle ;
- dans le cas où plusieurs versions de la base des signatures sont présentes dans le répertoire d'installation du produit, supprimer et/ou modifier la version la plus récente.

**Analyse des résultats**
Dans tous les cas, le refus de charger la base doit être accompagné d'une alerte ou notification. Dans le cas où une version saine (mais moins récente) est chargée, le produit anti-virus doit signaler clairement le fait qu'il n'est pas parvenu à recharger la base des signatures virales ou qu'il fonctionne en mode dégradé. Ces tests nous permettent donc de valider les auto-tests d'intégrité de chacun des fichiers de la base des signatures virale, le système de reprise sur erreur et le système des remontées d'alertes du produit anti-virus.

## A.2 Test de conformité des fonctions de sécurité

**TST_FS.1 (*Alternate Data Streams* NTFS)**

**Objectif**
L'objectif est de tester la capacité du produit à détecter du code malicieux caché dans un flux NTFS associé à un fichier ou répertoire. La protection temps réelle désactivée, le programme de test créé un répertoire et y associe le flux ICON contenant la chaîne EICAR. Il créé aussi deux fichiers et y associe respectivement un flux ICON et deux flux, BACKUP et EICAR contenant chacun la chaîne EICAR. Cette suite de caractère sert de référence pour les

tests de détection de produit antivirus. Nous effectuons une analyse sur le répertoire créé par le test et observons la réaction du produit.

**Principe**
Le système de fichier NTFS permet d'associer un ou plusieurs flux de données à n'importe quel fichier ou répertoire d'un volume NTFS. Le nom complet d'un flux `<nom du fichier> :<nom du flux> :<type de flux>`. Ces flux sont invisibles par l'explorateur Windows et l'interpréteur de commande et ne sont pas pris en compte dans le calcul de la taille du fichier. Ils peuvent contenir n'importe quel type de données.

**Analyse des résultats**
Comme les flux NTFS sont une menace connue, on s'attend à ce que le produit détecte du code malicieux dans ces flux. Le produit antivirus doit être capable de détecter un fichier malicieux caché dans un flux NTFS.

**TST_FS.2 (Test des chemins longs NTFS)**

**Objectif**
L'objectif est de tester la capacité du produit à détecter du code malicieux enfoui profondément dans l'arborescence des fichiers (Certains produits antivirus ne sont pas capables de détecter un virus dont le chemin d'accès est trop long [Vig06g]). La protection temps réelle désactivée, le programme de test créer le fichier `c:\lco\c...\c\eicar.exe` contenant la chaîne EICAR. Cette suite de caractère sert de référence pour les tests de détection de produit antivirus. La longueur totale du nom de fichier est choisie. Nous effectuons ensuite une analyse du répertoire `c:\lco` et observons la réaction du produit.

**Principe**
La longueur maximale du nom complet d'un fichier que peut afficher l'explorateur Windows ou la console (cmd.exe) ne peut pas dépasser 260 caractères. Cette limite est héritée des anciennes versions de Windows (9x) qui ne prenaient pas en compte UNICODE. UNICODE est un standard qui code les caractères sur 16bit permettant ainsi de représenter virtuellement tous les symboles de toutes les langues du monde. Cependant, les versions NT de Windows prennent nativement en charge UNICODE et la limite maximum d'un nom de fichier a été repoussée à 32 767 caractères. Il est alors possible de créer par programmation un fichier de plus de 260 caractères en utilisant les versions UNICODE des fonctions de création de fichier. Le fichier résultant ne peut pas être supprimé par l'utilisateur car il excède la limite des 260 caractères.

**Analyse des résultats**
On s'attend à ce que le produit détecte un fichier malicieux ayant un nom supérieur à 260 caractères.

**TST_FS.3 (Scan sur plusieurs systèmes de fichiers)**

**Objectif**
L'objectif est de tester la capacité du produit à détecter du code malveillant placé sur différents systèmes de fichiers.

**Principe**
Pour cela, la protection temps réelle désactivée, un fichier contenant la chaîne EICAR est placé sur une disquette (FAT), sur le disque local (NTFS), sur un partage réseau (CIFS) et sur un CD-ROM (CDFS). Nous analysons ensuite les répertoires de test et observons le comportement du produit.

**Analyse des résultats**
On s'attend à ce que le produit détecte un tel code sur tous ces systèmes de fichiers et notifie l'utilisateur.

**TST_FS.4 (Secteurs défectieux NTFS)**

**Objectif**
L'objectif est de tester la capacité du produit à détecter du code malicieux caché dans de « vrai-faux » secteurs défectueux d'une partition NTFS. La protection temps réelle activée, nous modifions directement les structures

NTFS sur le disque pour créer de faux *bad clusters* et y cacher la chaîne EICAR. Cette suite de caractère sert de référence pour les tests de détection de produit antivirus.

**Principe**
Une partition NTFS est décrite sur le disque par plusieurs fichiers de méta données référencés dans la table principale de fichiers, la *master file table* - MFT, mais invisible à l'utilisateur. Chaque fichier possède un certain nombre d'attributs dont un attribut de données. Le fichier `$Bitmap` garde trace de tous les clusters alloués sur le volume. Le fichier `$BadClus` garde trace de tous les secteurs défectueux sur le volume. Nous utilisons le logiciel Runtime's Disk Explorer for NTFS [Run08] permettant de modifier directement les structures NTFS pour allouer un cluster (modification de l'attribut `$Data` du fichier `$Bitmap`) et le placer dans la liste des *clusters* défectueux (modification de l'attribut `$Data` du fichier `$BadClus`). Nous écrivons ensuite la chaîne EICAR dans le *cluster* ainsi marqué.
L'outil VxStripper propose également cette méthode forensique pour cacher un code viral dans les secteurs défectueux du système de fichier (du disque virtuel). La méthode est automatisée et sa mise en oeuvre furtive.

**Analyse des résultats**
Cette technique étant une technique avancée et peu commune dans les programmes viraux standards, on ne s'attend pas nécessairement à ce qu'un produit antivirus détecte du code malicieux placé dans un « vrai-faux » bad cluster. Ce test permet en revanche de valider l'efficacité de la fonction de détection des rootkits proposée par certains produits antivirus et HIPS.

**TST_MAIL (Analyse des mails entrants et sortants)**

**Objectif**
Ce test permet de valider les fonctions d'analyse des mails entrants et sortants. Les antivirus doivent surveiller les processus tentant d'envoyer ou de recevoir des mails et doivent bloquer le trafic lors de la détection d'un virus.

**Principe** Le scénario du test est simple : envoi et réception d'un mail comportant un fichier viral en pièce jointe. Dans un premier temps, la procédure décrite ci-dessus peut être conduite avec le fichier test EICAR [EIC06]. Le fichier test EICAR a été conçu initialement pour permettre aux éditeurs de produits antivirus d'étalonner leurs systèmes. Nous pouvons remarquer qu'un utilisateur sans connaissance particulière peut conduire ce test très aisément. Dans un second temps, la procédure décrite ci-dessus peut être itérée avec d'autres virus en pièces jointes (fichiers viraux hautement compressés, fichiers viraux archivés dans un format d'archive corrompu intentionnellement, code viral embarqué dans un document, etc.).

*En adaptant le client et le serveur permettant la mise en oeuvre d'une injection à distance de code relogeable (voir test TST_MEM), nous pouvons valider les fonctions de sécurité correspondant à l'analyse des mails entrants et sortants par utilisation des protocoles SMTP et POP/IMAP.*

**Analyse des résultats**
Conformément à la description des actions d'un antivirus donnée dans le profile de protection [CCE05] et à l'état de l'art dans ce domaine, nous attendons du produit anti-virus qu'il détecte et désinfecte ou supprime la pièce jointe.

**TST_FMT.1 (gestion des différents formats de fichier)**

**Objectif**
Certains formats de fichier ne sont pas correctement analysés par certains produits anti-virus. Ils sont classés non déterminé au lieu d'être décodés et analysés. Un attaquant peut donc employer des fichiers illicites afin de faire transiter un virus. Ce test nous permet de vérifier si le produit antivirus gère convenablement les différents formats de fichier et de compression, en particulier lorsque ces formats sont corrompus intentionnellement.
L'objectif est de tester la capacité du produit à détecter du code malveillant encapsulé dans différent type d'archives. Pour cela, la protection temps réelle étant désactivée, plusieurs fichiers archives contenant un fichier incluant la chaine EICAR sont créés. Nous effectuons alors une analyse du répertoire de test contenant les archives créées et observons la réaction du produit.

**Principe**
Nous utilisons les logiciels GNU Win32 pour archiver, compresser ou packer un virus informatique reconnu par le produit anti-virus (formats de fichiers ou extensions ARC, ARJ, BASE64, BZIP2, GZIP, LHA, LZO, MIME, MS-CABINET, MS-COMPRESS, RAR, SHAR, TAR, UPX, UUE, ZIP).
Nous procédons à l'analyse du répertoire contenant les fichiers packés, archivés et/ou compressés.

Cas du format UPX : Le format UPX (Ultimate Packer for eXecutables, [OM06]) permet de compresser un programme et de le décompresser avant son exécution. Certains produits anti-virus ne savent pas reconnaître comme tel un fichier UPX spécialement construit. Un attaquant distant peut donc créer un virus, le compresser en UPX et lui appliquer un traitement afin qu'il soit encore reconnu par le système comme étant un document UPX, mais plus par le produit anti-virus (il s'agit d'une attaque par évasion). Le logiciel antivirus peut considérer le format inconnu ou le fichier corrompu et ne pas signaler la présence d'un virus à l'utilisateur. L'utilisateur, alors en confiance, peut décider de l'exécuter. Nous avons adapté l'exécutable `upxredir` afin de pouvoir transformer un fichier UPX en un fichier UPX reconnu comme tel par le système, mais pas par le produit anti-virus (lorsque celui-ci ne gère pas ce format de manière correcte). Le test consiste donc à compresser un exécutable viral au format UPX, à s'assurer que le produit anti-virus détecte le virus dans ce format, puis à corrompre l'exécutable avant de le soumettre à nouveau à l'analyse.

Cas du format Zip : En utilisant un fichier Zip spécialement construit [Vig06e, Vig06h], un attaquant peut contourner certains anti-virus. Les fichiers Zip sont des archives compressées contenant un ou plusieurs documents et répertoires. Chacun de ces documents est précédé d'un entête. De même, un résumé, récapitulant tous les documents de l'archive, figure en fin de fichier. Un attaquant peut modifier les tailles figurant dans les entêtes et dans le résumé. Ainsi, un document supplémentaire peut être caché dans l'archive. WinZip et la fonctionnalité de décompression de Windows extraient correctement le fichier caché. Cependant, certains produits anti-virus ne le voient pas. Un attaquant peut ainsi créer une archive contenant un virus qui ne sera pas détecté avant d'atteindre le poste de l'utilisateur. Nous avons développé un outil permettant de corrompre un fichier d'archive au format Zip, en mettant à zéros les champs Uncompressed Size des structures Local File Header et Central Directory du fichier Zip.

*Les formats Zip et UPX ne sont pas les seuls formats qui soulèvent ces problèmes. Les formats suivants sont également l'objets d'attaque par évasion : RFC822 [Vig06i], MIME [Vig06b], LHA [Vig06a], URI [Vig06d], ANSI ESC [Vig06f], RAR [Vig06c]. Nous envisageons de développer les outils nécessaires pour être certains que ces formats sont bien gérés par les produits antivirus.*

**Analysis des résultats**
Concernant le cas des formats de fichiers corrompus intentionnellement, nous pouvons observer que certains produits anti-virus n'inspectent pas convenablement une archive au format Zip ou UPX illicite, et sont donc vulnérable à cette attaque par évasion. Tous les formats d'archive ou d'encodage doivent être supportés, quelle que soient les entorses faites aux spécifications de ces formats.

**TST_FMT.2 (filtrage des virus de document)**

**Objectif**
L'objectif est de tester la capacité du produit à détecter l'exécution de scripts (écrit en VB Script ou Java Script) embarqués (ou non) dans des documents Microsoft Office ou HTML.

**Principe**
Windows (et plus généralement de nombreuses suites logicielle) propose une variété de langage de script permettant de faire rapidement et simplement des tâches courantes (accès au système de fichier, au registre, ...). Un code mal intentionné peut utiliser ces commandes afin de se propager. Les utilisateurs réactivent souvent l'exécution des macros, des scripts ou des applets dans une application Office ou dans un navigateur, de manière à assouplir la configuration par défaut du logiciel. Nous avons adaptés les tests de conformités suggérés par eSafe pour valider son moteur de filtrage des scripts [eSa06].
À titre d'exemple, l'un des scripts de test utilisés créé un répertoire sur le bureau, y copie le fichier `win.ini` puis créé un fichier texte contenant la date et l'heure du test. Ce même script est ensuite intégré dans un document MS Word et configuré pour s'exécuter automatiquement lors de l'ouverture du document. La protection temps réelle

activée, nous exécutons dans un premier temps la version seul du script (fichier vbs) puis ouvrons le document MS Word contenant le script et observons la réaction du produit (pour ce test, le niveau de sécurité de MS Word à été placé sur faible afin d'autoriser sans conditions l'exécution des macros).

**Analyse des résultats**
Un langage de script autorisant l'accès total au système, on s'attend à ce que le produit détecte l'exécution d'un tel code. Le logiciel anti-virus doit être en mesure de notifier la présence de code viral embarqué dans un document.

**TST_MEM.1 (Exploitation d'un débordement de tampon)**

**Objectif**
L'objectif est de tester la capacité du produit à détecter un débordement de tampon (buffer overflow) qui est un des vecteurs d'attaque commun. Nous utilisons un programme sensible à ce type d'attaque. Pour cela, la protection temps réelle activée, nous appelons notre programme de test en lui passant une chaîne de caractère spécialement conçue et observons la réaction du produit.

**Principe**
Nous utilisons un programme serveur autorisant l'exploitation d'un débordement de tampon [FO04]. Ce type de vulnérabilité est utilisée comme vecteur de propagation des vers. Nous validons donc l'efficacité de la fonction de blocage des vers du produit antivirus. Nous injectons du code viral relogeable (shellcode viral) depuis un programme client.
Notre programme de test possède sur la pile un tampon de taille fixe dans lequel il copie les arguments lus depuis la ligne de commandes. Notre chaîne spéciale passée en argument fait déborder ce tampon (écrit au-delà de la zone allouée pour ce dernier) et permet alors l'exécution d'instructions créant le fichier `c:\yes.txt`.

*Les techniques utilisées pour l'injection de code à distance peuvent être améliorées afin de rendre compte des nouvelles générations d'exploits. L'utilisation de LSD [LSD02] ou du framework Metasploit pour les tests de pénétration peut fournir une approche complémentaire.*

**Analyse des résultats**
Le produit anti-virus doit détecter la présence d'une infection informatique en mémoire. Certains produits antivirus ont une fonction de surveillance des débordement de tampon (pile ou tas). Cette fonctionnalité peut être validée.

**TST_MEM.2 (Analyse de la mémoire vive)**

**Objectif**
L'objectif de ce test est de valider le fonctionnement du produit anti-virus, au niveau de la mémoire vive. Ce test permet de vérifier qu'un virus présent en mémoire sera bien détecté par le produit antivirus.

**Principe**
L'outil VxStripper propose l'utilisation de méthodes forensiques pour écrire un code viral directement en mémoire vive. Lorsque le produit antivirus propose une fonction d'analyse de la mémoire vive, ce test permet de valider son efficacité. Nous validons l'efficacité de la fonction de détection des virus résidants en mémoire.

**Analyse des résultats**
Le produit antivirus doit détecter la présence d'une infection informatique en mémoire.

**TST_EFF.1 (Analyse de la base virale)**

**Objectif**
L'objectif de ce test est de valider la complétude de la base des signatures virales du produit anti-virus.

**Principe**
Nous disposons d'une base de virus (comptant environ 30000 virus identifiés, au moment des analyses). La base

de virus comporte un large spectre du simple macro virus au ver polymorphe incluant des virus UNIX, DOS et Win16/32 (une majorité). Nous paramétrons le scanner de façon à cibler l'analyse sur le répertoire contenant la base virale. Afin de faciliter la maintenance de la base virale, nous avons développé un moniteur temps-réel permettant de bloquer l'exécution du code viral (en cas d'accès accidentel à un programme viral) et d'alerter l'administrateur de la base viral.

Ce moniteur temps-réel comporte trois composants :

- un programme client, qui récupère les E/S effectuées depuis l'espace utilisateur vers les systèmes de fichiers et procède à leur analyse en appliquant :
  - l'algorithme de recherche de motif Aho/Corasick [AC75, K+07] au regard d'une base de signatures virales ;
  - l'algorithme de recherche de motif FDA/NFA au regard d'une liste d'expressions rationnelles PCRE [Haz03] ;
- un pilote du noyau, permettant d'intercepter les IRP et les E/S rapides, communiqués par le gestionnaire des E/S vers les pilotes des systèmes de fichiers (FAT, NTFS, RDR[1]).

En cas de détection d'un virus, le pilote du noyau bloque l'accès au système de fichier et le programme client alerte l'utilisateur. Le programme client procède au chargement de la base des signatures, à l'initialisation des structures de recherche de motif, à l'ouverture d'un canal de communication vers le pilote et à la création des thread. Chaque thread connecté au pilote récupère les données du pilote et procède à l'analyse (par recherche de motif). En cas de détection, le thread déclenche une alerte. Le programme client envoie alors un message au pilote, imposant un blocage de la requête d'E/S par le pilote.

Le pilote s'enregistre auprès de l'exécutif NT, puis crée un port serveur de communication sur lequel il peut recevoir des requêtes de connexion depuis le mode utilisateur. Il associe au port un descripteur de sécurité limitant l'accès à l'administrateur. Il peut alors intercepter tous les paquets (IRP, Fast I/O) transitant du gestionnaire des E/S vers les pilotes des systèmes de fichiers (FAT, NTFS, RDR). Lorsqu'une requête d'E/S vers le système de fichier est effectuée en espace utilisateur, le pilote récupère les informations sur le fichier, lit le début de ce fichier, passe le contenu en espace utilisateur et envoie une requête d'analyse. Si le fichier est suspect, la requête d'E/S est bloquée. Si le fichier n'est pas suspect et que la requête d'E/S correspond à une ouverture en écriture du fichier, le fichier sera analysé à nouveau. Le contexte est récupéré. Si une nouvelle analyse est exigée, le pilote relit le début du fichier, passe le contenu en espace utilisateur et envoie une nouvelle requête d'analyse. Si le fichier est suspect, la requête d'écriture est bloquée.

Ce moniteur temps-réel implémente un verrouillage des fichiers. Un fichier ouvert par un processus ne pourra donc pas, avant sa fermeture, être lu et exécuté par un autre processus [MDWZ04]. Ce programme permet également de procéder à une analyse à la demande de tout ou partie de la base virale, au regard de la base des signatures.

L'outil VxStripper est utilisé pour maintenir une base virale constituée exclusivement de virus non packés (par utilisation d'un des packers du marché). VxStripper nous permet également de statuer rapidement sur la non corruption d'un exécutable (celui-ci s'exécute correctement et ne s'arrête pas sur une erreur du système) et de vérifier que la description d'un virus (identifié sur une des bases d'informations maintenues par les éditeurs de produits antivirus) correspond aux interactions observée du virus avec son environnement d'exécution (notamment les appels aux fonctions d'API). Cet outil nous est donc précieux pour la gestion de notre collection de virus, en nous permettant de vérifier automatiquement qu'un exécutable de la collection n'est pas endommagé, s'exécute effectivement et correspond à sa description publique.

**Analyse des résultats**

L'analyse de la base virale est conduite avec différents masques de configuration (moteur d'analyse heuristique activé/désactivé, analyse des fichiers d'archives, configuration par défaut, etc.). Les taux de détection obtenus sont des informations importantes, permettant de valider la complétude de la base de signatures virales du produit et de comparer plusieurs produits antivirus sur la base de ces résultats.

[1] Les système de fichiers RDR sont des redirecteurs NT, qui permettent aux services d'une station de travail d'accéder à distance aux fichiers via le gestionnaire d'E/S. Dans une représentation en couches (mode utilisateur/mode noyau) de l'architecture réseau de WinNT, les redirecteurs sont au dessus de l'interface des pilotes de transport TDI.

### TST_EFF.2 (Exécutables packés)

**Objectif**
L'objectif est de tester la capacité du produit à détecter du code malveillant encapsulé par utilisation d'un packer d'exécutable (un packer permet de protéger un exécutable en le compressant et/ou chiffrant. il ajoute un loader à l'exécutable qui le décompresse et/ou le déchiffre avant de lui rendre le contrôle).

**Principe**
Un virus connu du produit antivirus analysé est encapsulé en utilisant différents packers du marché. Nous effectuons une analyse du répertoire de test contenant les exécutables créés et une analyse en temps réel.

| | Détection (à la demande) | | Détection (temps réel) | |
|---|---|---|---|---|
| Packer | oui | non | oui | non |
| Armadillo | | | | |
| Asprotect | | | | |
| Petite | | | | |
| Shrinker | | | | |
| UPX | | | | |
| UPX corrompu | | | | |
| yC | | | | |

**Analyse des résultats**
On s'attend à ce que le produit antivirus détecte le code malveillant encapsulé, quel que soit le packer utilisé.

### TST_EFF.3 (Exécutables polymorphes, EPO)

**Objectif**
L'objectif est de tester la capacité du produit à détecter des variantes d'un virus, par applications de transformations sur le code, permettant de préserver toutes les fonctions du programme tout en modifiant sa forme. Ce type de transformation est implémentée par les virus pour rendre plus difficile la détection par recherche de signature et/ou rendre plus difficile l'analyse du code (intelligibilité). Il s'agit donc de vérifier qu'il n'est pas trop évident, pour un attaquant, de forger un nouveau virus à partir d'un virus existant, par l'utilisation d'outils qu'il est facile de se procurer (kits de génération de virus, moteur de mutation).
L'objectif de ce test est de valider le bon fonctionnement du moteur de détection d'un produit anti-virus donné. Ce test nous permet également de mieux comprendre les actions effectuées par le moteur de détection.

**Principe**
Nous appliquons à un virus reconnu par le produit antivirus les deux transformations élémentaires suivantes :
- introduction de junk code (c'est à dire du code qui ne modifie pas le fonctionnement du programme d'origine. Il doit donc préserver le plus possible l'état du processeur et de la pile d'exécution) dit mort, dans la mesure où il ne sera jamais exécuté ; ou vivant, c'est-à-dire effectivement exécuté par le processeur ;
- permutation des instructions au sein d'un basic block ou remplacement d'une instruction par un bloc d'instruction équivalent (substitution). Un basic block est un ensemble d'instructions assembleur qui est toujours exécuté séquentiellement de la première instruction à la dernière. Il comporte un unique point d'entrée (destination d'une instruction de saut, point d'entrée d'une procédure, instruction consécutive à une instruction déclenchant une exception) et un unique point de sortie (instruction de saut, instruction déclenchant une exception, instruction de retour à la procédure appelante).

Le code exécutable du virus est désassemblé sommairement, décomposé en basic blocs, instrumenté (insertion de junk code, permutation des instructions, insertion de basics blocks supplémentaires) puis régénéré, sur la base de l'information de relocation (cette information, parfois disponible dans un exécutable, toujours disponible dans une bibliothèque dynamique, permet de déterminer la liste des adresses dans le code exécutable qui doivent être modifiées lorsque le loader reloge le code à une autre adresse en mémoire). Nous utilisons cette information pour reconstruire l'exécutable, et lui permettre de supporter la greffe imposée par l'application (chirugicale ;0) des transformations d'obfuscation.

La section des relocations (fixups) d'un exécutable PE (Portable Executable), si elle est présente (les exécutable ne comportant pas de section des relocations ne peuvent pas être chargés en mémoire à une adresse différente de celle fixée par l'éditeur de lien), contient une table de fixup blocks. Chacune de ces entrée contient :

- la RVA d'une page de 4Ko de l'exécutable mappé en mémoire ;
- le nombre de fixups ;
- une table de fixups, codés sur 2 octets (les 12 bits de poids faible codent le décalage dans le bloc mémoire - 512 octets - qui doit être relogé et les 4 bits de poids fort codent le type de fixup).

Nous considérons une population $S_0$ de programmes sains du système. Nous générons une population $S_1 = T(S_0)$ de programmes sains (exécutables Win32) possédant des propriétés classiquement jugées suspectes par les produits antivirus, au niveau :

- de la structure de l'exécutable (entête, sections de données et de code) ;
- de la section code du fichier exécutable.

La transformation $T$ fut implémentée d'abord en utilisant une version adaptée du packer Y0da's packer [Y0d06]. Cette transformation est actuellement effectuée par l'obfuscateur décrit plus bas, en raison de sa meilleure modularité. Les fichiers exécutables générés par la transformation $T$ possèdent des caractéristiques incompatibles avec le code habituellement généré par un compilateur de langage haut niveau :

- des sections sont incorrectes (par exemple la dernière section est exécutable ou la première section est accessible en écriture, des sections sont incorrectement alignées ou ont des noms étranges) ;
- l'entête est incorrecte (valeurs incorrectes dans l'entête, point d'entrée inhabituel).

Nous considérons une population $V_0$ constituée de virus Win32. Nous construisons une population $V_1 = O(V_0)$ par application d'un obfuscateur $O$. Nous avons procédé, comme pour la validation du moteur de détection virale SAFE (Static Analyser For Executable, [CJ03]), à une adaptation du moteur Mistfall [Z0m00a, Z0m00b]. Notons que cette idée a également été utilisée pour valider l'efficacité du moteur de détection virale SAVE [SXCM04], même si les auteurs n'ont pas automatisée l'application des transformations d'obfuscation (les transformations ont été appliquées manuellement, à l'aide d'un simple éditeur hexadécimal). Le moteur viral Mistfall implémente un désassembleur (LDE) et un moteur de permutation (RPME). Nous avons instrumenté ce moteur de telle sorte qu'il réalise des transformations d'obfuscation plus complexes [Jos05]. Nous l'avons couplé avec un générateur de junk code et une base de prédicats opaques.

*Nous n'avons pas exploré les possibilités offertes par les générateurs de code viral (Virus Generator Kits [Szö05]). Leur utilisation pourrait compléter notre analyse (génération ex nihilo d'une population virale $V_2$. Nous n'avons pas non plus exploré les possibilités offertes par le fait de disposer du code source de certains virus. Nous pourrions à partir de ces programmes construire une population $V_3$ par application d'un obfuscateur $O$ comme Cloakware, pour les langages C/C++ et [Clo07] ou SandMark pour le langage Java [CMH03]. Par ailleurs, la chaîne de compilation LLVM paraît adaptée à l'application de transformations d'obfuscation sur du code source C.*

**Analyse des résultats**

L'analyse corrélée des populations $S_1$, $V_0$ et $V_1$ permet de tirer plusieurs conclusions sur le produit anti-virus :

- sa capacité à analyser les virus polymorphes ;
- sa capacité à analyser les virus EPO (Entry-Point Obscuring, [Gri06]).

Il est également possible de voir quelles sont les heuristiques utilisées, ainsi que les limites imposées au niveau du paramétrage par le constructeur du produit anti-virus (profondeur de l'analyse, etc.).

**TST_RK.1.1 (test du *Global Low Level Keyboard hook*)**

**Objectif**

L'objectif est de voir la réaction du produit antivirus par rapport à l'installation d'un crochet interceptant les événements clavier via une méthode documentée. La protection temps réelle activée, nous installons un crochet global interceptant les événements clavier et observons la réaction du produit.

**Principe**

L'API Windows propose des fonctionnalités permettant d'intercepter certains événements système tel que les mou-

vements de la souris ou les touches du clavier via la fonction `SetWindowsHookEx`. La protection temps réel de l'antivirus activée, le programme de test installe, suivant cette méthode, une fonction qui récupère les touches tapées par l'utilisateur et les stocke dans un fichier.

**Analyse des résultats**
Nous attendons du produit antivirus qu'il détecte l'installation et la présence de ces hooks.

**TST_RK.1.2 (IAT/EAT et *inline patching*)**

**Objectif**
L'objectif est de tester la capacité du produit antivirus à détecter les méthodes non conventionnelles d'instrumentation des fonctions d'API en espace utilisateur.

**Principe**
Une première méthode consiste à modifier en mémoire pour un processus cible la table des imports des fonctions (IAT). Cette table contient les adresses de toutes les fonctions des DLL chargées dans l'espace du processus. Le code du processus faisant appel à une de ces fonctions appelle l'entrée correspondante dans l'IAT qui effectue un saut vers l'adresse où réside la fonction. Nous remplaçons l'adresse de la fonction à intercepter dans l'IAT par l'adresse d'une fonction de remplacement qui effectuera (ou non) l'appel à la fonction originale.

Une autre méthode assez similaire consiste en la modification des adresses situées dans la table des exports (EAT) du module.

Le *runtime patching* (encore appelé *inline hooking*) consiste non plus à modifier l'adresse de la fonction dans la table des imports d'un processus cible, mais à insérer un saut vers une fonction de remplacement au tout début du code de la fonction cible.

**Analyse des résultats**
Nous attendons du produit antivirus qu'il détecte l'installation et la présence de ces hooks.

**TST_RK.2.1 (passage en mode noyau)**

**Objectif**
L'objectif de ce test est d'éprouver la robustesse du moteur de détection de rootkits vis-à-vis de plusieurs techniques de passage en mode superviseur.

**Principe**
Le chargement d'un module dont le code s'exécute en mode superviseur permet d'obtenir un accès à tout l'espace de mémoire virtuelle. Contrairement au mode utilisateur, un code s'exécutant dans l'espace noyau peut accéder à tout cet espace sans contrainte. Il n'est cependant pas forcément nécessaire d'utiliser un driver pour exécuter du code en mode superviseur.

La méthode documentée pour exécuter du code en mode superviseur est l'utilisation d'un driver. Le chargement normal d'un tel module se fait par la création d'un service via le gestionnaire de services. Ce dernier inscrit le driver dans le registre et appelle la fonction `ZwLoadDriver` (en ring 0) pour charger le driver. La routine d'initialisation est alors appelée. Le code du driver s'exécute en mode superviseur dans l'espace mémoire du noyau et a accès à toute la mémoire.

La fonction non documentée `ZwSetSystemInformation` et non exportée (de `ntdll.dll`) permet, en lui fournissant le paramètre `SystemLoadAndCallImage = 38`, de charger un driver sans avoir à créer de clés dans la base de registre. Le driver est alors chargé normalement, sa routine d'initialisation est appelée et son code s'exécute en mode superviseur dans l'espace mémoire du noyau et a accès à toute la mémoire.

Ces deux techniques impliquent l'utilisation de driver. Il est cependant possible d'exécuter du code en mode super-

viseur, sans utiliser de driver. Les principales méthodes sont les suivantes :

- installation d'une *Call Gate* dans la table des descripteurs globaux (GDT). (descripteur permettant le changement de niveau de privilège, contenant entre autres l'adresse d'une fonction à exécuter en mode superviseur et le niveau de privilège requis pour effectuer l'appel. Un code utilisateur peut ajouter un tel descripteur et l'utiliser pour exécuter n'importe quel code. Une fois la porte installée, il suffit de l'appeler et la fonction qui y est associée est exécutée en mode superviseur) ;
- installation d'un gestionnaire d'interruption dans la table des interruptions (cette table regroupe un ensemble de descripteurs associant un numéro d'interruption à un gestionnaire qui sera exécuté en mode superviseur. Un code utilisateur peut ajouter un descripteur à cette table et l'utiliser pour exécuter du code en mode superviseur. Une fois le gestionnaire installée, il suffit de déclencher l'interruption associée et le gestionnaire correspondant est exécuté en mode superviseur) ;
- en remplaçant la valeur du registre `IA32_SYSENTER_EIP` par une adresse que nous contrôlons afin d'exécuter du code en mode superviseur (normalement, c'est l'adresse de la routine système `KiSystemService` qui se trouve dans `IA32_SYSENTER_EIP`. Cette routine s'occupe de rechercher le service natif correspondant au numéro de service placé dans le registre `EAX` dans la table des services système SSDT puis appelle le service) ;
- en remplaçant un des services système par le notre (une table des services regroupe les adresses de toutes les fonctions exportées du noyau. Elle est utilisée notamment par la fonction noyau `KiSystemService`. Certains de ces services ne sont jamais ou très rarement appelés), en plaçant son numéro dans `EAX` puis en exécutant l'instruction `SYSENTER` - ou `SYSCALL` pour les processeurs AMD (ou simplement en appelant la fonction d'API correspondante qui s'occupera de l'appel).

Remarquons tout d'abord que les différentes tables et registres mentionnés ci-dessus ne sont normalement pas modifiables depuis le mode utilisateur. En effet, afin de respecter le principe de la protection en anneau, il ne faut pas que du code utilisateur puisse choisir d'exécuter du code arbitraire en mode superviseur. Cependant, il est possible pour du code utilisateur ayant les privilèges nécessaires de lire/écrire dans l'espace de mémoire du noyau, soit en utilisant le périphérique `\Device\PhysicalMemory` qui représente la mémoire physique de la machine, soit en utilisant la fonction système `ZwSystemDebugControl`.

**Analyse des résultats**
Le produit antivirus doit détecter l'exécution de code viral en mode superviseur.

### TST_RK.2.2 (pilote filtre/keylogger)

**Objectif**
L'objectif est de tester la capacité du produit à détecter un *keyloggeur* utilisant la possibilité de l'empilement de drivers. Pour cela, la protection temps réelle activée, nous chargeons le driver de notre *keyloggeur* via l'outil `InstDriver` et observons la réaction du produit.

**Principe**
Nous utilisons `klog` qui attache son driver à celui du clavier (`kbdclass`) et intercepte les données retournées par ce driver. Pour cela, `klog` utilise le principe documenté des drivers en couche (*filter drivers*) : un driver peut être attaché à un autre et filtrer les données (*Interrupt Request Packet* - IRP) circulant depuis et vers le matériel. `klog` créé le fichier `c:\klog.txt` et y stocke les touches capturées.

**Analyse des résultats**
Nous attendons du produit antivirus qu'il détecte l'installation et la présence de ce logiciel.

### TST_RK.3.1 (test DKOM)

**Objectif**
L'objectif est de tester la capacité du produit à détecter la manipulation directe des objets du noyau NT. La protection temps réelle activée, nous utilisons un outil pour cacher le processus `explorer.exe` (le shell) et le driver et observons la réaction du produit.

**Principe**
Le noyau NT stocke des objets décrivant les processus et threads. En modifiant ces objets, il est possible de dissimuler des processus, d'élever leur niveau de privilège (ajout de privilèges et de groupes au token d'un processus ou d'un thread) et d'effectuer d'autres modifications. La corruption d'autres objets de l'exécutifs NT permet également de dissimuler les pilotes. Nous disposons d'un outil, adapté de `fu` [Hog06], permettant d'effectuer les manipulations suivantes sur le noyau NT :

- corruption de la liste doublement chaînée de structures EPROCESS afin de dissimuler certains processus aux yeux des outils de l'administrateur système (TaskMgr.exe, par exemple)
- corruption de la liste doublement chaînée de structures MODULE_ENTRY afin de dissimuler certains pilotes aux yeux des outils de l'administrateur système (`Drivers.exe`, par exemple)
- corruption du token au sein d'une structure EPROCESS afin par exemple d'élever le niveau de privilège du processus correspondant.

Il est possible d'effectuer cette modification de manière furtive aux yeux des outils de l'administrateur système (le Windows Event Viewer, en particulier).

**Analyse des résultats**
Nous attendons du produit antivirus qu'il détecte le chargement du driver, la modification direct du noyau et qu'il notifie l'utilisateur.

**TST_RK.3.2 (autres hooks en mode noyau)**

**Objectif**
L'objectif est de tester la capacité du produit à détecter plusieurs variétés de rootkit afin de déterminer s'il intègre des fonctionnalités anti rootkits. Nous disposons de programmes permettant la corruption des objets du noyau (IDT, SSDT, Sysenter, DRIVER_OBJECT Major Function IRP Hook). La protection temps réelle activé, nous utilisons nos différents programmes de test afin de modifier le noyau Windows.

**Principe**
Un driver gérant des requêtes d'entrée/sortie possède une table associant le type de requête (lecture, écriture,...) à une fonction de traitement dans le driver (*Major Function Interrupt Request Packet table*). Dans ce test, nous modifions la table du driver réseau afin de placer notre propre gestionnaire et récupérer ainsi les paquets reçus sur l'interface réseau. Le code noyau de MS Windows fonctionne en mode superviseur et le code des applications fonctionne en mode utilisateur. Le noyau expose un grand nombre de services système regroupés dans la *System Services Descriptor Table*. Le code utilisateur utilise alors une l'interruption `0x2E` ou l'instruction spécifique `SYSENTER` afin de passer en mode superviseur. Dans les deux cas, l'adresse du gestionnaire associé est placée respectivement dans la table des interruptions (*Interrupt Descriptor Table*) ou dans le registre spécial `SYSENTER_EIP_MSR`. Un de nos outils permet de placer un gestionnaire personnalisé permettant ainsi à du code utilisateur d'exécuter n'importe quel code en mode superviseur.

**Analyse des résultats**
Nous attendons du produit antivirus qu'il détecte l'installation et la présence de logiciel utilisant ces techniques de rootkits et qu'il notifie l'utilisateur.

# Annexe B

# Critères de sécurité boîte noire et génération aléatoire de S-Boxes vérifiant ces critères

Je présente dans cette annexe les principaux critères de sécurité sur lesquels fonder la construction de S-Boxes (section B.1) puis le résultat de mes expérimentations concernant la génération aléatoire de S-Boxes vérifiant ces critères (section B.2).

L'essentiel de la revue des critères de sécurité est fondée sur les travaux de thèses [Fon98, Fil01, Vid05, Bra06] et les papiers [KSS97, Ada97, PJ99, Can01, Can02, CP02]

## B.1 Critères de sécurité boîte noire

L'analyse en boîte noire du DES a conduit à la définition de critères de sécurité généraux (comme le critère de stricte avalanche, critère de propagation) des boîtes de confusion d'un système de chiffrement itératif vis-à-vis des cryptanalyses linéaires et différentielles. Les fonctions vectorielles $f : \mathbb{F}_2^m \to \mathbb{F}_2^n$ qui sont les plus robuste contre ces attaques sont les fonctions courbes ou parfaitement non linéaires, lorsque $m \neq n$ ($m$ pair et $m \geq 2n$), et les fonctions presque courbes dans le cas contraire ($m = n$ est impair). Ces fonctions sont caractérisées par une corrélation minimale avec les fonctions affines et les fonctions décalées de $f$.
Ces critères sont fondamentaux également pour la conception et l'évaluation dans le contexte boîte blanche pour plusieurs raisons :

- d'une part une implémentation boîte blanche doit également être résistante à la cryptanalyse en boîte noire ;
- d'autre part, la cryptanalyse en boîte blanche utilise des méthodes boîtes noires, avec une granularité plus fine (ici, chaque lookup table peut être vue comme une boîte noire, dont on cherche à extraire de l'information ou la clé) ;
- enfin, la conception d'un algorithme de chiffrement spécialement pensé pour être résistant en boîte blanche peut s'appuyer sur ces critères pour prouver sa sécurité.

Il est tentant d'essayer de caractériser des notions de diffusion et de confusion, introduites par Shanon [Sha49], par des critères sur les fonctions booléennes vectorielles. L'analyse en boîte noire du DES a conduit à la définition de critères de sécurité généraux sur les boîtes de diffusion et de confusion d'un système de chiffrement itératif par bloc vis-à-vis des cryptanalyses linéaire et différentielle, et de manière plus générale aux attaques par oracle aléatoire sur le dernier tour.
Les fonctions vectorielles $f$ les plus robustes vis-à-vis de ces attaques sont les fonctions courbes ou parfaitement non linéaires, dans le cas où $f$ a autant de variables que de composantes, et les fonctions presques courbes (qui sont aussi presque parfaitement non linéaires, la réciproque étant fausse) dans le cas contraire. Ces fonctions sont caractérisées par une corrélation minimale avec les fonctions affines et les décalées de $f$
Cependant, nous pouvons remarquer qu'il n'y a pas de consensus réel sur la caractérisation des notions de diffusion et de confusion. La notion de confusion est le plus souvent caractérisée par une non linéarité optimale. Cependant,

une autre approche consiste à interpréter la notion de confusion comme une décorrélation parfaite des composantes de $f$ prises deux à deux.
La notion de diffusion est parfois confondue avec celle de critère de propagation d'ordre élevé, qui est maximal pour les fonctions courbes, ou avec la notion de multipermutation. Une autre approche consiste à caractériser la propriété de diffusion d'une fonction booléenne vectorielle $f$ par une corrélation uniformément petite avec les fonctions affines et les décalées de $f$.

Concernant l'efficacité des primitives de chiffrement itératif par blocs dans le contexte WBAC, nous pouvons faire les observations suivantes : tout d'abord il est assez naturel d'avoir essayé de porter des algorithmes connus pour leur robustesse en boîte noire, dans la mesure où un algorithme de chiffrement résistant dans le contexte WBAC doit également être résistant en boîte noire. Nous pouvons ensuite remarquer que les premières spécifications sont assez recentes : Spécification WB-DES (2002, [CEJvO02a]), spécifications WB-AES (2002, [CEJvO02b]). Les attaques sont principalement des cryptanalyses différentielles : Cryptanalyse WB-DES par injection de fautes (2002, [JBF03]), Cryptanalyse WB-AES (2004, [BGEC04]), Amélioration WB-DES (2005, [LN05b]), Cryptanalyse WB-DES (2005, [PW05]), Cryptanalyses WB-DES (2007, [WMGP07] et [GMQ07]). L'efficacité de ces attaques est bien plus importante que les meilleures attaques menées dans un contexte boîte noire sur le DES ou l'AES.

Ce résultat formalise l'idée suivante : l'obfuscation de programme ne peut pas fournir une sécurité forte en pratique car les ressources et techniques à disposition de l'attaquant dans un contexte boîte blanche sont nombreuses et puissantes. Dans ces conditions, dès lors qu'un attaquant dispose d'information sur l'algorithme $\mathcal{O}$, il est en mesure de développer un outil adapté et spécialement optimisé pour analyser l'exécution des programmes obfusqués par $\mathcal{O}$.

Je pense que rendre aléatoires les boites de confusion et de diffusion serait une première étape vers la réalisation d'un algorithme de chiffrement par bloc itératif résistant à l'analyse boîte blanche, dans la mesure où leur connaissance par un attaquant permet de donner trop d'efficacité à la reconstruction des bijections délimitant les phases linéaires de permutation et au recouvrement de la clé embarquées dans les boîtes de substitution.

Ce pari fait l'hypothèse qu'il est possible de concevoir un algorithme d'obfuscation de clé dans un algorithme dont la connaissance ne fournit pas suffisamment d'information pour permettre la mise en oeuvre d'attaque par cryptanalyse linéaire ou différentielle, visant à reconstruire les primitives cryptographiques de confusion et de diffusion, les bijections aléatoires ou à retrouver la valeur des clés de tour. Nous sommes donc à même de nous demander si une implémentation boîte-blanche forte du chiffrement par bloc est réalisable. Malgré l'efficacité des cryptanalyses et les résultats généraux d'impossibilité concernant l'obfuscation, il n'est pas prouvé qu'il n'est pas possible de réaliser une implémentation boîte-blanche forte d'un algorithme de chiffrement par bloc. Nous savons seulement que la tâche n'est pas aisée. Une première piste de recherche pourrait être de développer un algorithme de chiffrement par bloc spécifique, conçu spécialement pour une implémentation boîte blanche, plutôt que d'utiliser des algorithmes de chiffrement itératif par blocs connus pour être résistants en boîte noire et dont l'implémentation est simplement adaptée au contexte boîte blanche.
La grande force de ces algorithmes repose en effet sur l'utilisation répétée d'une fonction de tour. Dans le contexte WBAC, les attaques peuvent être menées sur un nombre réduit de tour, voyant leur efficacité décuplée.
Il est peut être possible de concevoir un algorithme d'obfuscation de clé dans un algorithme dont la connaissance ne fournit pas suffisamment d'information pour permettre la mise en oeuvre d'attaque par cryptanalyse linéaire ou différentielle, visant à reconstruire les primitives cryptographiques de confusion et de diffusion, les bijections aléatoires ou à retrouver la valeur des clés de tour.
Les techniques actuelles d'obfuscation des algorithmes de chiffrement par bloc consistent, en quelque sorte à reproduire un contexte boîte noire à un niveau de maillage plus fin : celui du tour. En effet, chaque tour $r$ est grosso modo implémenté par une lookup table « noircie » grâce à l'adjonction de bijections aléatoires en entrée et sortie. Cette boîte transmet une donnée (le texte en cours de chiffrement) à la lookup table du tour suivant de manière « codée », là encore grâce à l'insertion de bijections aléatoires. La sécurité de cette implémentation boîte blanche semble alors réduite à la robustesse des lookup tables « noircies », c'est-à-dire sur l'incapacité d'extraire de l'information de ces dernières, ainsi que sur celle du codage des données transitant entre les lookup tables. L'utilisation d'un schéma de type « confusion / diffusion » apparaît clairement (surtout dans l'implémentation boîte blanche de l'AES) pour coder les données transitant entre ces lookup tables noircies. Intuitivement, il n'est donc pas étonnant que ces implémentations soit « cassées » puisque, si l'on considère les lookup tables comme des boîtes noires (on

suppose qu'il n'y a aucune fuite d'information), ces boîtes communiquent finalement entre elles à l'aide d'un schéma de chiffrement de type « confusion / diffusion » ne comportant qu'un unique tour, là où dans la cryptographie boîte noire classique, dix tours pour l'AES-128 et seize pour le DES sont préconisés.

### Caractérisation des propriétés de diffusion et confusion

Les boîtes de diffusion et de confusion sont des constituants de la pluspart des primitives cryptographiques conventionnelles (systèmes de chiffrement par blocs, fonctions de hachage, etc.).

- Les boîtes de confusion servent à faire disparaître toute structure particulière, algébrique ou statistique, des données. Il s'agit (en particulier) de dissimuler toute structure linéaire.
- Les boîtes de diffusion permettent de fusionner plusieurs données tout en propageant toute modification, même minime, des entrées dans leurs sorties. On utilise de telles boîtes dans les systèmes de chiffrement, notamment afin de propager l'information contenue dans le texte clair et dans la clé.

L'étude des fonctions booléennes permet de dégager un certain nombre de critères de sécurité sur ces composants.

- De multiples critères de non-linéarité (strict avalanche criterium, critère de propagation) ont été développés notamment à partir de l'analyse des boîtes noires du DES. Une des conditions les plus forte est que les composantes des boîtes de confusion opérant sur des données binaires soient des fonctions $f : \mathbb{F}_2^m \to \mathbb{F}_2$ telles que, pour tout vecteur $a \in \mathbb{F}_2^m$, $f(x) = f(x \oplus a)$ pour exactement la moitié des valeurs de $x \in \mathbb{F}_2^m$. Une telle fonction est dite parfaitement non linéaire puisqu'une fonction linéaire vérifie au contraire $f(x) = f(x \oplus a)$ pour toutes les valeurs de $x$ ou pour aucune d'entre elles. Les boîtes correspondantes sont celles qui résistent le mieux à la cryptanalyse différentielle. Ces fonctions correspondent exactement aux fonctions courbes, et se caractérisent notamment par une propriété sur la transformée de Fourier de la fonction signe des composantes. Elles n'existent que dans le cas où le nombre de variable $m$ est pair.
- Cette interprétation de la notion de confusion a pour défaut de ne pas tenir compte des relations entre les composantes de la fonction booléenne vectorielle. Une autre approche consiste à interpréter la notion de confusion comme signifiant que la corrélation entre chaque couple de composantes de la fonction booléenne vectorielle est minimale.
- L'idée intuitive selon laquelle la propriété de générateur aléatoire localement localement parfait est proche de celle de diffusion a été formalisée à travers la notion de multipermutation. Une boîte contenant une telle fonction assure une diffusion parfaite, dans la mesure où la modification de $t$ valeurs d'entrées induit une modification d'au moins $n - t + 1$ valeurs de sortie. Son utilisation au sein d'une primitive cryptographique permet d'éviter bon nombre d'attaques, notamment celles qui consistent à trouver un couple de valeurs d'entrées proches dont les images par la boîte sont également très proches. Il est en outre indispensable que les multipermutations utilisées dans les boîtes de confusion d'une primitive cryptographique aient un ordre de non-corrélation binaire élevé. Cette interprétation de la notion de diffusion se focalise donc sur des propriétés de propagation de la fonction vectorielle booléenne.
- Ici encore, cette interprétation de la notion de diffusion ne tient peut être pas suffisamment compte des relations entre les composantes de la fonction booléenne vectorielle. Une autre approche consiste à caractériser la propriété de diffusion d'une fonction booléenne vectorielle par un spectre de Walsh-Hadammard (corrélation de $f$ avec les fonctions linéaires) et un spectre d'autocorrélation (corrélation de $f$ avec ses décalées) aux valeurs uniformément petites.

Il est à noter que quelque soit l'interprétation faite des notions de confusion et de diffusion d'une fonction booléenne vectorielle, ces deux notions induisent des conditions nécessaires sur la structure des fonctions de tour des chiffrements itératifs par blocs : les composantes des fonctions booléennes vectorielles de tour doivent être hautement non linéaire et le plus possible décorrélées. Le premier critère démontre la résistance vis-à-vis des cryptanalyses différentielle et linéaire. En effet, soit $F : \mathbb{F}_2^m \to \mathbb{F}_2^n$. Notons $\lambda_F$ la résistance de $F$ à la cryptanalyse linéaire et $\delta_F$ la résistance de $F$ à la cryptanalyse différentielle. On a le résultat suivant :

- Si $m = n$, $\lambda_F$ est maximal si et seulement si $F$ est presque courbe. Ceci ne peut se produire que lorsque $m$ est impair.
  $\delta_F$ est maximal si et seulement si $F$ est presque parfaitement non linéaire. On conjecture que ceci ne peut se produire que lorsque $m$ est impair.
- Si $m \neq n$, $\lambda_F$ est maximal si et seulement si $F$ est courbe. Ceci ne peut se produire que lorsque $m$ est pair.

$\delta_F$ est maximal si et seulement si $F$ est parfaitement non linéaire (ou de manière équivalente courbe). Ceci ne peut se produire que lorsque $m$ est pair.

Rappelons que si $F$ est presque courbe, alors $F$ est presque parfaitement non linéaire, la réciproque étant fausse.

**Propriétés des fonctions booléennes**

L'étude des fonctions booléennes et de leurs propriétés est incontournable lorsque l'on cherche à modéliser des primitives cryptographiques conventionnelles. L'étude des fonctions booléennes vectorielles est naturellement adaptée à l'étude des boîtes de diffusion et de confusion d'un algorithme de chiffrement itératif par blocs.

**Définition B.1.1** *(Fonction booléenne vectorielle ). On appelle fonction booléenne vectorielle à $m$ variables et $n$ composantes toute application $f$ de $\mathbb{F}_2^m$ dans $\mathbb{F}_2^n$. Lorsque $n = 1$, on parle de fonction booléenne scalaire ou plus simplement de fonction boléenne.*

Nous donnons à présent quelques unes des propriétés des fonctions booléennes (scalaires). L'étude de ces propriétés va nous servir de base pour l'étude des propriétés des fonctions booléennes vectorielles et des critères de sécurité qu'elles induisent sur les primitives cryptographiques de type chiffrement itératif par blocs.

**Définition B.1.2** *(Support, poids et distance de Hamming d'un vecteur de $\mathbb{F}_2^m$). Soient $x, y \in \mathbb{F}_2^m$, on définit :*

- *Le support de $x$ : $supp(x) = \{u \in \{1, \ldots, m\}\ x_u \neq 0\}$. $supp(x)$ est l'ensemble des indices des composantes non nulles de $x$.*
- *Le poids de Hamming de $x$ : $wt(x) = \#supp(f) = \sum_{u=1}^{m} x_u$. $wt(x)$ est le nombre de composantes non nulles de $x$.*
- *La distance de Hamming de $x$ à $y$ : $d(x, y) = wt(x \oplus y)$. $d(x, y)$ est le nombres de composantes de valeur différentes de $x$ et $y$.*

**Définition B.1.3** *(Support, poids et distance de Hamming d'une fonction booléenne). Soient $f, g : \mathbb{F}_2^m \to \mathbb{F}_2$, on définit :*

- *Le support de $f$ : $supp(f) = \{u \in \mathbb{F}_2^m\ f(u) \neq 0\}$. $supp(f)$ est l'ensemble des vecteurs de $\mathbb{F}_2^m$ pour lesquels la valeur de la fonction $f$ est non nulle.*
- *Le poids de Hamming de $f$ : $wt(f) = \#supp(f)$. $wt(f)$ est le cardinal de son support, c'est-à-dire le poids de Hamming de son vecteur de valeurs $(f(e_u),\ u = 0, \ldots, 2^m - 1)$.*
- *La distance de Hamming de $f$ à $g$ : $d(f, g) = wt(f \oplus g)$. La distance entre deux fonctions booléennes est le nombre de vecteurs en lesquels leurs valeurs diffèrent.*

On note $d(f, \mathcal{E})$ la distance d'une fonction booléenne $f$ à un ensemble de fonction $\mathcal{E}$. On a :

$$d(f, \mathcal{E}) = \min_{g \in \mathcal{E}} wt(f \oplus g).$$

On définit le produit scalaire dans $\mathbb{F}_2^m$ par

$$u, v \in \mathbb{F}_2^m,\ < u, v >= \bigoplus_{i=1}^{m} u_i v_i.$$

Nous notons par la suite $\varphi_y$ la fonction booléenne linéaire définie par $\varphi_y(x) =< x, y >$.

**Représentation d'une fonction booléenne** Nous pouvons représenter une fonction booléenne $f$ à $n$ variables sous différentes formes :

- sa forme algébrique normale (ANF), qui est la représentation de la fonction de manière formelle sous la forme d'un polynôme multivarié (à $m$ variables) ;
- sa table de vérité, c'est-à-dire un vecteur de longueur $2^m$ indiquant la valeur de la fonction pour chacune des entrées ;

– son spectre de Walsh-Hadamard, la table de vérité de la transformée de Fourier discrète appliquée à la fonction signe de $f$. C'est également un vecteur de taille $2^m$.

Chacune de ces représentation est équivalente et correspond de manière univoque à la fonction $f$.

**Définition B.1.4** *(Forme Algébrique Normale). La Forme Algébrique Normale d'une fonction booléenne à $m$ variables est l'unique polynôme $P_f$ de $\mathbb{F}_2^n[x_1, \ldots, x_m]/(x_1^2 - x_1, \ldots, x_m^2 - x_m)$ tel que $\forall x \in \mathbb{F}_2^m$, $f(x) = P_f(x)$. On appelle degré de $f$ et on note $deg(f)$ le degré du polynôme $P_f$. Une fonction booléenne de degré 1 est dite affine, et si de plus elle est nulle en 0, elle est dite linéaire.*

C'est le théorème d'interpolation de Lagrange qui assure que toute fonction booléenne admet une unique représentation sous la forme d'un polynôme multivarié.

**Définition B.1.5** *(Transformée de Möbius). Soit $f$ une fonction booléenne à $m$ variables. On définit sa transformée de Möbius comme la fonction $f^o : \mathbb{F}_2^m \to \mathbb{F}_2$ telle que $\forall u \in \mathbb{F}_2^m$, $f^o(u) = \bigoplus_{v \leq u} f(v)$ (avec $v \leq u \Leftrightarrow \forall i$, $v_i = 1 \Rightarrow u_i = 1$).*

On montre par récurrence sur $m$ le résultat suivant :

**Proposition 8** *La Forme Algébrique Normale d'une fonction booléenne $f$ à $m$ variables est égale à*

$$f(x) = \bigoplus_{u=(u_1,\ldots,u_m)\in\mathbb{F}_2^m} f^o(u) x_1^{u_1} \ldots x_n^{u_m}.$$

Nous pouvons remarquer que le degré $deg(f)$ d'une fonction booléenne $f$ est égale au poids maximum du vecteur $u = (u_1, \ldots, u_m)$ tel que le coefficient $f^o(u)$ de la forme algébrique normale de $f$ est non nul :

$$deg(f) = \max_{u\in\mathbb{F}_2^m,\, f^o(u)\neq 0} wt(u).$$

**Définition B.1.6** *(Table de vérité). Soit $f : \mathbb{F}_2^m \to \mathbb{F}_2$. On appelle table de vérité de $f$ l'ensemble :*

$$\{(x, f(x)),\, x \in \mathbb{F}_2^m\}.$$

La table de vérité de $f$ permet de représenter $f$ par le vecteur des valeurs de $f$ sur $\mathbb{F}_2^m$ : $(f(e_0), \ldots, f(e_{2^m-1}))$ où $(e_0, \ldots, e_{2^m-1})$ sont les éléments de $\mathbb{F}_2^m$ rangés dans un certain ordre. Cette représentation permet bien de représenter $f$ de manière univoque dans la mesure où il existe un isomorphisme canonique entre les espaces vectoriels $\mathbb{F}_2^m$ et $\mathbb{F}_{2^m}$.

**Définition B.1.7** *(Transformée de Fourier). Soit $f : \mathbb{F}_2^m \to \mathbb{F}_2$ une fonction booléenne. Sa transformée de Fourier est la fonction $\widehat{f} : \mathbb{F}_2^m \to \mathbb{R}$ définie par*

$$\widehat{f}(u) = \sum_{v\in\mathbb{F}_2^m} f(v)(-1)^{<u,v>} \ .$$

On montre par récurrence sur $m$ le lemme

$$\text{sur } \mathbb{F}_2^m,\ \widehat{1} = 2^m \delta_0 \ .$$

On établit la transformée inverse de $f$ par

$$\text{sur } \mathbb{F}_2^m,\ \widehat{\widehat{f}} = 2^m f \ .$$

Soit $f : \mathbb{F}_2^m \to \mathbb{F}_2$. On note $\mathcal{X} = \mathcal{X}_{Id}$ la fonction signe i.e la fonction $\mathcal{X} : \mathbb{F}_2^m \to \{-1, 1\}$ définie par

$$\mathcal{X}_{Id} = 1 - 2\mathrm{id} \ ,$$

et on étend cette définition à celle de fonction signe de $f$ :

$$\mathcal{X}_f = \mathcal{X}_{\mathrm{id}} \circ f = (-1)^f = 1 - 2f \ .$$

**Définition B.1.8** *(Transformée de Walsh-Hadammard). La transformée de Walsh-Hadamard de $f$ est la fonction $\widehat{\mathcal{X}}_f$.*

La transformée de Walsh-Hadammard d'une fonction booléenne $f$ est définie comme la transformée de Fourier de la fonction signe de $f$. Elle représente la corrélation de la fonction signe de $f$ avec la fonction signe des fonctions linéaires :

$$\widehat{\mathcal{X}}_f(x) = \sum_{y\in\mathbb{F}_2^m} (-1)^{f(y)}(-1)^{<x,y>} = \sum_{y\in\mathbb{F}_2^m} (-1)^{f(y)\oplus<x,y>} = \sum_{y\in\mathbb{F}_2^m} (-1)^{f(y)\oplus\varphi_x(y)}.$$

Notons $c_{f,g}$ la corrélation croisée des fonctions booléennes $f$ et $g$, définie par [Sar02] :

$$c_{f,g}(x) = \sum_{y\in\mathbb{F}_2^m} (-1)^{f(y)\oplus g(x\oplus y)}.$$

La corrélation des fonctions $f$ et $g$ est définie par :

$$c(f,g) = c_{f,g}(0) = \sum_{y\in\mathbb{F}_2^m} (-1)^{f(y)\oplus g(y)}.$$

On a donc :

$$\widehat{\mathcal{X}}_f(x) = c(f,\varphi_x).$$

La transformée de Walsh-Hadammard est par définition inversible, ce qui garantit qu'elle permet de représenter $f$ de manière univoque.

On a le résultat suivant :

$$\text{sur } \mathbb{F}_2^m,\ \widehat{\mathcal{X}}_f = 2^m\delta_0 - 2\widehat{f}\ ,$$

et on retrouve la formule de Parseval

$$\sum_{\mathbb{F}_2^m} \widehat{\mathcal{X}}_f^2 = 2^{2m}\ .$$

Nous pouvons remarquer que

$$\widehat{f}(0) = wt(f)\ ,$$

et que

$$\widehat{\mathcal{X}_f}(0) = 2^m - 2wt(f)\ .$$

**Définition B.1.9** *(Spectre de Walsh-Hadammard). On appelle spectre de Walsh-Hadammard de $f$ la table de vérité de la transformée de Fourier de la fonction signe de $f$ :*

$$\{\widehat{\mathcal{X}}_f(y) = c(f,\varphi_y),\ y\in\mathbb{F}_2^m\}.$$

**Définition B.1.10** *(Dérivées d'une foncion booléenne). Soit $f:\mathbb{F}_2^m\to\mathbb{F}_2$ et $\tau_y$ la fonction de translation définie par $\tau_y(x) = x\oplus y$. On définit la dérivée de $f$ relativement au vecteur $y$ par $D_yf(x) = f(x)\oplus f\circ\tau_y(x) = f(x)\oplus f(x\oplus y)$.*

*On étend récursivement la définition de dérivée au cas des dérivées d'ordre supérieur : Soit $y_1,\ldots,y_r$ $r$ vecteurs linéairement indépendants et $\mathcal{F}\subset\mathbb{F}_2^m$ le sous-espace vectoriel engendré par ces $r$ vecteurs. On définit la dérivée d'ordre $r$ de $f$ relativement à $\mathcal{F}$ par :*

$$D_{\mathcal{F}}f = \bigoplus_{y\in\mathcal{F}} f\circ\tau_y.$$

L'auto-corrélation de $f$ doit mesurer la corrélation de $f$ avec ses décalées $f\circ\tau_y$.

$$c(f,f\circ\tau_y) = c_{f,f\circ\tau_y}(0) = \sum_{x\in\mathbb{F}_2^m} (-1)^{f(x)\oplus f\circ\tau_y(x)}$$

$$= \sum_{x\in\mathbb{F}_2^m} (-1)^{f(x)\oplus f(x\oplus y)} = \sum_{x\in\mathbb{F}_2^m} (-1)^{D_yf(x)}.$$

Le spectre d'auto-corrélation est donc naturellement définit comme suit :

**Définition B.1.11** *(Spectre d'auto-corrélation). Soit $f : \mathbb{F}_2^m \to \mathbb{F}_2$. On appelle spectre d'auto-corrélation de $f$ l'ensemble :*

$$\{\widehat{\mathcal{X}}_{D_y f}(0) = c(f, f \circ \tau_y),\ y \in \mathbb{F}_2^m\}$$

**Propriétés des fonctions booléennes**

**Définition B.1.12** *(Degré algébrique d'une fonction booléenne). Le degré algébrique d'une fonction booléenne est le degré du polynôme représentant sa forme algébrique normale.*

**Définition B.1.13** *(Equilibre d'une fonction booléenne). Une fonction booléenne $f$ est dite équilibrée lorqu'elle prend autant de fois la valeur $0$ que la valeur $1$.*

La table de vérité d'une fonction booléenne équilibrée doit donc contenir autant de zéros que de uns.

En d'autre termes, si $X_1, \ldots, X_m$ sont des variables aléatoires indépendantes et uniformément distribuées, $f$ est équilibrée si la variable aléatoire $f(X)$ est uniformément distribuée.

Cette condition impose un spectre de Walsh-Hadamard nul en zéro :
$f$ est équilibrée si

$$wt(f) = 2^{m-1}\ ,$$

ou de manière équivalente si

$$\widehat{\mathcal{X}_f}(0) = 0\ .$$

Nous voyons donc que $\widehat{\mathcal{X}_f}(0)$ permet de caractériser le déséquilibre de $f$. Plus précisemment, $|\widehat{\mathcal{X}_f}(0)|$ permet de mesurer la distance de $f$ à l'ensemble des fonctions équilibrées. En effet, si on note $\mathcal{E}_m$ l'ensemble des fonctions booléennes à $m$ variables équilibrées, on a :

$$|\widehat{\mathcal{X}_f}(0)| = 2.d(f, \mathcal{E}_m).$$

Le critère d'immunité aux corrélations permet dans la mesure du possible de se prémunir des attaques tirant profit d'une éventuelle corrélation entre une entrée de la fonction et une des sorties.

Il s'agit de conserver la distribution des valeurs de la fonction lorsque l'on fixe arbitrairement une ou plusieurs variables.

**Définition B.1.14** *(Ordre de résistance aux corrélations). On dira d'une fonction qu'elle est immune aux corrélations d'ordre $t$ si et seulement si la fonction conserve sa distribution de valeur lorsque l'on fixe $t$ variables.*

En d'autres termes, $f$ est résistante aux corrélations d'ordre $t$ si, pour tout sous-ensemble $T$ des indices de cardinal inférieur ou égal à $t$, $f$ est sans corrélation par rapport à $T$, c'est-à-dire que la distribution de probabilité de la sortie $f(X)$ est inchangée quand les entrées indicées par $T$ sont fixées (et que les autres forment un ensemble de variables aléatoires indépendantes et uniformément distribuées).

Une fonction booléenne $f$ est résistante aux corrélations à l'ordre $t$ si et seulement son spectre de Walsh-Hadamard s'annule sur tout mot non nul de poids au plus $t$.

Une fonction booléenne $f$ est dite sans corrélation d'ordre $k$ si sa distribution de valeur ne change pas lorque l'on fixe au plus $k$ entrées, i.e que si $X_1, \ldots, X_m$ sont des v.a. i.i.d., la variable aléatoire $f(X)$ est indépendante de $X_{i_1}, \ldots, X_{i_k}$ $\forall 1 \leq i_1 < \ldots < i_k \leq m$ ou de manière équivalente, $f(X)$ est indépendante de $< u, (X_{i_1}, \ldots, X_{i_k}) >$ $\forall 1 \leq i_1 < \ldots < i_k \leq m,\ \forall u \in \mathbb{F}_2^k$ ce qui équivaut à $f(X)$ est indépendante de $< u, X >\ \forall u\ wt(u) \leq k$. Une v.a. étant toujours indépendante d'une constante, on peut considérer $u \neq 0$. On a donc la condition équivalente

$$f(X) \text{ est indépendante de } < u, X >\ \forall u \neq 0\ wt(u) \leq k$$

On établit le lemme suivant : soit $u \in \mathbb{F}_2^m,\ u \neq 0$ ,

$$\widehat{f}(u) = 0 \text{ ssi } f(X) \text{ est indépendante de } < u, X >\ .$$

On en déduit le théorème de Xiao et Massey : $f$ est sans corrélation d'ordre $k$ ssi

$$\widehat{f}(u) = 0,\ \forall u\ 1 \leq wt(u) \leq k\ ,$$

où

$$wt(u) = \#\{i\ u_i \neq 0\}\ .$$

Ce théorème admet le corollaire suivant : $f$ est sans corrélation d'ordre $k$ ssi

$$\widehat{\mathcal{X}}_f(u) = 0,\ \forall u\ 1 \leq wt(u) \leq k\ .$$

**Définition B.1.15** *(Ordre de résilience). Une fonction booléenne est $t$-résiliente lorsqu'elle est $t$-immune et équilibrée.*

Une fonction booléenne $f$ est $t$-résiliente si et seulement si son spectre de Walsh-Hadamard s'annule sur tout mot de poids au plus $t$.

Une fonction équilibrée et sans corrélation d'ordre $k$ est dite $k$-résiliente. Le théorème de Xiao et Massey admet le corollaire suivant : $f$ est $k$-résiliente ssi

$$\widehat{f}(0) = 2^{m-1} \text{ et } \widehat{f}(u) = 0,\ \forall u\ 1 \leq wt(u) \leq k\ .$$

Ou de manière équivalente si

$$\widehat{\mathcal{X}}_f(u) = 0,\ \forall u\ 0 \leq wt(u) \leq k\ .$$

**Définition B.1.16** *(Non linéarité). La non-linéarité d'une fonction booléenne $f$ est la plus petite distance de $f$ à l'espace des fonctions affines.*

On définit l'ensemble $\mathcal{A}$ des fonctions affines

$$\mathcal{A} = \{a(x) =< u, x >,\ u \in \mathbb{F}_2^m\} \bigcup \{a(x) = 1 \oplus < u, x >,\ u \in \mathbb{F}_2^m\}\ ,$$

et on définit la non-linéarité d'une fonction booléenne

$$nl(f) = \min_{a \in \mathcal{A}} d(f, a)\ .$$

La non-linéarité s'exprime en fonction du spectre de W.-H.

$$nl(f) = \frac{1}{2}\left[2^m - \max_{u \in \mathbb{F}_2^m} |\widehat{\mathcal{X}}_f(u)|\right]\ .$$

On déduit de la formule de Parseval que si $m$ est pair, $nl(f)$ est maximale ssi

$$\forall u \in \mathbb{F}_2^m,\ \widehat{\mathcal{X}}_f(u) =_-^+ 2^{m/2}\ .$$

Une telle fonction $f$ est dite courbe.

On a le résultat suivant :

**Proposition 9** *Soit $f$ une fonction booléenne à $m$ variables. $f$ est courbe si et seulement si la fonction dérivée $D_y f : \mathbb{F}_2^m \leftrightarrow \mathbb{F}_2$ définie par $D_y f(x) = f(x \oplus y) \oplus f(x)$ est équilibrée pour tout vecteur non nul $y$ de $\mathbb{F}_2^m$.*

**Définition B.1.17** *(Critère de propagation $PC(l)$). Une fonction booléenne à $m$ variables satisfait le critère de propagation de degré $l$, noté $PC(l)$, si pour tout mot $y \in \mathbb{F}_2^m$ tel que $0 < wt(y) \leq l$, la fonction $D_y f$ définie sur $\mathbb{F}_2^m$ par $D_y f(x) = f(x \oplus y) \oplus f(x)$ est équilibrée.*

Le critère de propagation de degré 1, $PC(1)$, est connu sous le nom de *strict avalanche criterion*. Une fonction booléenne $f$ à $m$ variables qui satisfait au critère $PC(m)$ est dite parfaitement non-linéaire, ou de manière équivalente, courbe.

La non-linéarité impose une distribution constante (uniforme) du spectre de Walsh-Hadammard.

Ce critère est naturellement incompatible avec les autres critères de résistance aux corrélations, ce qui va contraindre le concepteur à faire un compromis.

**Compromis sur les exigences de sécurité d'une fonction booléenne**
**Théorème de siegenthaler** Si on note $nl$ la non linéarité, $d$ le degré, $t$ l'ordre d'immunité aux corrélations et $e$ une variable qui vaut 1 si la fonction est équilibrée, 0 sinon, on a

$$t + d + e \leq nl \ .$$

**Visualisation** On a vu que les propriétés des fonctions booléennes s'expriment de manière assez précise en fonction de leur
- transformée de Fourier discrète ;
- spectre de Walsh-Hadamard ;
- spectre d'auto-corrélation.

Cela induit une approche assez visuelle, permettant, avec un peu d'habitude, de statuer assez rapidement sur les qualités d'une fonction booléenne, compte tenu du rôle auquel on la destine :

- Le spectre de Walsh-Hadammard $\{\widehat{\mathcal{X}}_f(x) = c(f, \varphi_x),\ x \in \mathbb{F}_2^m\}$ est compris entre $-2^m$ et $2^m$.
- Nous avons vu que la valeur du spectre en 0 mesure la distance de $f$ à l'ensemble des fonctions équilibrées (qui est aussi la corrélation de $f$ avec la fonction nulle). Plus cette valeur est importante, plus la fonction $f$ est déséquilibrée.
- L'amplitude maximale $l_f$ de $|\widehat{\mathcal{X}}_f(x)| = |c(f, \varphi_x)|$ mesure la corrélation maximale de $f$ avec les fonctions linéaires. $l_f = \max |\widehat{\mathcal{X}}_f(x)|$ est minimale pour les fonctions courbes lorsque $m$ est pair et pour les fonctions presques courbes lorsque $m$ est impair. Dans les deux cas, le spectre de Walsh-Hadammard a un profil bien particulier : il ne comporte que les valeurs $\pm 2^{\frac{m}{2}}$ lorsque $f$ est courbe (ou de manière équivalente parfaitement non linéaire) ; et ne comporte que les valeurs 0 et $\pm 2^{\frac{m+1}{2}}$ lorsque $f$ est presque courbe.

- Le spectre d'auto-corrélation $\{\widehat{\mathcal{X}}_{D_x f}(0) = c(f, f \circ \tau_x),\ x \in \mathbb{F}_2^m\}$ est compris entre $-2^m$ et $2^m$.
- Nous avons vu que le spectre d'auto-corrélation mesure la corrélation de $f$ avec ses décalées. Le spectre d'auto-corrélation d'une fonction courbe a un profil bien particulier, puisqu'il est nul partout sauf en 0. Cela signifie que toutes les dérivées de $f$ sauf la dérivée relativement au vecteur nul sont équilibrées.

À titre d'exemple, les figures B.1, B.2 et B.3 montrent les spectres de Walsh des fonctions de rétroaction du système de chiffrement par flot COSvD $(2, 128)$ [FFJ04]. Ces fonctions booléennes illustrent le compromis qui peut être fait entre haute non-linéarité, immunité aux corrélations et équilibre. Le tableau B.1 donne le résulat de l'application des critères de sécurité boîte présentés dans cette section aux fonctions *f9a*, *f9b* et *f11*.

| $f$ | deg(f) | OIC(f) | $NL(f)$ | $NLmax$ |
|---|---|---|---|---|
| $f11$ | 7 | CI(2) | 960 | 992 |
| $f9a$ | 6 | CI(2) | 224 | 240 |
| $f9b$ | 6 | CI(2) | 224 | 240 |

Tab. B.1 – Caractéristiques des fonctions de rétroaction du système de chiffrement COSvd $(2, 128)$

**Corrélation croisée des fonctions booléennes** Nous avons introduit au paragraphe précédent $c_{f,g}$, la corrélation croisée des fonctions booléennes $f$ et $g$, définie par :

$$c_{f,g}(x) = \sum_{y \in \mathbb{F}_2^m} (-1)^{f(y) \oplus g(x \oplus y)}.$$

La corrélation des fonctions $f$ et $g$ est définie par :

$$c(f, g) = c_{f,g}(0) = \sum_{y \in \mathbb{F}_2^m} (-1)^{f(y) \oplus g(y)}.$$

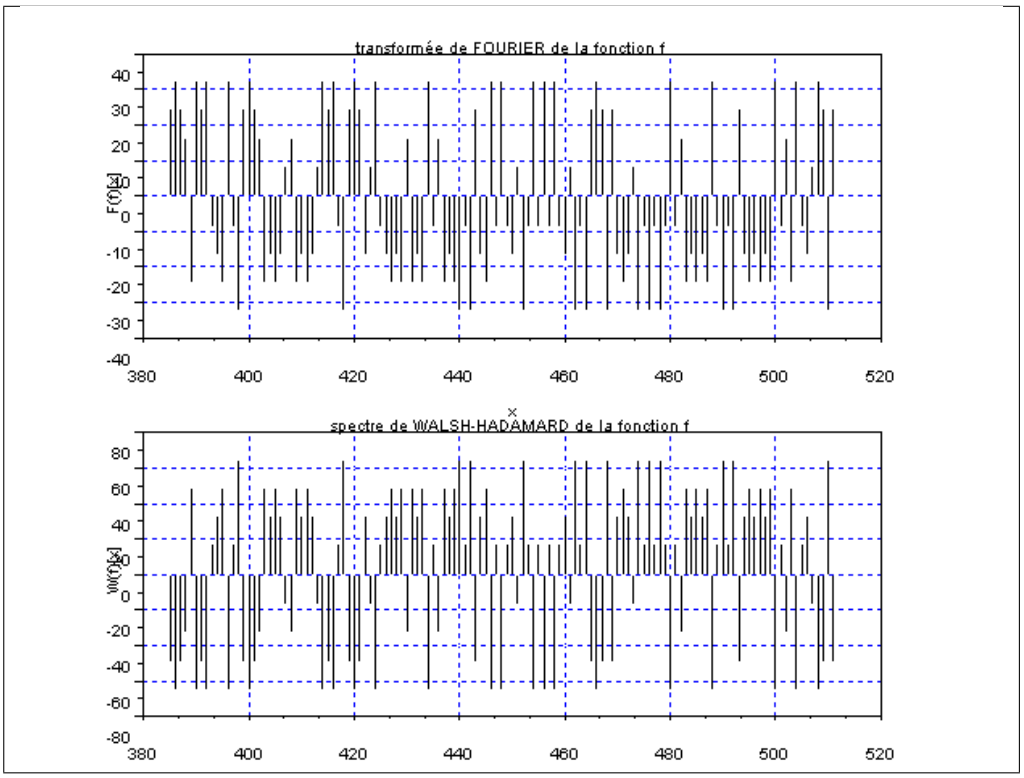



FIG. B.1 – Transformée de Fourier et spectre de Walsh de la fonction *f9a*

On a donc :

$$\widehat{\mathcal{X}}_f(x) = c(f, \varphi_x).$$

Cette notion permet de caractériser la notion de confusion d'une fonction booléenne vectorielle par la faible corrélation croisée de ses composantes (prises deux à deux).

**Définition B.1.18** *Deux fonction booléennes $f, g : \mathbb{F}_2^m \to \mathbb{F}_2^n$ sont parfaitement décorrélées si :*

$$\forall x \in \mathbb{F}_2^m,\ c_{f,g}(x) = 0.$$

Avec cette définition, on peut faire l'hypothèse que si les composantes d'une fonction booléenne vectorielle sont deux à deux parfaitement décorrélées, alors elle impose la meilleure confusion possible au cryptosystème.

Une forme plus faible de cette notion peut être obtenue en restreignant l'ensemble des $x \in \mathbb{F}_2^m$ pour lesquels $c_{f,g}(x) = 0$ :

**Définition B.1.19** *Deux fonction booléennes $f, g : \mathbb{F}_2^m \to \mathbb{F}_2^n$ sont $t$-décorrélées (ou décorrélées à l'ordre $t$) si :*

$$\forall x \in \mathbb{F}_2^m,\ \textit{tel que}\ 0 \leq wt(x) \leq t,\ c_{f,g}(x) = 0.$$

Une autre approche peut consister à imposer que $\forall x \in \mathbb{F}_2^m$, $|c_{f,g}(x)|$ est borné par une constante suffisamment petite. cela permet d'assurer que la corrélation croisée entre les composantes d'une fonction vectorielle sont uniformément petites.

### Propriétés des fonctions booléennes vectorielles

**Définition B.1.20** *(Spectre de Walsh-Hadammard d'une fonction vectorielle). Le spectre de Walsh-Hadammard d'une fonction booléenne vectorielle $F : \mathbb{F}_2^m \to \mathbb{F}_2^n$ est composé des spectres de Walsh-Hadammard des fonctions $\varphi_y \circ F$, $y \in \mathbb{F}_2^n - \{0\}$. Il correspond donc à l'ensemble :*

$$\{\widehat{\mathcal{X}}_{\varphi_y \circ F}(x),\ x \in \mathbb{F}_2^m,\ y \in \mathbb{F}_2^n - \{0\}\}.$$

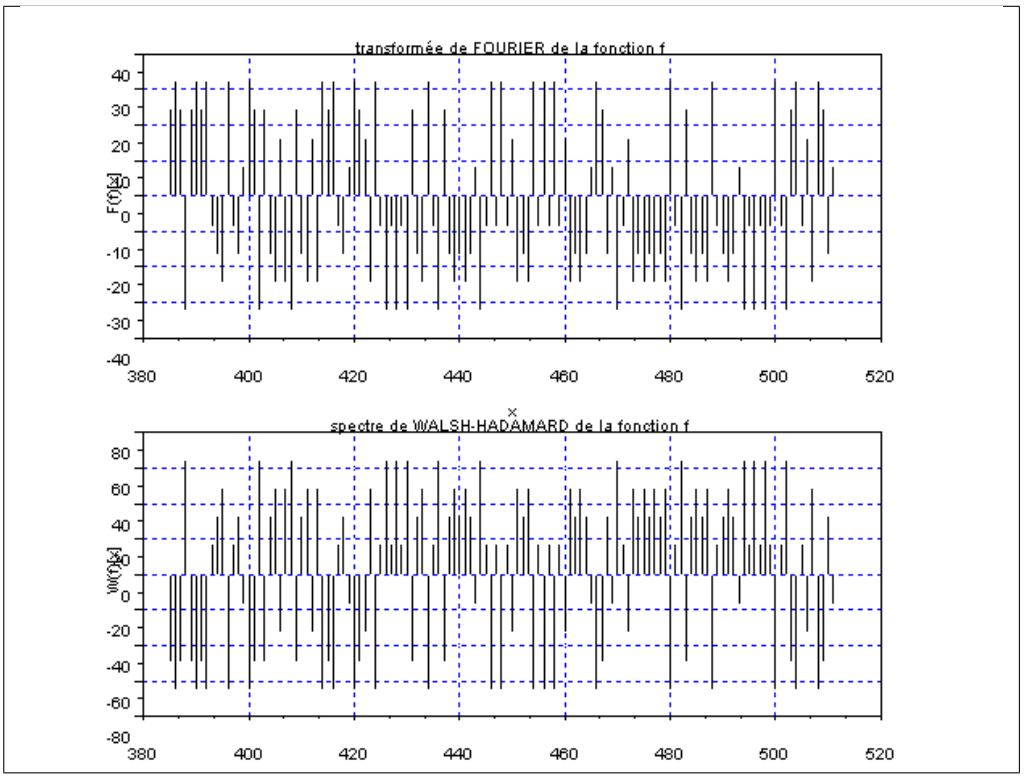



FIG. B.2 – Transformée de Fourier et spectre de Walsh de la fonction *f9b*

On note $l_F$ le coefficient de Walsh-Hadammard d'amplitude maximale du spectre de Walsh-Hadammard de la fonction vectorielle $F$ et on définit la non-linéarité de $F$ par :

$$nl_F = \frac{1}{2}(2^m - l_F).$$

**Définition B.1.21** *(Fonction presque courbe). Soit $F : \mathbb{F}_2^m \to \mathbb{F}_2^m$. $F$ est dite presque courbe lorsque $l_F = 2^{\frac{m+1}{2}}$.*

Le spectre de Walsh-Hadammard d'une fonction presque courbe $F : \mathbb{F}_2^m \to \mathbb{F}_2^m$ ne contient que les valeurs 0 et $\pm 2^{\frac{m+1}{2}}$.

**Définition B.1.22** *(Fonction courbe). Soit $F : \mathbb{F}_2^m \to \mathbb{F}_2^n$. $F$ est dite courbe lorsque $l_F = 2^{\frac{m}{2}}$.*

**Définition B.1.23** *(Fonction presque parfaitement non-linéaire). Soit $1_y$ la fonction indicatrice de $y \in \mathbb{F}_2^m$ et soit $F : \mathbb{F}_2^m \to \mathbb{F}_2^m$. $F$ est dite presque parfaitement non-linéaire lorsque :*

$$\max_{x \in \mathbb{F}_2^m, y \in \mathbb{F}_2^m - \{0\}} wt(1_y \circ D_x F) = 2.$$

**Définition B.1.24** *(Fonction parfaitement non-linéaire). Soit $1_y$ la fonction indicatrice de $y \in \mathbb{F}_2^n$ et soit $F : \mathbb{F}_2^m \to \mathbb{F}_2^n$. $F$ est dite parfaitement non-linéaire lorsque :*

$$\max_{x \in \mathbb{F}_2^m, y \in \mathbb{F}_2^n - \{0\}} wt(1_y \circ D_x F) = 2^{m-n}.$$

Les fonctions parfaitement non-linéaires n'existent que lorsque $m$ est pair et $m \geq 2n$. Elles correspondent exactement aux fonctions courbes.

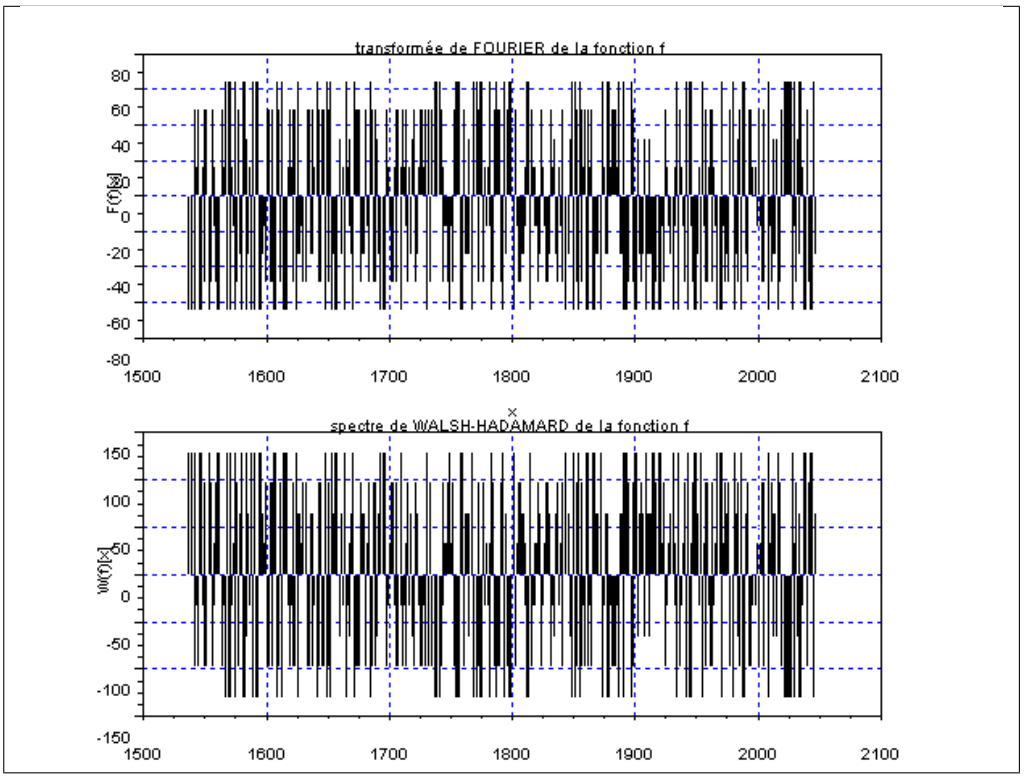



FIG. B.3 – Transformée de Fourier et spectre de Walsh de la fonction *f11*

**Critères de résistance à la cryptanalyse Linéaire**

Cette attaque à clair connu s'applique à tous les systèmes de chiffrement itératifs par blocs.

Son principe général est d'approcher la fonction itérée $F(X,K) : \mathbb{F}_2^m \times \mathbb{F}_2^l \rightarrow \mathbb{F}_2^m$ par une fonction linéaire. Il s'agit donc de trouver $\alpha \in \mathbb{F}_2^m$, $\beta \in \mathbb{F}_2^m$ et $\gamma \in \mathbb{F}_2^l$ tels que : $\alpha.F(X,K) = \beta.X \oplus \gamma.K$ pour un nombre significatif de valeurs de $X \in \mathbb{F}_2^m$. En chaînant de telles équations, on aboutit à une approximation linéaire des $r$ tours de l'algorithme de chiffrement, de la forme :

$$\alpha.Y(0) \oplus \beta.Y(r) \oplus \gamma.(K_1 \| \ldots \| K_r) = 0$$

avec $\alpha \in \mathbb{F}_2^m$, $\beta \in \mathbb{F}_2^m$ et $\gamma \in (\mathbb{F}_2^l)^r$. Matsui a utilisé cette méthode pour modéliser l'action du DES en terme d'approximations linéaires.

Soit $p$ la probabilité que la relation $\alpha.Y(0) \oplus \beta.Y(r) \oplus \gamma.(K_1 \| \ldots \| K_r) = 0$ soit satisfaite pour un ensemble de $N$ textes clairs. L'algorithme de la cryptanalyse linéaire est le suivant :

1. Calculer $T = \#\{Y(0) \,|\, \alpha.Y(0) \oplus \beta.Y(r) = 0\}$.
   (a) Si $T > N/2$, alors $\gamma.K$ vaut 0 si $p > 1/2$ et 1 sinon.
   (b) Si $T < N/2$, alors $\gamma.K$ vaut 1 si $p > 1/2$ et 1 sinon.

Le taux de succès de cette attaque dépend du nombre de couples clair/chiffré $(Y(0), Y(r))$ considérés et est égal à :

$$\int_{-2\sqrt{N}|p-1/2|}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx.$$

Cette attaque exploite donc le fait que la fonction itérée puisse être approchée par une fonction linéaire. Puisque la non-linéarité d'une fonction est invariante par composition avec une transformation affine, il est donc équivalent de

considérer que cette attaque exploite le fait que l'action des fonctions utilisées dans l'étape de substitution puisse être approchée par une fonction linéaire.

Notons $S : \mathbb{F}_2^m \to \mathbb{F}_2^n$ la fonction booléenne vectorielle représentant l'étape de substitution d'un tour du chiffrement itératif. Pour minimiser le taux de succès d'une cryptanalyse linéaire, il faut que pour tout $\alpha \in \mathbb{F}_2^m$, $\beta \in \mathbb{F}_2^n - \{0\}$, le cardinal de l'ensemble $\{X \in \mathbb{F}_2^m \,|\, \alpha.X \oplus \beta.S(X) = 0\}$ soit le plus proche possible de $2^{m-1}$. Les fonction $S$ qui conviennent sont donc celles qui minimisent

$$\lambda_S = |\max_{\alpha,\beta\neq 0} \#\{X \in \mathbb{F}_2^m \,|\, \alpha.X \oplus \beta.S(X) = 0\} - 2^{m-1}|.$$

Il s'agit donc de minimiser

$$\lambda_S = \frac{1}{2} \max_{x\in\mathbb{F}_2^m} |\widehat{\mathcal{X}}_S(x)| = \frac{1}{2} l_S.$$

En d'autre terme, on cherche à ce que la non-linéarité de $S$ :

$$nl_S = 2^{m-1} - \frac{1}{2} \max_{x\in\mathbb{F}_2^m} |\widehat{\mathcal{X}}_S(x)| = 2^{m-1} - \frac{1}{2} l_S$$

soit la plus élevée possible.

Pour toute fonction booléenne vectorielle $S : \mathbb{F}_2^m \to \mathbb{F}_2^n$, on a : $l_S \geq 2^{\frac{m}{2}}$ si $m \neq n$ et $l_S \geq 2^{\frac{m+1}{2}}$ si $m = n$.

Dans le cas où $m \neq n$, la fonction $S = (S_1, \ldots S_n)$ présente la meilleur résistance à la cryptanalyse linéaire si et seulement si $l_S = 2^{\frac{m}{2}}$, ou de manière équivalente si toutes les combinaisons linéaires $f$ des fonctions $S_i$, $i = 1, \ldots, n$ présentent la meilleur non-linéarité possible. Ceci ne peut se produire que lorsque $m$ est pair. De telles fonctions $f$ sont dites courbes. La fonction $S : \mathbb{F}_2^m \to \mathbb{F}_2^n$ est alors dite courbe également.

Dans le cas où $m = n$, la la fonction $S = (S_1, \ldots S_m)$ présente la meilleur résistance à la cryptanalyse linéaire si et seulement si $l_S = 2^{\frac{m+1}{2}}$. Ceci ne peut se produire que lorsque $m$ est impair[1]. La fonction $S : \mathbb{F}_2^m \to \mathbb{F}_2^m$ est alors dite presque courbe.

**Critères de résistance à la cryptanalyse différentielle**

Cette méthode de cryptanalyse s'applique dans le contexte d'une attaque à clair choisi. Le principe général consiste à générer des couples de textes en clair, vérifiant certaines conditions sur leurs différences, puis à analyser l'évolution des différences sur les chiffrés. Le principe de cette analyse est d'assigner différentes probabilités aux clés possibles. Cette attaque à clair choisi s'applique à tous les systèmes de chiffrement itératif par blocs.

En appliquant ce principe, Biham et Shamir ont trouvé une attaque contre le DES plus efficace que la recherche exhaustive.

Cette attaque exploite le fait que la fonction itérée $F(X, K) = F_K(X)$ d'un chiffrement itératif par blocs peut être cryptographiquement faible, c'est-à-dire que la connaissance d'un couple de sorties $(Y, Y')$ de $F$ et de la différence $\Delta(X)$ de leurs entrées donne une information sur la clé $K$. On soumet donc des couples $(X, X') = (Y(0), Y'(0))$ de textes clairs (dont la différence $\alpha = X \oplus X'$ est fixée) au chiffrement et on estime la valeur $\Delta Y(r-1)$ des différences de leur chiffrés après $r-1$ tours. Si $\Delta Y(r-1)$ prend la valeur $\beta$ avec une probabilité $P(\,\Delta Y(r-1) = \beta \,|\, \Delta Y(0) = \alpha)$ élevée, et que $F$ est cryptographiquement faible, alors il devient possible de retrouver la clé de tour $K_r$. Cette probabilité est appelée différentielle au tour $r-1$. L'algorithme de la cryptanalyse différentielle est le suivant :

1. En premier lieu, déterminer un couple $(\alpha, \beta) = (\Delta Y(0), \Delta Y(r-1))$ dont la probabilité $P(\,\Delta Y(r-1) = \beta \,|\, \Delta Y(0) = \alpha)$ est élevée.
2. Répéter les étapes suivantes jusqu'à ce qu'une des valeurs $k$ apparaisse plus souvent que les autres :
   (a) Choisir aléatoirement un texte clair $M$ et soumettre $M$ et $M \oplus \alpha$ au chiffrement.

[1] Lorsque $m$ est pair, on connait des fonctions telles que $l_S = 2^{\frac{m}{2}+1}$. On peut conjoncturer le fait que cette valeur soit minimale.

(b) Déterminer toutes les valeurs possibles $k$ de $K_r$, en supposant que $\Delta Y(r-1) = \beta$.

(c) Incrémenter le compteur des apparitions de la valeur $k$.

La complexité de cette attaque est égale au nombre de messages clairs choisis qu'il faut soumettre au chiffrement pour trouver la clé du dernier tour $K_r$. Si on note $p_m = \max_{\alpha \neq 0, \beta \neq 0} P(\Delta Y(r-1) = \beta \,|\, \Delta Y(0) = \alpha)$, et si $n = |X|$ désigne la taille des textes clairs choisis (en nombre de bits), la complexité de l'attaque est minorée par

$$\frac{2}{p_m - \frac{1}{2^n - 1}}.$$

On voit donc que l'attaque est hors de portée si toutes les différentielles au tour $r-1$ ont une probabilité proche de $\frac{1}{2^n-1}$. Pour cela, il faut que pour tout $\alpha \in \mathbb{F}_2^n - \{0\}$, la distribution de $\Delta Y(r-1)$ soit proche de la distribution uniforme sur $\mathbb{F}_2^n - \{0\}$.

Un chiffrement de Feistel, comme le DES, dont les boîtes $S$ sont définies par les fonctions vectorielles $S : \mathbb{F}_2^m \leftrightarrow \mathbb{F}_2^n$, résiste d'autant mieux à la cryptanalyse différentielle que la valeur $\delta_S$ est petite, avec

$$\delta_S = \max_{a \in \mathbb{F}_2^m - \{0\}, b \in \mathbb{F}_2^n} \#\{X \in \mathbb{F}_2^m \,|\, S(X \oplus a) \oplus S(X) = b\}.$$

On cherche donc à minimiser la proportion de valeur de $x \in \mathbb{F}_2^m$ pour lesquelles $D_a S(x) = S(x \oplus a) \oplus S(x) = b$, ce qui revient à minimiser le poids de la fonction booléenne $1_b \circ D_a S$ :

$$\delta_S = \max_{a \in \mathbb{F}_2^m - \{0\}, b \in \mathbb{F}_2^n} wt(1_b \circ D_a S).$$

Pour toute fonction vectorielle $S : \mathbb{F}_2^m \leftrightarrow \mathbb{F}_2^n$, on a :

- si $m \neq n$, $\delta_S \geq 2^{m-n}$. La fonction $S$ présente la meilleur résistance à la cryptanalyse différentielle si et seulement si $\delta_S = 2^{m-n}$. Les fonctions $S$ atteignant cette borne sont dites parfaitement non linéaires. De telles fonctions n'existent que si $m$ est pair et $m \geq 2n$.
- si $m = n$, $\delta_S \geq 2$. Les fonctions $S$ vérifiant $\delta_S = 2$ sont dites presque parfaitement non linéaires. On conjecture que si $m = n$ est pair, $\delta_S \geq 4$. Si cette conjecture est exacte, les fonctions presque parfaitement linéaires n'existent que pour $m = n$ impair.

On peut relier cette propriété à un critère concernant les $n$ composantes $S_i$ de la fonction vectorielle $S = (S_1, \ldots, S_n)$ de la manière suivante : Si la fonction $S$ utilisée dans le chiffrement de Feistel offre la meilleur résistance à la cryptanalyse différentielle, alors ses composantes $S_i$, $i = 1, \ldots, n$ vérifient toutes : $\delta_{S_i} = 2^{m-1}$.

On peut faire le bilan suivant sur les fonctions vectorielles $S : \mathbb{F}_2^m \to \mathbb{F}_2^n$ qui sont optimales au regard des cryptanalyses linéaires et différentielles :

- Si $m$ est pair, on a : $\lambda_S$ est optimal si et seulement si $\delta_S$ est optimal. La fonction $S$ est alors dite courbe, ou de manière équivalente, parfaitement non linéaire. De telles fonctions n'existent que si $m$ est pair et $m \geq 2n$.
- Si $m$ est impair, on a : $\lambda_S$ est optimal $\Rightarrow$ $\delta_S$ est optimal. Notons que la réciproque est fausse. La fonction $S$ est alors dite presque courbe. De telles fonctions n'existent que pour $m$ impair.

## B.2 Génération aléatoire de S-Boxes

Nous avons vu au chapitre 2 que la cryptanalyse de l'implémentation boîte blanche du DES [WMGP07] exploite la connaissance des propriétés en terme de propagation des S-Boxes du DES. En analysant la propagation d'une différence sur l'entrée d'une T-Box encodée sur plusieurs tours, il est possible de distinguer les T-Boxes non-linéaires des T-Boxes n'embarquant aucune S-Box du DES. L'étape suivante de cette cryptanalyse différentielle se fonde sur une implémentation boite noire traditionnelle du DES pour retrouver les T-Boxes non linéaires, en recouvrant partiellement la permutation aléatoire appliquée aux T-Boxes non-linéaires. Ensuite, une fois identifiés un ensemble de différences, il est possible de retrouver la clé embarquée dans l'implémentation boîte blanche.
Si nous définissons la complexité de la cryptanalyse comme le nombre de fonctions de tour de l'implémentation boîte blanche qu'il est nécessaire d'exécuter, la complexité est en $O(2^{14})$, pour un espace mémoire utile négligeable. Cette attaque est donc très efficace pour retrouver la clé embarquée.

En conclusion, cette attaque exploite la faiblesse de la fonction de tour du DES. Pour contrer une telle attaque, l'ultime ressort me parait être la randomisation des S-Boxes.

Dans cette section, j'étudie la possibilité de diversifier l'implémentation boîte blanche des algorithmes de chiffrement par blocs de type S/P, à travers la randomisation des S-Boxes.

Après avoir présenté les travaux déjà effectués dans ce domaine de recherche et rappelé les principaux critères de sécurité boîte noire s'appliquant aux S-Boxes, je propose une manière d'améliorer les implémentations boîte blanche des algorithmes de chiffrement itératifs par blocs, fondée sur la randomisation des S-Boxes. J'illustre ensuite cette proposition par les résultats empiriques obtenus.
J'ai développé un outil de génération et d'analyse de S-Boxes, fondé sur un algorithme naïf pour la génération aléatoire de fonctions hautement non-linéaires.
Je présente les résultats des expérimentations que j'ai menées pour étalonner le générateur de S-Boxes et les compare avec les résultats obtenus en appliquant les critères de sécurité boîte noire aux S-Boxes du DES et de l'AES.
Je présente enfin les limites de cette approche et les axes de recherche futures que j'identifie.

### B.2.1 Autres travaux

Les critères de robustesse en boîte noire d'un algorithme de chiffrement itératif par blocs contre les cryptanalyse linéaire et différentielle ont fait l'objet de recherches importantes. Le problème plus spécifique de la conception de S-Boxes adaptées a permi le développement de critères [WT86, DT91, AT93, MA96] et d'algorithmes déterministes permettant de construire des S-Boxes satisaisant certains de ces critères, la non-linéarité notamment. Certains de ces critères ont été brevetés [AM+00].
Concernant les méthodes de génération purement aléatoire de S-Boxes, peu de travaux ont été effectués. L'idée consistant à imposer des contraintes sur la forme algébrique normale des fonctions booléennes, que j'ai exploitée pour la conception du générateur de S-Boxes n'est cependant pas nouvelle [GCS06].
Ces méthodes de génération et de diversification ne s'appliquent qu'aux S-Boxes. Une autre option consiste à revoir la conception du début. Des travaux ont déjà été effectués dans ce domaine de recherche, principalement dans le contexte GBAC. Dans le cas de l'AES, nous pouvons citer notamment le *design* appelé *dual ciphers* [BB02], proposé par Barkan & Biham dans le contexte GBAC. Dans le cas du DES, nous pouvons citer le *design* FDE (*Fast Data Encryption*), proposé par Hendessi & Shafieinejad [SHG04].

### B.2.2 Crières de sécurité boîte noire des S-Boxes

Notons $\mathbb{F}_2$ le corps fini d'ordre 2 (ou plus simplement l'ensemble $\{0, 1\}$). Le poids de Hamming $wt(b)$ d'un vecteur binaire $b$ est simplement le nombre de 1 de ce vecteur (définition B.1.3). Une S-Box $S : \mathbb{F}_2^m \to \mathbb{F}_2^n$ peut être décomposée en une séquence de fonctions booléennes $f_i : \mathbb{F}_2 \to \mathbb{F}_2$, appelées les composantes ou les colonnes de la S-Box. Une fonction booléenne $f$ est dite équilibrée si sa table de vérité a le même nombre de 0 que de 1 (définition B.1.13). L'ensemble des fonction équilibrées est défini par :

$$\mathcal{E}_m = \{f \,|\, wt(f) = 2^{m-1}\}.$$

Une fonction booléenne qui n'est pas affine est dite non-linéaire. L'ensemble des fonctions affines est défini par :

$$\mathcal{A}_m = \{\langle y, .\rangle,\ \langle y, .\rangle \oplus 1 \,|\, y \in \mathbb{F}_2^m\}.$$

L'ensemble des combinaisons linéaires non triviale (i.e. exception faite de la combinaison linéaire nulle) des colonnes d'une S-Box est défini par :

$$\mathcal{C}_m(S) = \{\langle y, .\rangle \circ S \,|\, y \in \mathbb{F}_2^m \backslash \{0\}\}.$$

#### Non-linéarité d'une S-Box

La non-linéarité d'une fonction booléenne (définition B.1.16) peut être définie comme sa distance à l'ensemble des fonctions affines :

$$nl_f = \min\{wt(f + a) \,|\, a \in \mathcal{A}_m\}.$$

La non-linéarité d'une S-Box est définie comme la plus petite distance de toutes les combinaisons linéaires non triviale des colonnes de $S$ à l'ensemble des fonctions affines :

$$nl_S = \min\{nl_f \mid f \in \mathcal{C}_m\}.$$

Si les colonnes de $S$ sont proches d'une fonction linéaire, le processus de chiffrement peut être approximé par des équations linéaires. En conséquence, une haute non-linéarité fournit une résistance vis-à-vis des attaques linéaires. Rappelons que les fonctions booléennes optimalement non-linéaires sont dites courbes.

### Valeur xor d'une S-Box

La valeur xor d'une S-Box est définie comme la plus grande entrée de la table xor :

$$XOR(S) = \max\{wt(1_b o D_a S) \mid a \in F_2^m \backslash \{0\}, b \in F_2^m\}.$$

Nous attendons de la table xor que ses valeurs soient distribuées de manière presque constante et uniformément petites. Cette seconde propriété participe à la protection contre les cryptanalyses différentielles.

### Distance dynamique d'une S-Box

La distance dynamique d'une fonction fournit une mesure sur laquelle il est possible de fonder d'autres propriétés dynamiques telles que le critère de strict avalanche, SAC (*strict avalanche criterion*), le critère d'indépendance du bit de sortie, BIC (*output bit independance criterion*), ainsi que des "distances" précises à ces critères. La distance dynamique d'ordre $k$ est définie de la manière suivante :

$$DD_k(f) = \max\{\frac{1}{2}|2^{m-1} - wt(D_y f)| \mid wt(y) \in ]0, k]\}.$$

Elle correspond à la distance à SAC d'ordre $k$. Si une S-Box ne satisfait pas le critère SAC, certains bits de sortie ne vont dépendre seulement que de certains bits d'entrée. Il est possible d'utiliser cette dépendance dans une attaque à clair choisi dès lors que suffisemment de textes chiffrés et de textes en clairs sont disponibles. En outre, cette dépendance est également utiles pour les recherches d'espace de clé. Remarquons que le crière SAC d'ordre maximal est assuré si des fonctions courbes sont choisies pour les colonnes de la S-Box.

Le critère d'indépendance du bit de sortie impose que les bits de sortie d'une S-Box doivent changer indépendemment lorsqu'un seul bit d'entrée est inversé. La distance au critère BIC d'ordre $i$ est défini par :

$$DBIC_{i,j} = \max\{DD_j(S_y) \mid wt(y) = i\},$$

où $S$ est la matrice binaire $(2^m, n)$ correspondant à la S-Box.

En résumé, une haute non-linéarité assure la résistance contre les attaques linéaires, une distribution uniformément petite de la table xor assure la résistance contre les attaques diférentielles, les autres propriétés (critères de stricte avalance et d'indépendance du bit de sortie) assurent que la S-Box a de bonnes propriétés dynamiques et sont cruciales dans la conception d'un algorithme de chiffrement, particulièrement concernant la propriété de diffusion.

### Autres critères

D'autres critères sont important pour assurer la sécurité en boîte noire d'une S-Box. En premier lieu, les colonnes de la S-Box doivent être équilibrées :

$$\forall f_i, wt(f_i) = 2^{m-1}.$$

Par ailleurs, si nous considérons une S-Box comme une séquence de fonctions booléennes, nous devons prendre en considération d'autres critères tels que le degré algébrique (définition B.1.12) et l'ordre d'immunité aux corrélations (définition B.1.14). Dans le contexte BBAC, le problème de complétude se pose également. Pour une S-Box fixée, ce problème correspond à la question de la conception d'une fonction de tour adaptée et au nombre de tours requis pour assurer la sécurité de l'algorithme de chiffrement.
Malheureusement, toutes ces propriétés ne peuvent être satisfaites simultanément.

### B.2.3 Randomisation des S-Boxes

Je propose à présent une manière d'améliorer les implémentations boîte blanche des algorithmes de chiffrement itératifs par blocs, fondée sur la randomisation des S-Boxes. J'illustrerai ensuite cette proposition par les résultats empiriques obtenus.

#### Modes de génération

Les S-Boxes peuvent être générées soit de manière aléatoire, soit de manière déterministe ou algébrique (comme c'est le cas pour les algorithmes Safer, IDEA ou DES). L'avantage de la première approche est qu'il n'existe pas de structure mathématique simple qui puisse être potentiellement exploitée par une cryptanalyse. L'avantage de la seconde approche est que les S-Box ont une représentation plus compacte et peuvent être implémentées plus facilement dans les applications. Remarquons que les deux approches peuvent être utilisées conjointement pour la construction de S-Boxes. Cependant, les algorithmes déterministes connus sont coûteux et réduisent le niveau de diversité obtenu à travers la randomisation. J'ai décidé pour cette raison de ne pas les utiliser dans les expérimentations que j'ai menées.

#### Algorithme de génération

J'ai expérimenté un algorithme naïf pour la génération aléatoire de fonctions hautement (parfois parfaitement) non linéaires. Rappelons qu'une fonction booléenne peut être représentée de manière équivalente par une table de vérité (définition B.1.6), un spectre de Fourier (définition B.1.9) ou un polynôme multivarié appelé ANF (*Algebraic Normal Form*, définition B.1.4). L'idée de cet algorithme de génération aléatoire est d'imposer des contraintes sur le degré algébrique des monômes multivariés de l'ANF des fonctions booléennes, de manière à assurer une haute non-linéarité.
Partant de l'observation que le degré algébrique des monômes multivariés de la représentation ANF d'une fonction courbe vérifient : $cord(f) \leq m/2$, nous générons aléatoirement pour $ord = 0, \ldots, m/2$, $cmin$ et $cmax$ tels que $0 \leq cmin \leq cmax \leq C_m^{ord}$, puis un nombre de coefficients $cord \in [cmin, cmax]$. Nous fixons aléatoirement à 1 les coefficients d'ordre $cord$ de l'ANF. Cet algorithme de génération est itéré tant que la fonction booléenne obtenue n'est pas courbe (ou pas suffisamment non-linéaire). Chacune des composantes de la S-Box est ensuite équilibrée à la volée, de manière aléatoire.

Je vais présenter les résultats des expérimentations que j'ai menées sur les S-Boxes et les comparer avec les résultats obtenus en appliquant les critères de sécurité boîte noire aux S-Boxes du DES et de l'AES.

#### Description générale du programme de génération et d'analyse

J'ai développé un programme permettant d'analyser une S-Box fournie en entrée (ou générée aléatoirement) sous l'une des formes suivantes :
- forme algébrique normale des composantes,
- table de vérité (matrice) ou support des composantes,
- spectre de Walsh,
- lookup table.

L'analyse couvre les critères suivants :
- non-linéarité d'une S-Box,
- valeur xor d'un S-Box,
- degré du critère de propagation des composantes,
- distance dynamique à SAC d'ordre $k$ des composantes,
- ordre de non-corrélation des composantes (OIC, Ordre d'Immunité aux Corrélations),
- degré algébrique des composantes,
- distance dynamique à BIC d'ordre $(k, l)$ d'une S-Box.

Je donne les résultats de l'analyse pour le DES et l'AES.

### Analyse des S-Boxes du DES

Le tableau B.2 donne le résulat de l'application des critères de sécurité boîte noire présentés section B.2.2 aux S-Boxes du DES.

| Critères | S-Box 1 | | | | S-Box 2 | | | | S-Box 3 | | | | S-Box 4 | | | | Max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | |
| $NL(f_i)$ | 18 | 20 | 22 | 18 | 22 | 20 | 18 | 18 | 18 | 22 | 20 | 18 | 22 | 22 | 22 | 22 | 26 |
| $PC(f_i)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| $deg(f_i)$ | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | |
| $DD_1(f_i)$ | 8 | 8 | 6 | 8 | 4 | 8 | 14 | 8 | 12 | 8 | 12 | 10 | 6 | 6 | 6 | 6 | 16 |
| $NL(S)$ | 14 | | | | 16 | | | | 16 | | | | 16 | | | | 26 |
| $xor(S)$ | 16 | | | | 16 | | | | 16 | | | | 16 | | | | 64 |
| $DBIC(S)$ | 6 | | | | 6 | | | | 6 | | | | 16 | | | | 16 |

| Critères | S-Box 5 | | | | S-Box 6 | | | | S-Box 7 | | | | S-Box 8 | | | | Max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | |
| $NL(f_i)$ | 22 | 20 | 18 | 20 | 20 | 20 | 20 | 20 | 18 | 22 | **14** | 20 | 22 | 20 | 20 | 22 | 26 |
| $PC(f_i)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| $deg(f_i)$ | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | |
| $DD_1(f_i)$ | 6 | 8 | 10 | 6 | 8 | 10 | 8 | 8 | 12 | 8 | 12 | 8 | 10 | 10 | 10 | 8 | 16 |
| $NL(S)$ | 12 | | | | 18 | | | | 14 | | | | 16 | | | | 26 |
| $xor(S)$ | 16 | | | | 16 | | | | 16 | | | | 16 | | | | 64 |
| $DBIC(S)$ | 6 | | | | 6 | | | | 6 | | | | 8 | | | | 16 |

Tab. B.2 – Critères de sécurité BBAC du DES, S-Box 1 à 8

### Analyse de la S-Box de l'AES

Le tableau B.3 donne le résulat de l'application des critères de sécurité boîte noire présentés section B.2.2 à l'AES.

| Critères | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | $f_6$ | $f_7$ | max |
|---|---|---|---|---|---|---|---|---|---|
| $NL(f_i)$ | 112 | 112 | 112 | 112 | 112 | 112 | 112 | 112 | 118 |
| $PC(f_i)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| $deg(f_i)$ | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | |
| $DD_1(f_i)$ | 6 | 8 | 8 | 8 | 8 | 6 | 6 | 6 | 64 |
| $NL(S)$ | 112 | | | | | | | | 118 |
| $xor(S)$ | 4 | | | | | | | | 256 |
| $DBIC(S)$ | 8 | | | | | | | | 64 |

Tab. B.3 – Critères de sécurité BBAC de l'AES

### Analyse des S-Boxes générées aléatoirement

Examinons à présent la qualité cryptographique de S-Boxes 6x4 et 8x8 générées par application de cet algorithme. Nous avons généré $10^7$ S-Boxes 6x4 et $4.10^5$ S-Boxes 8x8. Les tableaux B.4 et B.5 présentent respectivement les caractéristiques des fonctions booléennes et des boîtes de confusion obtenues par application de cet algorithme de génération aléatoire, au regard des critères :

- d'équilibre des fonctions booléennes composantes ;
- de non linéarité ;
- de distribution de la table XOR ;

– et d'indépendance du bit de sortie.

| S-Box | | NL | | |
|---|---|---|---|---|
| in | out | res | bk | thb |
| 6 | 1 | 26 | 26 | 26 |
| 8 | 1 | 116 | 116 | 118 |
| 9 | 1 | 240 | 240 | 244 |
| 10 | 1 | 480 | 492 | 494 |
| 11 | 1 | 960 | 992 | 1000 |
| 12 | 1 | 1946 | 2010 | 2014 |

Tab. B.4 – Non-linéarités des fonctions booléennes générées aléatoirement

| S-Box | | NL | | XOR | | DBIC(2,1) | |
|---|---|---|---|---|---|---|---|
| in | out | res | thb | res | th | res | th |
| 6 | 4 | 22 | 26 | 10 | $[0, 64]$ | 2 | $[0, 16]$ |
| DES | | 18 | 26 | 16 | $[0, 64]$ | 6 | $[0, 16]$ |
| FDE | | 22 | 26 | 12 | $[0, 64]$ | 4 | $[0, 16]$ |
| 8 | 8 | 98 | 118 | 8 | $[0, 256]$ | 8 | $[0, 64]$ |
| AES | | 112 | 118 | 4 | $[0, 256]$ | 8 | $[0, 64]$ |

Tab. B.5 – Caractéristiques des S-Boxes générées aléatoirement

Afin d'établir une comparaison avec l'existant, j'ai indiqué pour ces critères :
- les meilleurs scores obtenus par des méthodes déterministes (bk) ;
- les bornes théoriques (thb[2] pour des composantes équilibrées) ;
- les meilleures valeurs obtenues parmi les boîtes de confusion du DES et de FDE ;
- et les valeurs obtenues pour l'AES 128.

Dans le cas du DES, il est clairement possible d'obtenir des boîtes de confusion de qualité cryptographique comparable ou supérieure. Les valeurs obtenues pour les critères (non-linéarité, valeur xor et critère d'indépendance du bit de sortie) sont même meilleures que celles obtenues par une construction déterministe pour FDE. Observons cependant que le pic dans la distribution des boîtes de confusion (figure B.4) correspond à une non-linéarité certes élevée mais pas optimale ($NL(S) = 20$). Plus du tiers des S-Boxes générées aléatoirement possèdent une valeur xor $XOR(S) \leq 12$ (figure B.6). La distribution obtenue pour le critère DBIC parait à première vue moins encourageante, dans la mesure où le spectre est plus étalé (figure B.8). Environ 30% des S-boxes générées possèdent cependant une distance $DBIC(S) \leq 6$.
Il est possible d'obtenir par application de cet algorithme une haute non-linéarité pour les boîtes de confusion 8x8, mais sensiblement inférieure à celle de la boîte de confusion de l'AES. Le pic dans la distribution des boîtes de confusion (figure B.5) est atteint pour une non-linéarité égale à 94. Près de la moitié des S-Boxes générées aléatoirement possèdent une distance xor $XOR(S) \leq 10$ (figure B.7). La distribution obtenue pour le critère DBIC est centré autour de la valeur $DBIC(S) = 10$ (figure B.9).

### B.2.4 Conclusion

J'ai présenté dans cette annexe une revue des critères de sécurité boîte noire sur lesquels fonder la construction de S-Boxes puis le résultat de mes expérimentations concernant la génération aléatoire de S-Boxes vérifiant ces critères.

[2] Pour les fonctions booléennes équilibrées, rappelons que la borne théorique pour la non-linéarité est donnée par :

$$\text{thb} = \begin{cases} 2^{m-1} - 2^{m/2-1} - 2 & \text{pour } m \text{ pair,} \\ \llbracket 2^{m-1} - 2^{m/2-1} \rrbracket & \text{pour } m \text{ impair,} \end{cases}$$

où $\llbracket x \rrbracket$ désigne le plus grand entier pair inférieur ou égal à $x$.

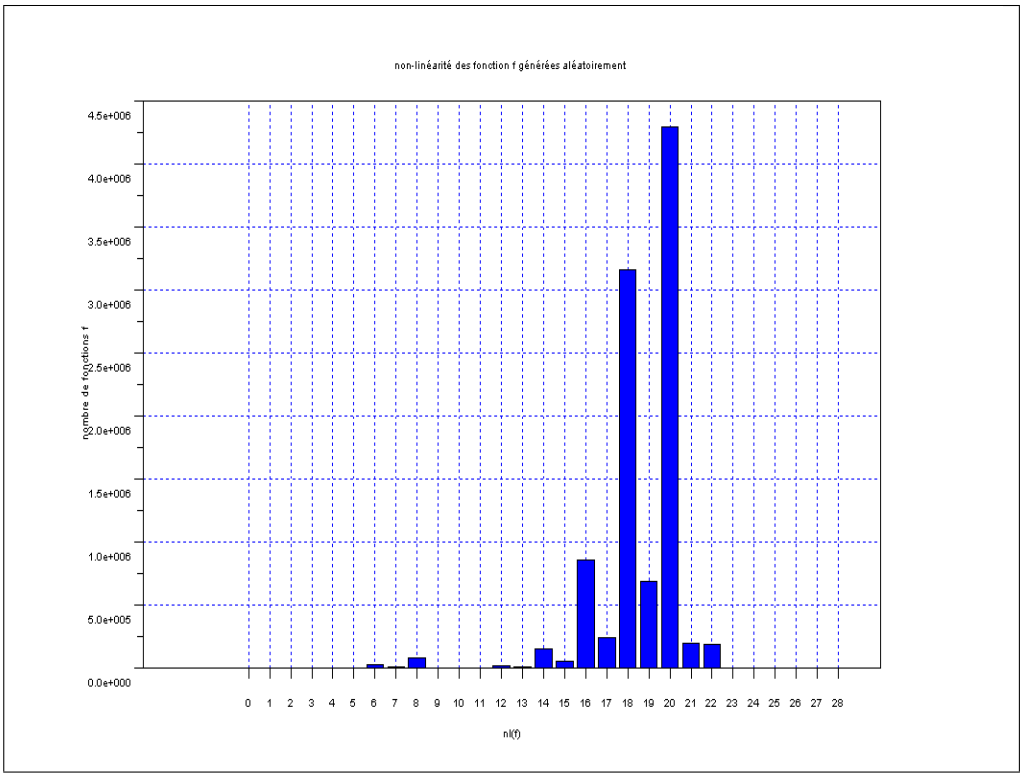



FIG. B.4 – Non Linéarité des S-Boxes $(6, 4)$ générées aléatoirement

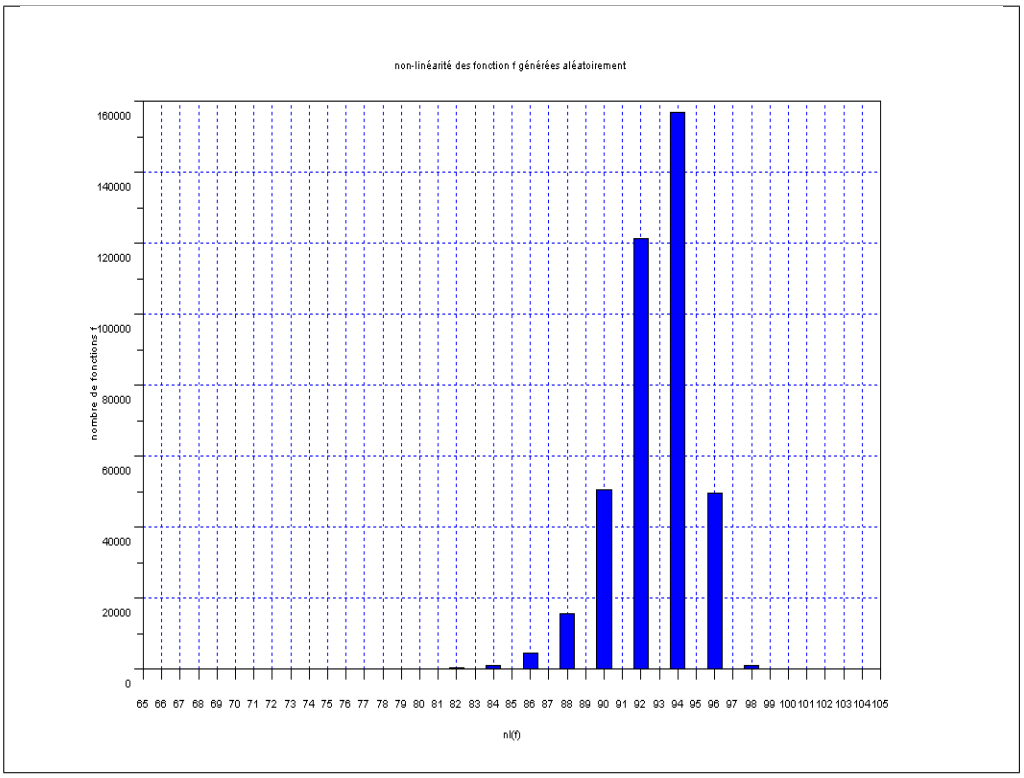



FIG. B.5 – Non Linéarité des S-Boxes $(8, 8)$ générées aléatoirement

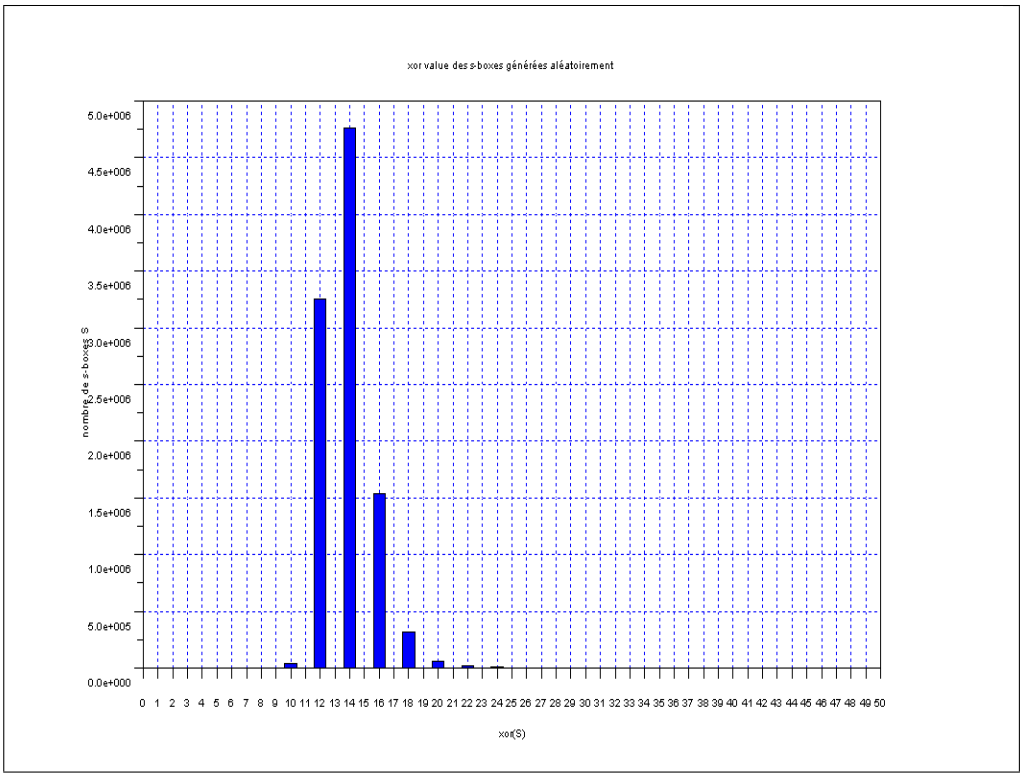



Fig. B.6 – Valeur xor des S-Boxes $(6, 4)$ générées aléatoirement

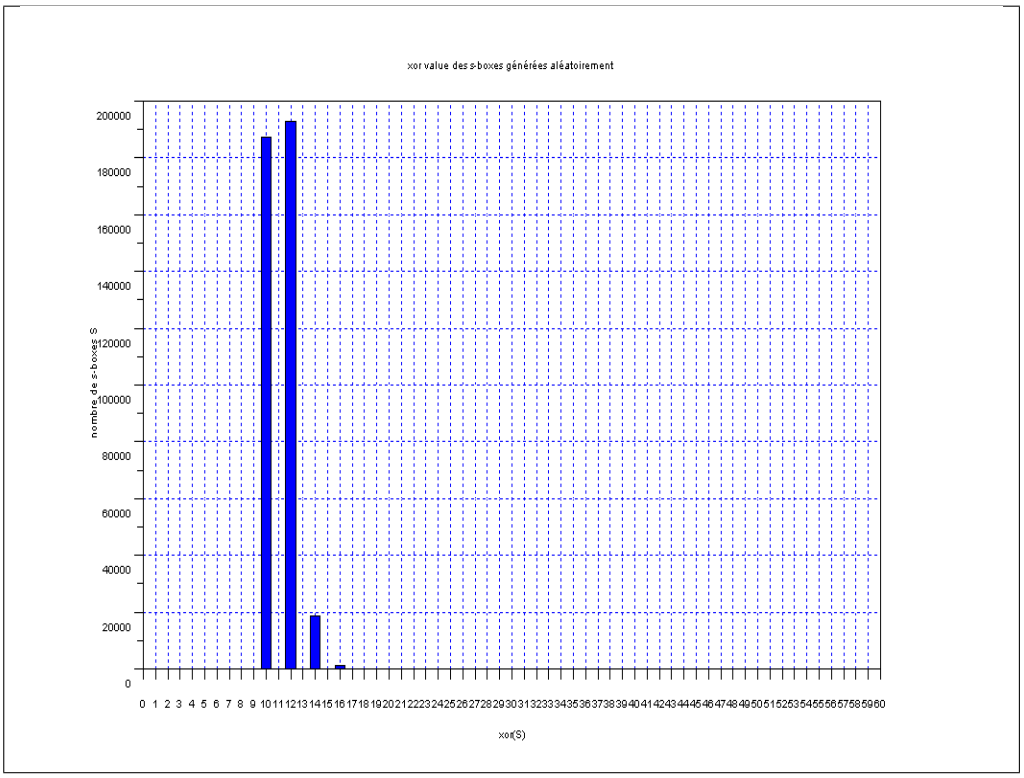



Fig. B.7 – Valeur xor des S-Boxes $(8, 8)$ générées aléatoirement

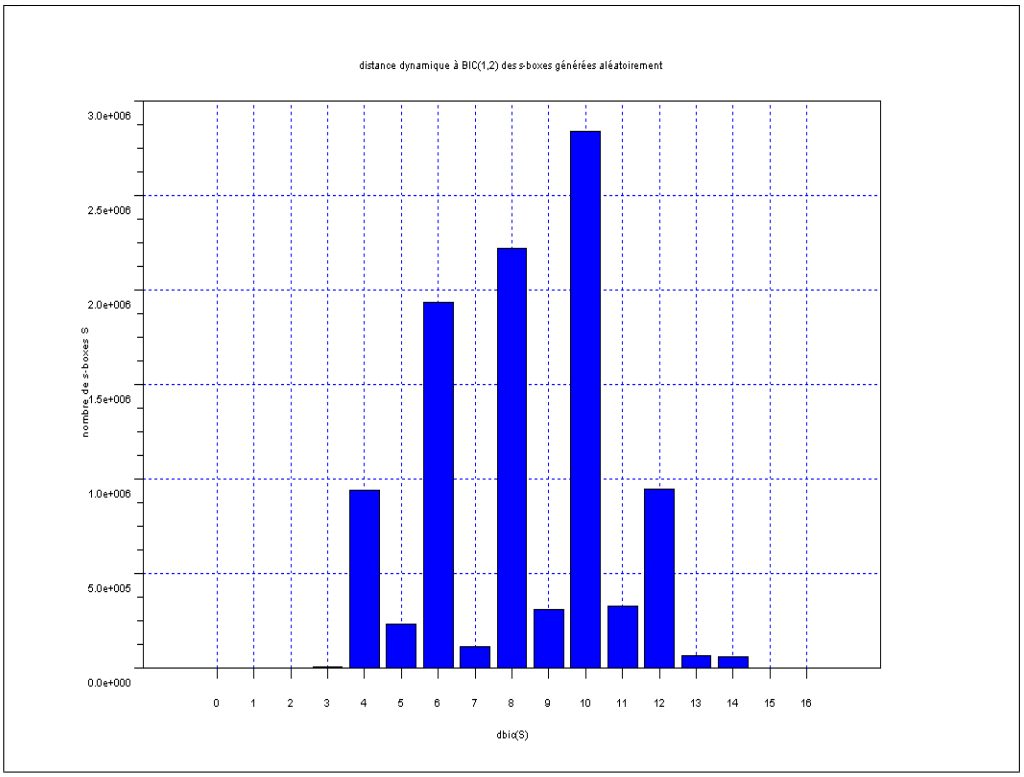



FIG. B.8 – Distance DBIC(1,2) des S-Boxes $(6, 4)$ générées aléatoirement

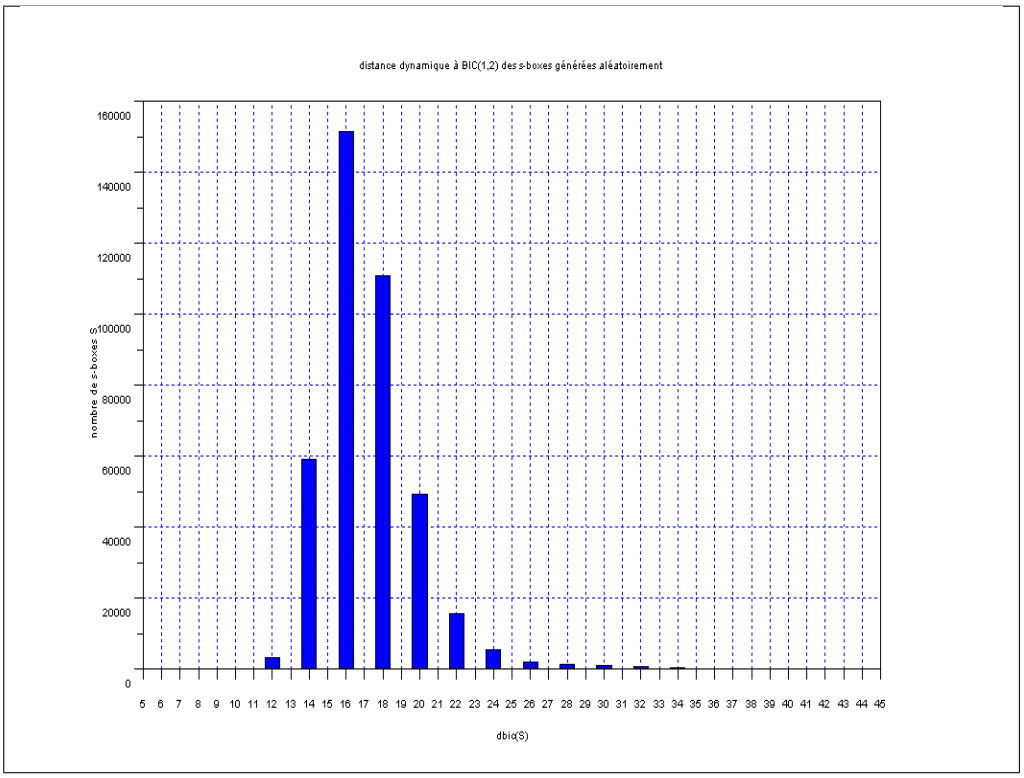



FIG. B.9 – Distance DBIC(1,2) des S-Boxes $(8, 8)$ générées aléatoirement

Il apparaît à l'issue de ces travaux préliminaires qu'il est possible de générer aléatoirement des fonctions booléennes hautement non-linéaires en imposant des contraintes simples sur la forme algébrique normale de ces fonctions booléennes et de construire des S-Boxes de qualité cryptographique comparable (parfois supérieure) à celles utilisées dans les implémentations boîte noire du DES et de FDE au regard des critères de robustesse connus. Bien que moins optimales que celle de l'AES, les S-Boxes $(8, 8)$ générées aléatoirement par application de notre algorithme possèdent néanmoins de bonnes qualités cryptographiques.
Nous pouvons donc envisager la possibilité de diversifier l'implémentation boîte blanche des algorithmes de chiffrement par blocs de type S/P, à travers la randomisation des S-Boxes.

Ces premières expérimentations confirment donc l'idée qu'il est possible assez facilement de générer aléatoirement en grand nombre des boîtes de confusions hautement non linéaires et susceptibles de résister aux cryptanalyses linéaires et différentielles. Concernant les limites de cette approche, remarquons que la satisfaction de ces critères ne constitue pas une preuve de sécurité. Des simulations supplémentaires sont à envisager pour caractériser la sécurité de la fonction de tour obtenue, en terme de probabilité des transitions linéaires et différentielles sur un et plusieurs tours.

J'identifie comme tâche future l'amélioration de l'algorithme de génération aléatoire de S-Boxes en étudiant de manière plus approfondie les propriétés de la représentation sous forme ANF d'une fonction booléenne.

Annexe C

# Implémentation et cryptanalyse du DES et de l'AES boîte blanche

Nous donnons dans cette annexe le code de l'implémentation boîte blanche du DES (section C.1) tel que spécifié dans [CEJvO02a], ainsi qu'une description des deux cryptanalyses proposées dans [WMGP07] (section C.2) et [BGEC04] (section C.3).

## C.1 Implémentation boîte blanche de l'algorithme DES

Nous donnons dans cette section le code (en langage C) d'une implémentation boîte blanche du DES conforme aux spécification [CEJvO02a] représentant l'algorithme de chiffrement ou de déchiffrement.
La fonction `lookup` ci-dessous implémente le parcours du réseau de *lookup tables* `T` représentant chaque étape de l'algorithme DES. Remarquons que les tailles de blocs (`SIZE_IN` et `SIZE_OUT`) peuvent varier selon les étapes (ces tailles prennent les valeur 96 ou 64).

```
void lookup(BYTE a[SIZE_IN], BYTE b[SIZE_OUT], BYTE T[SIZE_OUT/4][SIZE_IN/8][256], int *step){
  BYTE i, j, t, tp;
  for (i=0; i<SIZE_OUT/4; i++){
   b[i*4]=b[i*4+1]=b[i*4+2]=b[i*4+3]=0;
   t=       a[0];
   t=(t<<1)|a[1];
   t=(t<<1)|a[2];
   t=(t<<1)|a[3];
   t=(t<<1)|a[4];
   t=(t<<1)|a[5];
   t=(t<<1)|a[6];
   t=(t<<1)|a[7];
   t=T[i][0][t];
   for (j=1; j<SIZE_IN/8; j++){
    tp=        a[8*j];
    tp=(tp<<1)|a[8*j+1];
    tp=(tp<<1)|a[8*j+2];
    tp=(tp<<1)|a[8*j+3];
    tp=(tp<<1)|a[8*j+4];
    tp=(tp<<1)|a[8*j+5];
    tp=(tp<<1)|a[8*j+6];
    tp=(tp<<1)|a[8*j+7];
    tp=T[i][j][tp];

    t=(tp<<4)|t;
    if (step == -1) {
```

```
     t=TXor[t];
    } else {
     if (j == 1) {
      t=TXOR_0[step][i][t];
     } else {
      t=TXOR_1[step][i][t];
     }
   }
   b[i*4  ]=((t>>3) & 1);
   b[i*4+1]=((t>>2) & 1);
   b[i*4+2]=((t>>1) & 1);
   b[i*4+3]=( t     & 1);
  }
}
```

La fonction `des` ci-dessous implémente les 16 tours de l'algorithme de chiffrement ou de déchiffrement. Nous pouvons remarquer que les seules opérations effectuées sont des parcours de réseaux de *lookup tables* (`TIP`, `T0`, `Ts`, `TF`, etc.).

```
void des(BYTE *in, BYTE *out) {                 // in :   bloc en entrée (64 bits)
                                                // out:   bloc en sortie (64 bits)
  static BYTE i;
  static BYTE block[64], LXR[96], LYR[96];      // blocs d'entrée/sortie (64 ou 96 bits)

  unpack8(in,block);

  lookup(block,LYR,TIP,-1);                     // permutation initiale
  lookup(LYR, LXR, T0, 0);                      // tour zéro
  for (i=0; i<16; i++) {                        // les 16 tours du DES
   lookup(LXR, LXR, Ts__[i], 3*i+1);            // couche non-linéaire
   lookup_bij8(LXR, Ts[i]);                     // substitution
   lookup(LXR, LXR, Ts___[i], 3*i+2);

   if (i==15){                                  // dernier tour ?
    lookup(LXR,LYR,T1s,-1);
    lookup(LYR,block,TF,-1);
   } else {                                     // sinon
    lookup(LXR, LXR, T0T1s[i], 3*i+3);          // couche linéaire
   }
  }

  pack8(out,block);
}
```

## C.2 Cryptanalyse de l'algorithme DES boîte blanche

**Distinguer les T-Boxes non-linéaires** Le coeur de cette étape réside dans l'identification des différences $\Delta v$ qui se traduisent en une différence $f_i^r(\Delta v)$ portant sur un simple (ou certains doubles[1]) bit-flip(s) de $R_{r-1}$. De telles inversions de bits peuvent avoir deux origines comme le montre la Figure 2.1 (partie boîte blanche) : soit de $X_{r-1}$ soit de $\overline{R_{r-1}}$.

**Les inversions de bits affectant $R_{r-1}$** Les différences étant appliquées "à l'aveugle" sur les entrées encodées des $T_i^r$ $(i = 1 \ldots 12)$, elles peuvent tout aussi bien toucher les bits de $L_{r-1}$ ou $Y_r$. La première étape consiste donc à déterminer si $\Delta v$ affecte bien $R_{r-1}$. Les propriétés 1 et 2 permettent de décider cela.

**Propriété 1** *Si $\Delta v$ représente en une simple inversion de bit de $R_{r-1}$ alors la propagation affecte au plus 2 T-Box au tour $r + 2$.*

[1] Il apparaîtra dans la suite que certaines doubles inversions "bien placés" ne peuvent pas être distingués et passent alors pour de simples inversions de bit

**Propriété 2** *Si $\Delta v$ se décode en une inversion de bits de $L_{r-1}$ ou $Y_r$, il y a une forte probabilité pour que plus de 2 T-Box du tour $r+2$ soient affectées.*

Ainsi, lorsque deux T-Box, au plus, du tour $r+2$ sont affectées, la Propriété 1 dit qu'il est possible que cela soit dû à un simple bit-flip de $R_{r-1}$. La Propriété 2 quant à elle, assure qu'il est peu probable qu'une inversion de bits de $L_{r-1}$ ou $X_{r-1}$ affecte moins de 2 T-Box au tour $r+2$. Il est néanmoins possible que cet événement survienne, auquel cas la différence $\Delta v$ représente un *faux positif*. La détection de ces derniers est possible en rejouant la différence $\Delta v$ avec des valeurs différentes pour les $v_j^r$, $j \neq i$. Chaque itération diminue la probabilité de sélectionner un faux positif.

Un second problème intervient dans l'identification des simples bit-flips de $R_{r-1}$. En effet, il est impossible de distinguer un simple bit-flip de $\overline{R_{r-1}}$ d'un double bit-flip de $\overline{R_{r-1}}$ mettant en jeu les deux bits du milieu d'une S-Box du tour $r+2$ (embarquée dans une certaine T-Box $T_k^{r+2}$). En effet, les deux bits du milieu ne touchent, par définition, qu'une seule S-Box. Ainsi un tel *double* bit-flip sur $\overline{R_{r-1}}$ n'affectera l'entrée encodée que d'une seule T-Box $T_k^{r+2}$ au tour $r+2$.

En définitive les propriétés 1 et 2 permettent d'identifier tous les simples bit-flips de $R_{r-1}$, il y en a 32 (16 originaires de $X_{r-1}$ et 16 originaires de $\overline{R_{r-1}}$). Ces propriétés ne permettent pas d'écarter certains doubles bit-flips de $\overline{R_{r-1}}$, il y en a 8 (il s'agit des 8 doubles bit-flips correspondant aux 8 paires $(b_4, b_5)$ de bits du milieu des 8 S-Box).

Sur la base des propriétés 1 et 2, il est donc possible de construire l'ensemble

$$\delta_R(T_i^r) = \{\Delta v = v_i^r \oplus v' \mid v' \in \mathbb{F}_2^8; v' \neq v_i^r; f_i^r(\Delta v) \text{ est une inversion de bits de } R_{r-1}\}$$

avec $|f_i^r(\Delta v)| \in \{1, 2\}$ selon que la différence représente une simple inversion bit ou une des doubles inversions de bits indécelables de $R_{r-1}$. De manière générale, $\delta_R(T_i^r)$ représente l'ensemble des inversions de bits (simples ou doubles) de $R_{r-1}$ qui affectent au plus 2 T-Box au tour $r+2$.

**Identifier les T-Boxes non linéaires** Une fois les différences affectant un ou deux bits de $R_{r-1}$ identifiées, l'attaque sépare $\delta_R(T_i^r)$ en deux sous ensembles disjoints notés $\delta_{\overline{R}}(T_i^r)$ et $\delta_{R\setminus\overline{R}}(T_i^r)$, représentant respectivement les inversions de bits dont l'origine provient d'une différence appliquée à $\overline{R_{r-1}}$ ou à $X_{r-1}$. La construction de ces deux sous ensembles se fait en observant la propagation des différences au tour $r+4$.

**Propriété 3** *Lorsqu'une différence $\Delta v$ représente une inversion de bits de $\overline{R_{r-1}}$, exactement deux différences se propagent au tour $r+2$ : $\Delta m$ et $\Delta n$. L'une d'entre elles (disons $\Delta m$) affecte la T-Box $T_m^{r+2}$ et se propage au tour $r+4$ en affectant plus de 2 T-box. L'autre différence (Deltan) affectera quant à elle, au plus 2 T-Box au tour $r+4$. Cette propagation caractérisera $T_m^{r+2}$ comme étant une T-Box non-linéaire.*

Il est à noter que $\Delta m$ représente une inversion d'un ou deux bits des *middle bits* de la S-Box encapsulée dans $T_m^{r+2}$. Ce détail sera exploité dans l'étape suivante.

**Propriété 4** *Lorsqu'une différence $\Delta v$ représente une inversion de bits de $R_{r-1}\setminus\overline{R_{r-1}}$, exactement deux différences se propagent au tour $r+2$. Les deux T-Box alors affectées sont des T-Boxes non-linéaires qui, à leur tour, affecteront strictement plus de 2 T-Box au tour $r+4$.*

Ces deux propriétés permettent alors de distinguer si une différence $\Delta v$ représente une inversion de bit de $\overline{R_{r-1}}$ (Propriété 3) ou de $R_{r-1}\setminus\overline{R_{r-1}}$ (Propriété 4). Les T-boxes non linéaires qui sont repérées sont celles du tour $r+2$. Ceci conclue la première étape de l'attaque.

**Localiser les S-Box** Les S-Box localisées sont celles du tour $r+3$ et non $r+2$. Leur localisation est rendue possible par l'identification des T-Boxes non linéaires du tour $r+2$ ainsi que des différences représentant une inversion d'un ou deux bits des *middle bits* de la S-Box encapsulée dans $T_m^{r+2}$. Soit $\delta_M(T_m^{r+2}) = \{\Delta m_1, \Delta m_2, \Delta m_3\}$ ces trois différences. De part la construction des S-Box, chaque bit en sortie est inversé au moins une fois suite à l'application successive de ces trois différences (à l'exception de S-Box $S_8$). Ainsi en passant en revue toutes les différences, il est possible d'agir sur tous les bits de la sortie d'une S-Box. Enfin de part la permutation $P$ appliquée en sortie de $S$ et ses propriétés de diffusion, chaque bit de la sortie affecte une S-Box du tour $r+3$ différente. Les

différences de $\delta_M(T_m^{r+2})$ appliquées en entrée de $T_m^{r+2}$ se propagent donc au tour $r+3$ en des différences représentant de simples inversions de bits. Soit donc $\delta_S(T_i^{r+3})$ l'ensemble de ces différences propagée jusqu'à l'entrée de $T_i^{r+3}$.

Soit $T_j^{r+3}$ l'une de ces T-Boxes, l'attaque consiste à deviner la S-Box encapsulée ainsi que la valeur de son entrée $f_j^{r+3}|_{2\ldots7}(v_j^{r+3}) \oplus k_j^{r+3}$ générée au moment de la phase d'initialisation.

Pour cela une implémentation standard du DES est utilisée afin d'établir des comparaisons avec la version obfusquée. L'ensemble $\Omega(T_j^{r+3}) = \{(S_q, w_l) \mid 1 \leq q \leq 8,\ w_l \in \mathbb{F}_2^6\}$ de toutes les paires possible de S-Box et d'entrée initiale va être élagué afin de le réduire à un singleton. Ce couple ainsi obtenu sera composé de la S-Box encapsulée et de son entrée générée lors de la phase initiale.

Afin de réduire l'ensemble $\Omega$, des différences $\Delta v_i \in \delta_S(T_j^{r+3}) \cup \delta_M(T_j^{r+3})$ sont appliquées à $T_j^{r+3}$, puis le nombre de T-Box affectées au tour $r+4$ est comptabilisé par $\delta_i$. Pour tester une paire $(S_q, w_l)$, une différence $\Delta w_i$ représentée par $\Delta v_i$ est additionnée à $w_l$, puis le nombre de S-Box affectées au tour $r+4$ est comptabilisé par $\delta'_i$. Il convient donc de sélectionner des différences $\Delta v_i$ desquelles les différences $\Delta w_i = f_j^{r+3}|_{2\ldots7}(\Delta v_i)$ sont facilement déductibles. Pour une meilleure lisibilité l'indice $i$ intervenant notamment dans l'indexation de $\delta$ et $\delta'$ sera numéroté de 1 à 7 avec la convention suivante : pour les valeurs 1 à 6 l'indice $i$ indique une inversion sur le $i$ème bit de l'entrée de $S_q$, tandis que l'indice $i = 7$ signifie que la différence $\Delta w_i = f_j^{r+3}|_{2\ldots7}(\Delta v_i) = 001100$ est appliquée.

La sélection du couple $(S_q, w_l)$ ou son élimination se fait sur la base d'une comparaison entre $\delta_i$ et $\delta'_i$.

Si $(S_q, w_l)$ est valide, les conditions suivantes doivent être satisfaites :
- $\{\Delta v\} = \delta_M(T_j^{r+3}) \setminus \delta_S(T_j^{r+3})$ est une différence correspondant aux inversions des deux *middle bits*, i.e. $\Delta w = 001100$. Si le couple $(S_q, w_l)$ est correct, alors nécessairement $\delta'_7 = \delta_7$.
- $\{\Delta u, \Delta v\} = \delta_M(T_j^{r+3}) \cap \delta_S(T_j^{r+3})$ représentent les deux simples inversions, notés $\Delta w$ et $\Delta x$, des *middle bits* de $S_q$. Si le couple $(S_q, w_l)$ est correct, alors nécessairement $\{\delta'_3, \delta'_4\} = \{\delta_3, \delta_4\}$.
- De la même manière $\{\delta'_2, \delta'_5\} \in \{\delta_1, \delta_2, \delta_5, \delta_6\}$.

Toute paire $(S_q, w_l)$ ne remplissant pas ces conditions est éliminées de $\Omega$. Si une seule S-Box $S_q$ subsiste parmi éléments de $\Omega$, il est alors possible d'en déduire que $T_j^{r+3}$ encapsule $S_q$. Le fait de découvrir $S_q$ se traduit en une avalanche de nouvelles découvertes en raison des contraintes existant sur l'enchaînement des S-Box du DES tout au long des tours. Ainsi par exemple $S_1$ du tour $r$ ne peut pas, de part les spécifications du DES, affecter les S-Box $S_1$ et $S_7$ du tour $r+1$.

Un certain nombre d'autres contraintes réduisent, par propagation, l'ensemble $\Omega$ à un singleton $(S_q, w_l)$. La composante $w_l$ révèle ainsi l'entrée $f_j^{r+3}|_{2\ldots7}(v_j^{r+3}) \oplus k_j^{r+3}$ de $S_q$, générée au moment de la phase d'initialisation.

**Deviner la clé $K$** La découverte de la clé se fait en initiant un choix sur un bit de cette dernière puis en itérant la procédure suivante :
- l'expansion $E$ duplique certains bits de son entrée. Par exemple le dernier bit (le plus à droite) de l'entrée d'une S-Box $S_i^{r+3}$ $(1 < i < 8)$ est identique au second bit de l'entrée de $S_{i+1}^r$. De même que le premier bit (le plus à gauche) de l'entrée de $S_{i+1}^{r+3}$ est identique à l'avant dernier bit de l'entrée de $S_i^{r+3}$. Soit $b$ l'un des bits dupliqués affectant deux S-Box $S_i^{r+3}$ et $S_{i+1}^{r+3}$ et soit $k$ sa position dans le mot binaire. Les entrées $f_i^{r+3}|_{2\ldots7}(v_i^{r+3}) \oplus k_i^{r+3}$ et $f_{i+1}^{r+3}|_{2\ldots7}(v_{i+1}^{r+3}) \oplus k_{i+1}^{r+3}$ étant connues grâce à l'étape précédente, si le bit en position $k$ de $k_i^{r+3}$ est fixé, il est possible d'en déduire le bit en position $k$ de $k_{i+1}^{r+3}$.
- La valeur d'un bit peut être tracée à travers les tours. Si l'on considère un bit $b$ en position $k$ de $R_{r-1}$ , alors dans le tour $r$, après l'expansion et l'addition de la clé de tour $k_j^r$, $b$ constitue l'entrée d'une S-Box $S_j^r$. Au tour $r+1$, $b$ est additionné avec le bit $P^{-1}(k)$ de la sortie d'une certaine S-Box $S_l$. Enfin au tour $r+2$, après l'expansion et l'addition avec la clé de tour $k_i^{r+2}$, $b$ est en entrée d'une S-Box $S_i$. Ainsi $b$ intervient en entrée de 2 S-Box aux tours $r$ et $r+2$ en mettant également en jeu 2 bits de clé, un bit de $k_j^r$ et un bit de $k_i^{r+2}$. $b$ intervient également dans une addition, au tour $r+1$, avec un bit de sortie de $S_j^r$. Les entrées et sorties des S-Box étant connues, cela fournit une équation avec 3 bits (connus) de S-Box et 2 bits inconnus de deux clés de tour. Le fait de fixer l'un des deux bits de clé permet d'en déduire le second.

Cette méthode effectue un choix (car le système n'est pas resoluble en l'état) puis propage ce choix pour en déduire des valeurs cohérentes avec le choix effectué, i.e. en déduire une instantiation qui satisfait les contraintes portant sur les valeurs inconnues. L'intérêt est que, quelque soit les choix réalisés, la solution finale est une solution correcte grace à la propriété de complémentarité du DES. Cette propriété est défini comme ci-suit :

$$DES_K = \oplus_1 \circ DES_{K\oplus 1} \circ \oplus_1$$

où $\oplus_1$ représente l'addition modulo 2 avec le vecteur "tout à 1".

Ainsi même dans la variante du WB-DES avec des encodages externes, lorsqu'une clé $K$ et des encodages externes $F$ et $G$ sont utilisés, la clé complémentée $K \oplus 1$ est une solution valide avec les encodages externes $F' = \oplus_1 \circ F$ et $G' = G \circ \oplus_1$ définis comme ci-dessous :

$$\begin{aligned} G \circ DES_K \circ F &= G \circ \oplus_1 \circ DES_{K\oplus 1} \circ \oplus_1 \circ F \\ &= G' \circ DES_{K\oplus 1} \circ F' \end{aligned}$$

$\forall l \neq j$, $\Delta v_l^r \leftarrow 0$ (on positionne les $v_l$ à leur valeur d'initialisation).
**Pour** $\Delta v \in \mathbb{F}_2^8 \setminus \{0\}$ **faire**
  calculer 2 fonctions de tour
  **Si** (# de boîtes $T^{r+2}$ affectées $\leq 2$) **Alors**
    $\forall l \neq j$, $v_l = \text{rand}()$
    calculer 2 fonctions de tour
    **Si** (# de boîtes $T^{r+2}$ affectées $\leq 2$) **Alors**
      $S_R(T_j^r) \leftarrow \Delta v$
    **Fin Si**
  **Fin Si**
**Fin Pour**

**Pour** $\Delta v \in S_R(T_j^r)$ **faire**
- calculer 2 fonctions de tour
- $\Delta m$, $\Delta n \leftarrow$ différences propagées au tour $r+2$ de $T_m^{r+2}, T_n^{r+2}$, $m \neq n$
- $\delta m \leftarrow$ # de boîtes $T^{r+4}$ affectées par la propagation de $\Delta m$
- $\delta n \leftarrow$ # de boîtes $T^{r+4}$ affectées par la propagation de $\Delta n$
- **Si** ($\delta m > 2$ et $\delta n = 2$) **Alors**
  - $S_{\overline{R}}(T_j^r) \leftarrow \Delta v$
  - $S_M(T_m^{r+2}) \leftarrow \Delta m$
  - Marquer $T_m^{r+2}$ comme boîte $T$ non linéaire
- **Fin Si**
- **Si** ($\delta m = 2$ et $\delta n > 2$) **Alors**
  - $S_{\overline{R}}(T_j^r) \leftarrow \Delta v$
  - $S_M(T_m^{r+2}) \leftarrow \Delta n$
  - Marquer $T_n^{r+2}$ comme boîte $T$ non linéaire
- **Fin Si**
- **Si** ($\delta m > 2$ et $\delta n > 2$) **Alors**
  - $S_{R\setminus\overline{R}}(T_j^r) \leftarrow \Delta v$
  - $S_M(T_m^{r+2}) \leftarrow \Delta m$
  - Marquer $T_m^{r+2}$ et $T_n^{r+2}$ comme boîtes $T$ non linéaires
- **Fin Si**

**Fin Pour**

## C.3 Cryptanalyse de l'algorithme AES boîte blanche

**Rendre les parasites affines** Soit $Q_j$ l'une des Q-box parasites de sortie de la R-box de la Figure 2.2. La première étape de l'attaque permet de trouver $\tilde{Q}_j$ telle que $Q_j = \tilde{Q}_j \circ A_j$ où $\tilde{Q}_j$ est une bijection non linéaire et $A_j$ est une transformation affine.

Ainsi il suffit de composer $\tilde{Q}_j$ avec les sorties des R-box pour transformer les parasites non linéaires de sortie des R-box des différents tours en parasites linéaires.

De plus $Q_j^r = P_j^{r+1}$ permet également de linéariser les parasites d'entrée des R-box.

**Ôter les parasites affines** La Figure 2.2 décrit le système d'équation suivant :

$$\begin{cases} y_0(x_0,x_1,x_2,x_3) = Q_0(02.T_0'(x_0) \oplus 03.T_1'(x_1) \oplus 01.T_2'(x_2) \oplus 01.T_3'(x_3)) \\ y_1(x_0,x_1,x_2,x_3) = Q_1(01.T_0'(x_0) \oplus 02.T_1'(x_1) \oplus 03.T_2'(x_2) \oplus 01.T_3'(x_3)) \\ y_2(x_0,x_1,x_2,x_3) = Q_2(01.T_0'(x_0) \oplus 01.T_1'(x_1) \oplus 02.T_2'(x_2) \oplus 03.T_3'(x_3)) \\ y_3(x_0,x_1,x_2,x_3) = Q_3(03.T_0'(x_0) \oplus 01.T_1'(x_1) \oplus 01.T_2'(x_2) \oplus 02.T_3'(x_3)) \end{cases}$$

avec $T_i'(x) = T_{i,j+i}^r(P_{i,j+i}^r(x))$ pour un certain $j$ et un certain $r$. À cette étape, bien que les notations soient inchangées, les parasites $Q_i$ et $P_j$ sont des transformations linéaires.

Cette phase de la cryptanalyse permet de découvrir deux choses :

1. les transformations linéaires $Q_0, Q_1, Q_2, Q_3$,
2. une application $\tilde{A}_0$ ainsi que certaines constantes $\delta_0, \delta_1, \delta_2, \delta_3$ et $c_0, c_1, c_2, c_3$ telles que $(S^{-1} \circ \Gamma_{\delta_i} \circ \tilde{A}_0^{-1})(y_0(x, 00, 00, 00) \oplus c_i) = P_i \oplus k_i$, où $S^{-1}$ est l'inverse de la S-box de l'AES.

Ainsi par construction, les sous clés $k_i$ sont découvertes car les $P_i = Q_i^{r-1}$ sont connus.

Reprenons cette cryptanalyse en détail. L'objectif est d'isoler les permutations parasites $Q_{i,j}$, $0 \leq i \leq 3$, $0 \leq j \leq 3$.

MixColumns est une opération linéaire qui satisfait à la relation :
$\forall c_1, c_2, c_3$, $\exists \alpha, \beta_{c_1,c_2,c_3} \in \mathbb{F}_{2^8}$ tel que $\forall x_0 \in \mathbb{F}_{2^8}$,

$$y_0(x_0, c_1, c_2, c_3) = MC(x_0, c_1, c_2, c_3) = \alpha.x_0 \oplus \beta_{c_1,c_2,c_3}.$$

La réalisation d'un tour complet $r$ de WB-AES vérifie donc :

$$y_0(x_0, c_1, c_2, c_3) = Q_{0,j}(\alpha.T_{0,j}(P_{0,j}(x_0)) \oplus \beta_{c_1,c_2,c_3}).$$

Pour $x_3 \in \mathbb{F}_{2^8}$, on a :

$$y_0(x_0, c_1, c_2, x_3) = Q_{0,j}(\alpha.T_{0,j}(P_{0,j}(x_0)) \oplus \beta_{c_1,c_2,x_3}).$$

Soit :

$$Q_{0,j}^{-1}(y_0(x_0, c_1, c_2, x_3)) = \alpha.T_{0,j}(P_{0,j}(x_0)) \oplus \beta_{c_1,c_2,x_3}$$

ou de manière équivalente :

$$\alpha.T_{0,j}(P_{0,j}(x_0)) = Q_{0,j}^{-1}(y_0(x_0, c_1, c_2, x_3)) \oplus \beta_{c_1,c_2,x_3}$$

D'où :

$$y_0(x_0, c_1, c_2, c_3) = Q_{0,j}(Q_{0,j}^{-1}(y_0(x_0, c_1, c_2, x_3)) \oplus \beta_{c_1,c_2,c_3} \oplus \beta_{c_1,c_2,x_3}).$$

Si on note $y_0(x_0, c_1, c_2, x_3) = z_{x_3}(x_0)$ et $y_0(x_0, c_1, c_2, x_3)^{-1} = z_{x_3}^{-1}(y_0(x_0, c_1, c_2, x_3)) = x_0$, on a :

$$y_0(x_0, c_1, c_2, c_3) \circ y_0(x_0, c_1, c_2, x_3)^{-1} = Q_{0,j}(Q_{0,j}^{-1}(x_0) \oplus \beta_{c_1,c_2,c_3} \oplus \beta_{c_1,c_2,x_3}).$$

On est donc capable de construire les *lookup tables* correspondant aux fonctions :

$$f(x_0) = Q_{0,j}(Q_{0,j}^{-1}(x_0) \oplus \beta = (Q_{0,j} \circ \oplus_\beta \circ Q_{0,j}^{-1})(x_0),\ \beta \in \mathbb{F}_{2^8}.$$

Notons $\mathcal{S}$ l'ensemble de ces 256 *lookup tables*. Il existe un isomorphisme $\varphi$ naturel entre $(\mathcal{S}, \circ)$ et $(\mathbb{F}_2^8, \oplus)$, donné par $\varphi(f) = \varphi(Q_{0,j} \circ \oplus_\beta \circ Q_{0,j}^{-1}) = [\beta]$. Même si nous ne connaissons pas $\varphi$, il est possible de sélectionner 8 fonctions $f_i = Q_{0,j} \circ \oplus_{\beta_i} \circ Q_{0,j}^{-1} \in \mathcal{S}$ telles que $(\varphi(f_i) = [\beta_i],\ 1 \leq i \leq 8)$ constitue une base de $\mathbb{F}_2^8$.

Notons $([e_i], 1 \leq i \leq 8)$ la base canonique de $(\mathbb{F}_2^8, \oplus)$ et $L$ l'isomorphisme de bases $L$ défini par : $L([e_i]) = [\beta_i]$, $1 \leq i \leq 8$. Il est possible de déterminer complètement l'isomorphisme $\psi = L^{-1} \circ \varphi$ : soit $f \in \mathcal{S}$, $\exists!(\varepsilon_1, \ldots, \varepsilon_8) \in \mathbb{F}_2^8$ tel que :

$$f = f_8^{\varepsilon_8} \circ \ldots \circ f_1^{\varepsilon_1}.$$

On a :

$$\varphi(f) = \varphi(f_8^{\varepsilon_8} \circ \ldots \circ f_1^{\varepsilon_1}) = \bigoplus_{i=1}^{8} \varepsilon_i.[\beta_i]$$

et

$$\psi(f) = L^{-1}(\varphi(f)) = L^{-1}(\bigoplus_{i=1}^{8} \varepsilon_i.[\beta_i]) = \bigoplus_{i=1}^{8} \varepsilon_i.L^{-1}([\beta_i]) = \bigoplus_{i=1}^{8} \varepsilon_i.[e_i]$$

Considérons le diagramme commutatif suivant (on a noté $Q = Q_{0,j}$) :

$$\begin{tikzcd}
x \arrow[rr, bend left, "\widetilde{Q}"] \arrow[r, "\oplus_{(Q\circ L)^{-1}([00])}"] & & \overline{Q}(x) \arrow[r, "Q"] & f(x) \\
\widetilde{Q}^{-1}(x) \arrow[rrr, bend right, "\widetilde{Q}^{-1}"'] & (Q\circ L)^{-1}(x) \arrow[l, "\oplus_{(Q\circ L)^{-1}([00])}"'] \arrow[u, "\oplus_{\psi(f)}"] & Q^{-1}(x) \arrow[l, "L^{-1}"] \arrow[u, "\oplus_{\varphi(f)}"] & x \arrow[l, "Q^{-1}"] \arrow[u, "Q\circ\oplus_{\varphi(f)}\circ Q^{-1}"'] \\
x \arrow[u, "\oplus_{\psi(f)}"]
\end{tikzcd}$$

On a posé :

$$\overline{Q}(x) = (L \circ \oplus_{(Q\circ L)^{-1}([00])})(x) = L(x \oplus (Q \circ L)^{-1}([00])) = L(x) \oplus Q^{-1}([00])$$

et :

$$\widetilde{Q} = Q \circ \overline{Q}.$$

D'après ce diagramme commutatif, on a : $f = \widetilde{Q} \circ \oplus_{\psi(f)} \circ \widetilde{Q}^{-1}$. Donc $f([00]) = \widetilde{Q}(\widetilde{Q}^{-1}([00]) \oplus \psi(f))$. Or : $\overline{Q}([00]) = L([00]) \oplus Q^{-1}([00]) = Q^{-1}([00])$ donc $(Q \circ \overline{Q})([00]) = [00]$ et $\widetilde{Q}^{-1}([00]) = (Q \circ \overline{Q})^{-1}([00]) = [00]$. D'où :

$$f([00]) = \widetilde{Q}(\psi(f)), f \in \mathcal{S}.$$

Puisque l'on a complètement déterminé $\psi$ et que l'on connaît $f \in \mathcal{S}$, on est donc en mesure de calculer $\widetilde{Q}$.

Nous sommes donc parvenu à reconstruire la partie non linéaire de $Q = Q_{0,j}$. Nous sommes donc en mesure de supprimer la partie non linéaire des permutations parasites en entrée et en sortie de chaque tour de WB-AES :

- on calcule ${}^{r-1}\widetilde{Q}$ et on le compose avec ${}^{r}P = {}^{r-1}Q^{-1}$ en entrée du tour $r$ pour obtenir ${}^{r-1}Q^{-1} \circ {}^{r-1}\widetilde{Q} = {}^{r-1}\overline{Q}$ ;
- on calcule ${}^{r}\widetilde{Q}^{-1}$ et on le compose avec ${}^{r}Q$ en sortie du tour $r$ pour obtenir ${}^{r}\widetilde{Q}^{-1} \circ {}^{r}Q = {}^{r}\overline{Q}^{-1}$.

On a donc :

$$y_i(x_0, x_1, x_2, x_3) = \overline{Q}_i \left( \bigoplus_{i=0}^{3} \alpha_{i,j}.(T_j \circ \overline{P}_j)(x_j) \right) = \overline{Q}_i \left( \bigoplus_{i=0}^{3} (\Lambda_{\alpha_{i,j}} \circ T_j \circ \overline{P}_j)(x_j) \right)$$

où $\Lambda_\alpha(x) = \alpha.x$ représente la mutliplication par $\alpha \in \mathbb{F}_{2^8}$ et $(\alpha_{i,j}, 0 \leq i, j \leq 3)$ représentent les coefficients de Mixcolumns dans un ordre inconnu et où les permutations parasites $\overline{P}$ et $\overline{Q}$ sont affines ($\overline{Q}_i = \oplus_{q_i} \circ A_i$). On a donc en particulier :

$$y_i(x_0, 00, 00, 00) = A_i(\alpha_{i,0}.(T_0 \circ \overline{P}_0)(x_0) \oplus c_i) \oplus q_i$$

(où l'on a posé : $c_i = \alpha_{i,1}.(T_0 \circ \overline{P}_0)(00) \oplus \alpha_{i,2}.(T_0 \circ \overline{P}_0)(00) \oplus \alpha_{i,3}.(T_0 \circ \overline{P}_0)(00)))$. De même, on a :

$$y_j(x_0, 00, 00, 00) = A_j(\alpha_{j,0}.(T_0 \circ \overline{P}_0)(x_0) \oplus c_j) \oplus q_j.$$

ou de manière équivalente :

$$(T_0 \circ \overline{P}_0)(x_0) = \alpha_{j,0}^{-1}(A_j^{-1}(y_j(x_0, 00, 00, 00) \oplus q_j) \oplus c_j).$$

D'où :

$$y_i(x_0, 00, 00, 00) = A_i(\alpha_{i,0}.\alpha_{j,0}^{-1}(A_j^{-1}(y_j(x_0, 00, 00, 00) \oplus q_j) \oplus c_j) \oplus c_i) \oplus q_i.$$

En posant $L = A_i \alpha_{i,0}.\alpha_{j,0}^{-1} A_j^{-1} = A_i \circ \Lambda_{\alpha_{i,0}/\alpha_{j,0}} \circ A_j^{-1}$ et $c = L(q_j \oplus A_j(c_j)) \oplus A_i(c_i) \oplus q_i$, on a donc :

$$y_i(x_0, 00, 00, 00) = L(y_j(x_0, 00, 00, 00)) \oplus c.$$

En outre, cette décomposition est unique.

Il est possible de retrouver $L : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ et $c \in \mathbb{F}_2^8$ en résolvant un système à $8 \times 8 + 8 = 72$ inconnues. En considérant les $2^8$ valeurs possibles pour $x_0$ il est possible d'établir $2^8$ équations de la forme :

$$y_i(x_0, 00, 00, 00) = L(y_j(x_0, 00, 00, 00)) \oplus c.$$

Nous sommes donc en mesure, connaissant $Q_0$, de retrouver $Q_1$, $Q_2$ et $Q_3$.

Pour $(i, j) = (0, 1)$, on a :

$$y_0(x_0, 00, 00, 00) = L_0(y_1(x_0, 00, 00, 00)) \oplus c_0$$

avec $L_0 = A_0 \circ \Lambda_{\alpha_{0,0}\alpha_{1,0}^{-1}} \circ A_1^{-1}$ et :

$$y_0(00, x_1, 00, 00) = L_1(y_1(00, x_1, 00, 00)) \oplus c_1$$

avec $L_1 = A_0 \circ \Lambda_{\alpha_{0,1}\alpha_{1,1}^{-1}} \circ A_1^{-1}$ Nous sommes donc capables de calculer $L = L_0 \circ L_1^{-1} = A_0 \circ \Lambda_{\alpha_{0,0}\alpha_{1,0}^{-1}} \circ A_1^{-1} \circ A_1 \circ \Lambda_{\alpha_{0,1}\alpha_{1,1}^{-1}} \circ A_0^{-1} = A_0 \circ \Lambda_\beta \circ A_0^{-1}$ avec $\beta = \alpha_{0,0}\alpha_{1,0}^{-1}\alpha_{0,1}\alpha_{1,1}^{-1}$.

Les coefficients $\alpha_{i,j}$ appartenant à l'ensemble des coefficients de MixColumns $\{01, 02, 03\}$, on vérifie que seules 16 valeurs sont possibles pour $\beta \in \mathbb{F}_{2^8}$. Etant établie cette valeur $\beta$, et la matrice $A_0 \circ \Lambda_\beta \circ A_0^{-1}$, on peut calculer la matrice $\widetilde{A}_0 = A_0 \circ \Lambda_\gamma$, où $\gamma \in \mathbb{F}_{2^8}$ est non nul et unique mais reste à déterminer. En d'autre termes, nous sommes parvenu à retrouver la partie linéaire $A_0$ de $\overline{Q}_0 = \oplus_{q_0} \circ A_0$ à $\Lambda_\gamma$ près.

Pour calculer $\gamma$ et la constante $q_0$, nous allons utiliser la propriété suivante : $x \mapsto S^{-1} \circ T_0 \circ \overline{P}_0 = S^{-1} \circ S \circ \oplus_{k_i} \circ \overline{P}_0 = \oplus_{k_i} \circ \overline{P}_0$ (où $S^{-1}$ représente l'opération inverse de SubByte) doit être une application affine. Nous allons utiliser cette contrainte pour retrouver $\gamma$ et $q_0$ :

On a :

$$\begin{aligned} y_0(x_0, 00, 00, 00) &= \overline{Q}_0(\alpha_{0,0}.T_0 \circ \overline{P}_0(x_0) \oplus c_0) = A_0(\alpha_{0,0}.T_0 \circ \overline{P}_0(x_0) \oplus c_0) \oplus q_0 \\ &= \widetilde{A}_0 \circ \Lambda_{1/\gamma}(\alpha_{0,0}.T_0 \circ \overline{P}_0(x_0) \oplus c_0) \oplus q_0. \end{aligned}$$

On en déduit :

$$S^{-1} \circ T_0 \circ \overline{P}_0(x_0) = S^{-1} \circ \alpha_{0,0}^{-1}.(\Lambda_\gamma \circ \widetilde{A}_0^{-1}(y_0(x_0, 00, 00, 00) \oplus q_0) \oplus c_0).$$

On établit donc l'existence d'uniques couples $((\delta_i, d_i),\ 0 \leq i \leq 3)$ avec $\delta_i \neq 0$ tels que les fonctions :

$$\begin{aligned} x_0 &\mapsto (S^{-1} \circ \Lambda_{\delta_0} \circ \widetilde{A}_0^{-1})(y_0(x_0, 00, 00, 00) \oplus d_0 \\ x_1 &\mapsto (S^{-1} \circ \Lambda_{\delta_1} \circ \widetilde{A}_0^{-1})(y_0(00, x_1, 00, 00) \oplus d_1 \\ x_2 &\mapsto (S^{-1} \circ \Lambda_{\delta_2} \circ \widetilde{A}_0^{-1})(y_0(00, 00, x_2, 00) \oplus d_2 \\ x_3 &\mapsto (S^{-1} \circ \Lambda_{\delta_3} \circ \widetilde{A}_0^{-1})(y_0(00, 00, 00, x_3) \oplus d_3 \end{aligned}$$

soient affines. En outre, ces fonctions sont exactement les fonctions

$$x \mapsto \overline{P}_i(x) \oplus k_i,\ 0 \leq i \leq 3.$$

On va donc tester exhaustivement les $(2^8)^2 = 2^16$ paires $(\delta_i, d_i)$ et vérifier si la fonction est affine. Sa *lookup table* doit donc être évaluée $2^8$ fois puis un système de $2^8$ équations à $8 \times 8 + 8 = 72$ inconnues (qui peut être pré-calculé) doit être résolu.

Lors de cette étape, nous retrouvons donc également la partie affine ${}^r\overline{P}$ de ${}^rP$ à l'addition près par la clé de tour ${}^rk$ : $(\oplus_{{}^rk_{i,j}} \circ {}^r\overline{P}_{i,j})$. En composant $(\oplus_{{}^rk_{i,j}} \circ {}^r\overline{P}_{i,j})$ avec ${}^{r-1}\overline{Q}_{i,j}$, on retrouve la clé de tour :

$$(\oplus_{{}^rk_{i,j}} \circ {}^r\overline{P}_{i,j}) \circ {}^{r-1}\overline{Q}_{i,j} = \oplus_{{}^rk_{i,j}} \circ {}^{r-1}\overline{Q}_{i,j}^{-1} \circ {}^{r-1}\overline{Q}_{i,j} = \oplus_{{}^rk_{i,j}} \circ Id_{2^8} = {}^rk_{i,j}.$$

**Extraire la clé d'un tour** La découverte des sous clés $k_i$ n'implique pas la connaissance immédiate de la clé utilisée pour le tour car les $k_i$ ne sont pas ordonnés. En utilisant les flux de données il possible d'appareiller les *lookup tables* de deux tours successifs deux à deux. Ces contraintes ajoutées à celles induites par la dérivation des sous clés par l'algorithme de cadencement standard (FIPS-197 [NIS01]) permettent d'ordonner ces sous clés et ainsi de retrouver la clé du tour.
Une fois une clé de tour révélée, la clé principale de l'AES l'est également en remontant l'algorithme de cadencement.

Annexe D

# Modélisation des transformations de programme par interprétation abstraite

## D.1 Rappels théorique

**Interprétation abstraite** L'interprétation abstraite est une théorie de l'approximation discrète des sémantiques d'un langage de programmation. La sémantique d'un programme est un modèle de calcul décrivant les exécutions effectives de ce programme dans tous les environnements possibles. La sémantique d'un langage est donnée pour chaque programme syntaxiquement correct de ce langage.

**Ensemble partiellement ordonné** Les sémantiques concrètes d'un programme $P$ sont calculées sur un ensemble appelé domaine concret, représenté par un ensemble partiellement ordonné $\mathrm{po}\langle E, \sqsubseteq\rangle$.

**Définition D.1.1** *(Poset ou Ensemble partiellement ordonné). Un ensemble $E$ muni d'une relation d'ordre $\sqsubseteq$ est appelé ensemble partiellement ordonné (poset, partial ordered set). On le note $\mathrm{po}\langle E, \sqsubseteq\rangle$.*

Soit $E$ un ensemble. Notons $\mathcal{P}(E)$ l'ensemble des parties de $E$. $\mathrm{po}\langle \mathcal{P}(E), \subseteq\rangle$ est un exemple d'ensemble partiellement ordonné. La relation d'ordre partielle sur $E$ représente la précision d'une description : $x \sqsubseteq y$ signifie que $x$ est une description plus précise, plus concrète, que $y$. Le domaine concret d'un programme avec des variables entières est simplement donné par l'ensemble partiellement ordonné $\mathrm{po}\langle \mathcal{P}(\mathbb{Z}), \subseteq\rangle$.

Les approximations des sémantiques concrètes d'un programme $P$, appelées sémantiques abstraites, sont calculées sur un ensemble partiellement ordonné $\mathrm{po}\langle F, \sqsubseteq\rangle$ appelé domaine abstrait. La relation d'ordre partielle sur $F$ représente la précision d'une description : $x \sqsubseteq y$ signifie que $x$ est une description plus précise, une meilleure approximation, que $y$.

**Définition D.1.2** *(Chaîne ou Ensemble totalement ordonné). Soit $\mathrm{po}\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Si $\sqsubseteq$ est une relation d'ordre totale sur $E$, $\mathrm{po}\langle E, \sqsubseteq\rangle$ est un ensemble totalement ordonné : $\forall x, y \in E$, $x \sqsubseteq y$ ou $y \sqsubseteq x$.*

$\mathrm{po}\langle \mathbb{N}, \leq\rangle$ est un exemple d'ensemble totalement ordonné.

**Définition D.1.3** *(Dual d'un ensemble partiellement ordonné). Soit $\mathrm{po}\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Son ensemble dual est l'ensemble noté $\mathrm{po}\langle E, \sqsupseteq\rangle$.*

**Définition D.1.4** *(Idéal d'un ensemble partiellement ordonné). Soit $\mathrm{po}\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Un ensemble $I \subseteq E$ est un idéal de $E$ si $\forall x \in I$, $\forall y \in E$, $y \sqsubseteq x \Rightarrow y \in I$.*

**Définition D.1.5** *(Filtre d'un ensemble partiellement ordonné). Soit $\mathrm{po}\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Un ensemble $F \subseteq E$ est un filtre de $E$ si $\forall x \in F$, $\forall y \in E$, $y \sqsupseteq x \Rightarrow y \in F$. En d'autre terme, un filtre est le dual d'un idéal.*

**Définition D.1.6** *(Idéal clôture d'un ensemble partiellement ordonné). Soit po$\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Soit $G \subseteq E$. On note $\downarrow G$ l'idéal clôture de $E$, défini par : $\downarrow G = \{y \in E \mid \exists x \in G : y \sqsubseteq x\}$.*

**Définition D.1.7** *(Filtre clôture d'un ensemble partiellement ordonné). Soit po$\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Soit $G \subseteq E$. On note $\uparrow G$ le filtre clôture de $E$, défini par : $\uparrow G = \{y \in E \mid \exists x \in G : y \sqsupseteq x\}$.*

**Définition D.1.8** *Soit po$\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Soit $F \subseteq E$. On note $\bigvee F = \inf\{y \mid \forall x \in F, x \sqsubseteq y\}$. Si $\bigvee F \in E$, alors on note $\top$ ce maximum.*

L'antisymétrie de la relation d'ordre partielle $\sqsubseteq$ sur $E$ assure l'unicité de $\top$.

**Définition D.1.9** *Soit po$\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Soit $F \subseteq E$. On note $\bigwedge F = \sup\{y \mid \forall x \in F, x \sqsupseteq y\}$. Si $\bigwedge F \in E$, alors on note $\perp$ ce minimum.*

L'antisymétrie de la relation d'ordre partielle $\sqsupseteq$ sur $E$ assure l'unicité de $\perp$.

**Définition D.1.10** *(Ensemble direct). Un ensemble partiellement ordonné po$\langle E, \sqsubseteq\rangle$ est dit direct si tout sous-ensemble fini de $E$ admet un maximum.*

**Définition D.1.11** *(Ordre partiel complet). Un ensemble partiellement ordonné po$\langle E, \sqsubseteq\rangle$ est dit complet si $\perp \in E$ et pour tout ensemble direct $D \subseteq E$, on a $\bigvee D \in E$.*

Remarquons que tout ensemble partiellement ordonné fini est complet. À titre d'exemple d'ensemble partiellement ordonné incomplet, on a le sous-ensemble $[3, 4] \subseteq \mathbb{Q}$ des nombre rationnels. po$\langle[3, 4], \leq\rangle$ n'est pas complet car la suite strictement croissante d'éléments de po$\langle[3, 4], \leq\rangle$ : $x_0 = 3$, $x_1 = 3.1$, $x_2 = 3.14$, ... (suite d'Archimède définie par $x_0 = 3$, $x_{n+1} = 6.2^n\sqrt{2 - 2\sqrt{1 - x_n^2}}$) converge vers $\pi$ et que $\pi \notin \mathbb{Q}$. Par conséquent, $\bigvee\{x_n \mid n \geq 0\} = max\{x_n \mid n \geq 0\} = \pi \notin$ po$\langle[3, 4], \leq\rangle$.

**Treillis**

**Définition D.1.12** *(Treillis). Un ensemble partiellement ordonné po$\langle E, \sqsubseteq\rangle$ est un treillis si $\forall x, y \in E$, $x \vee y$ et $x \wedge y$ sont dans $E$. Un treillis est complet si $\forall F \subseteq E$, on a $\bigvee F \in E$ et $\bigwedge F \in E$.*
*Un treillis complet est noté $\langle E, \sqsubseteq_E, \vee_E, \wedge_E, \top_E, \perp_E\rangle$, où $\top_E = \bigvee E$ et $\perp_E = \bigwedge E$.*

po$\langle\mathcal{P}(E), \subseteq\rangle$ est un exemple de treillis complet.

**Définition D.1.13** *(Complément). Soit po$\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné possédant un élément minimum $\perp$ et un élément maximum $\top$. Soit $x \in E$. On appelle complément de $x$ un élément $y \in E$ tel que $x \vee y = \top$ et $x \wedge y = \perp$.*

**Définition D.1.14** *(Ascending Chain Condition). Un ensemble partiellement ordonné po$\langle E, \sqsubseteq\rangle$ satisfait la condition ACC si toute suite strictement croissante d'éléments de $E$ est finie.*

**Définition D.1.15** *(Descending Chain Condition). Un ensemble partiellement ordonné po$\langle E, \sqsubseteq\rangle$ satisfait la condition DCC si toute suite strictement décroissante d'éléments de $E$ est finie.*

**Définition D.1.16** *(Treillis distributif). Un ensemble partiellement ordonné po$\langle E, \sqsubseteq\rangle$ est dit distributif si $\forall x, y, z \in E$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$.*

**Définition D.1.17** *Un treillis $\langle E, \sqsubseteq\rangle$ est dit complémenté si chaque élément a un complément.*

**Définition D.1.18** *(Algèbre booléenne). Un treillis complémenté et distributif est appelé algèbre booléenne. Une algèbre booléenne complète est un treillis complet, complémenté et distributif.*

**Définition D.1.19** *(Fonction monotone). Soient po$\langle E, \sqsubseteq_E\rangle$ et po$\langle F, \sqsubseteq_F\rangle$ deux ensembles partiellement ordonnés ; et une fonction $f : E \to F$. $f$ est dite monotone si $\forall x, y \in E$, $x \sqsubseteq_E y \Rightarrow f(x) \sqsubseteq_E f(y)$.*

**Définition D.1.20** *(Isomorphisme). Soient* $po\langle E, \sqsubseteq_E\rangle$ *et* $po\langle F, \sqsubseteq_F\rangle$ *deux ensembles partiellement ordonnés ; et une fonction surjective* $f : E \to F$. $f$ *est un isomorphisme si* $\forall x, y \in E$, $x \sqsubseteq_E y \Leftrightarrow f(x) \sqsubseteq_E f(y)$.

**Définition D.1.21** *(Fonction continue). Soient* $po\langle E, \sqsubseteq_E\rangle$ *et* $po\langle F, \sqsubseteq_F\rangle$ *deux ensembles partiellement ordonnés complets ; et une fonction* $f : E \to F$. $f$ *est continue si elle est monotone et qu'elle préserve les limites des ensembles directs : quelque soit l'ensemble direct* $D \subseteq E$, *on a :* $f(\bigvee_E D) = \bigvee_F f(D)$.

**Définition D.1.22** *(Fonction co-continue). Soient* $po\langle E, \sqsubseteq_E\rangle$ *et* $po\langle F, \sqsubseteq_F\rangle$ *deux ensembles partiellement ordonnés complets ; et une fonction* $f : E \to F$. $f$ *est co-continue si elles est monotone et qu'elle préserve les limites des ensembles directs : quelque soit l'ensemble direct* $D \subseteq E$, *on a :* $f(\bigwedge_E D) = \bigwedge_F f(D)$.

**Définition D.1.23** *(Fonction additive). Soient* $po\langle E, \sqsubseteq_E\rangle$ *et* $po\langle F, \sqsubseteq_F\rangle$ *deux ensembles partiellement ordonnés complets ; et une fonction* $f : E \to F$. $f$ *est additive si* $\forall G \subseteq E$, *on a :* $f(\bigvee_E G) = \bigvee_F f(G)$.

Une fonction additive est en particulier continue.

**Définition D.1.24** *(Fonction co-additive). Soient* $po\langle E, \sqsubseteq_E\rangle$ *et* $po\langle F, \sqsubseteq_F\rangle$ *deux ensembles partiellement ordonnés complets ; et une fonction* $f : E \to F$. $f$ *est co-additive si* $\forall G \subseteq E$, *on a :* $f(\bigwedge_E G) = \bigwedge_F f(G)$.

Une fonction co-additive est en particulier co-continue.

**Définition D.1.25** *(Point fixe). Soit* $po\langle E, \sqsubseteq\rangle$ *un ensemble partiellement ordonné et une fontion* $f : E \to E$. *Un élément* $x \in E$ *est un point fixe de* $f$ *si* $f(x) = x$. *On note* $Fix(f) = \{x \in E \mid f(x) = x\}$ *l'ensemble des points fixes de* $f$.

S'il existe, on définit le plus petit point fixe $lfp^{\sqsubseteq}f$ de $f$ comme étant l'unique élément $x \in Fix(f)$ tel que $\forall y \in Fix(f)$, $x \sqsubseteq y$.
On définit de même de manière duale le plus grand point fixe $gfp^{\sqsubseteq}f$ de $f$.

### Théorème du point fixe

**Théorème D.1.1** *(Théorème du point fixe de Kleene-Knaster-Tarski-Cousot [CC01]). Soit* $\langle E, \sqsubseteq\rangle$ *un treillis complet.*

1. *Si* $f : E \to E$ *une fonction monotone, alors* $\langle Fix(f), \sqsubseteq\rangle$ *est un treillis complet. En outre, il existe un unique plus petit point fixe* $lfp^{\sqsubseteq}f$ *de* $f$ *dans* $E$.
2. *Si* $f$ *est continue, le plus petit point fixe de* $f$ *peut être calculé récursivement en partant de l'élément* $\perp$ *:*
$$lfp^{\sqsubseteq}(f) = \bigvee_{n \geq 0} f^n(\perp).$$
3. *Si* $f$ *est continue et que le treillis* $E$ *vérifie la condition ACC, le plus petit point fixe de* $f$ *peut être calculé récursivement et en un nombre fini d'itérations en partant de l'élément* $\perp$ *:*
$$\exists m \text{ tel que } lfp^{\sqsubseteq}(f) = \bigvee_{n=0}^{m} f^n(\perp).$$
4. *De manière duale, le plus grand point fixe d'une fonction co-continue sur un treillis complet* $E$ *peut être calculé en partant de l'élément* $\top$ *:*
$$gfp^{\sqsubseteq}(f) = \bigwedge_{n \geq 0} f^n(\top).$$
5. *Si* $f$ *est co-continue et que le treillis* $E$ *vérifie la condition DCC, le plus grand point fixe de* $f$ *peut être calculé récursivement et en un nombre fini d'itérations en partant de l'élément* $\top$ *:*
$$\exists m \text{ tel que } gfp^{\sqsubseteq}(f) = \bigwedge_{n=0}^{m} f^n(\top).$$

Si $\langle E, \sqsubseteq, \vee, \wedge, \top, \bot \rangle$ est un treillis complet, alors $\langle E^k, \sqsubseteq_k, \vee_k, \wedge_k, \top_k, \bot_k \rangle$ défini par :
- $(x_1, \ldots, x_k) \sqsubseteq_k (y_1, \ldots, y_k) \Leftrightarrow \forall i = 1, \ldots, k,\ x_k \sqsubseteq y_k$
- $(x_1, \ldots, x_k) \vee_k (y_1, \ldots, y_k) = (x_1 \vee y_1, \ldots, x_k \vee y_k)$
- $(x_1, \ldots, x_k) \wedge_k (y_1, \ldots, y_k) = (x_1 \wedge y_1, \ldots, x_k \wedge y_k)$
- $\top_k = \top^k$ et $\bot_k = \bot^k$

forme un treillis complet.

Si l'on considère une équation de la forme $x = F(x)$ avec $k$ variables :

$$\begin{cases} x_1 & = & f_1(x_1, \ldots, x_k) \\ & \vdots & \\ x_k & = & f_k(x_1, \ldots, x_k) \end{cases}$$

et que l'on suppose les fonctions $f_i : E^k \to E$ monotones, alors la fonction $F : E^k \to E^k$ est monotone.

On est alors assuré de l'existence d'un unique point fixe. En outre, $Fix(F)$ forme un treillis complet pour la relation d'ordre partiel $\sqsubseteq_k$ (théorème D.1.1, monotonie).

Sous ces mêmes hypothèses, on peut également définir le plus petit point fixe de $F$ :

$$lfp(F) = \bigvee_k \{x \in E^k \mid F(x) \sqsubseteq_k x\}$$

et le plus grand point fixe de $F$ :

$$gfp(F) = \bigwedge_k \{x \in E^k \mid x \sqsubseteq_k F(x)\}.$$

Ce résultat nous permet donc de discuter de l'existence d'une solution au système d'équations, mais ne nous fournit pas un algorithme permettant de construire les points fixes. Des hypothèses supplémentaires sont requises pour fournir des méthodes constructives.

Si les fonctions $f_i : E^k \to E$ sont supposées continues (elles sont donc en particulier monotones), alors la fonction $F : E^k \to E^k$ est également continue. On peut alors construire récursivement le plus petit point fixe de $F$ en partant de l'élément $\bot_k$ (théorème D.1.1, continuité) :

$$lfp(F) = \bigvee_{n \geq 0} F^n(\bot_k)$$

et le plus grand point fixe de $F$ en partant de l'élément $\top_k$ :

$$gfp(f) = \bigwedge_{n \geq 0} F^n(\top_k).$$

Puisque $\bot_k$ est le plus petit élément du treillis complet $E^k$, on a : $\bot_k \sqsubseteq_k F(\bot_k)$. Puisque $F$ est monotone, on a : $F^n(\bot_k) \sqsubseteq_k F^{n+1}(\bot_k)$. Les approximations successives $\bot_k = F^0(\bot_k), F^1(\bot_k), \ldots, F^n(\bot_k), \ldots$ forment donc une suite croissante $\bot_k = F^0(\bot_k) \sqsubseteq_k F^1(\bot_k) \sqsubseteq_k \ldots \sqsubseteq_k F^n(\bot_k) \sqsubseteq_k \ldots$ d'où :

$$\bigvee_{n=0}^{m} F^n(\bot_k) = F^m(\bot_k)$$

et :

$$\bigvee_{n=0}^{\infty} F^n(\bot_k) = \lim_{m \to \infty} F^m(\bot_k).$$

Une infinité d'approximations successives peuvent donc être nécessaires pour atteindre la limite lorsque la séquence des approximations croît strictement indéfiniment. Dans le cas contraire, les $p$ premiers termes de la séquence sont strictement croissants et les termes suivants sont égaux à $F^p(\bot_k)$. On est dans cette dernière situation lorsque le treillis $E$ satisfait la condition ACC.

### Correspondance de Galois

**Définition D.1.26** *(Correspondance de Galois). Deux ensembles partiellement ordonnés po$\langle E, \sqsubseteq_E\rangle$ et po$\langle F, \sqsubseteq_F\rangle$ et deux fonctions monotones $f : E \to F$ et $g : F \to E$ forment une correspondance de Galois $(E, f, g, F)$ si :*
- $\forall x \in E,\ x \sqsubseteq_E g(f(x))$ *et*
- $\forall y \in F,\ f(g(y)) \sqsubseteq_F y$.

**Définition D.1.27** *(Insertion de Galois). Une insertion de Galois $(E, f, g, F)$ est une correspondance de Galois $(E, f, g, F)$ telle que : $\forall y \in F,\ f(g(y)) = y$.*

Le lien entre les domaines concret po$\langle E, \sqsubseteq_E\rangle$ et abstrait po$\langle F, \sqsubseteq_F\rangle$ est modélisé par une correspondance de Galois $(E, f, g, F)$. La fonction monotone $f : E \to F$ est appelée fonction d'abstraction et la fonction $g : F \to E$ est appelée fonction de concrétisation. Etant donnée une correspondance de Galois $(E, f, g, F)$, on dit que $F$ est une interprétation abstraite de $E$ et que $E$ est une concrétisation de $F$. $\forall x \in E$, $f(x)$ est la représentation abstraite de $x$ et $\forall y \in F$, $g(y)$ est la représentation concrète de $y$.

### Opérateur de clôture

**Définition D.1.28** *(Opérateur de clôture). Soit po$\langle E, \sqsubseteq\rangle$ un ensemble partiellement ordonné. Une clôture est un opérateur $\rho : E \to E$*
- *extensif :* $\forall x \in E,\ x \sqsubseteq \rho(x)$ *;*
- *monotone :* $\forall x, y \in E,\ x \sqsubseteq y \Rightarrow \rho(x) \sqsubseteq \rho(y)$ *;*
- *idempotent :* $\forall x \in E,\ \rho(\rho(x)) = \rho(x)$.

On note $uco(E)$ l'ensemble des opérateurs de clôture de po$\langle E, \sqsubseteq\rangle$. Si $\langle E, \sqsubseteq\rangle$ est un treillis complet, alors $\forall \rho \in uco(E)$, on a : $\rho(y) = \bigwedge\{x \in E \,|\, x = \rho(x) \text{ et } y \sqsubseteq x\}$. Ce qui signifie que l'image d'un élément $y$ par $\rho$ est le plus petit point fixe de $\rho$ plus grand que $y$. En outre, $\rho$ est uniquement déterminé par son image $\rho(E)$ qui est l'ensemble des points fixes de $\rho$, $\rho(E) = Fix(\rho)$.

Etant donné un treillis complet $\langle E, \sqsubseteq_E, \vee_E, \wedge_E, \top_E, \bot_E\rangle$, où $\top_E = \bigvee E$ et un opérateur de clôture $\rho \in uco(E)$, l'image $\rho(E)$ est un treillis complet $\langle \rho(E), \sqsubseteq_{\rho(E)}, \vee_{\rho(E)}, \wedge_{\rho(E)}, \top_{\rho(E)}, \bot_{\rho(E)}\rangle$ avec :
- $\forall F \subseteq \rho(E), \vee_{\rho(E)}(F) = \rho(\vee_E F)$ ;
- $\wedge_{\rho(E)} = \wedge_E$ ;
- $\top_{\rho(E)} = \top_E$ ;
- $\bot_{\rho(E)} = \rho(\bot_E)$.

On a également le résultat suivant : soit $\langle E, \sqsubseteq_E, \vee_E, \wedge_E, \top_E, \bot_E\rangle$ un treillis complet. Alors $\langle uco(E), \sqsubseteq, \sqcup, \sqcap, \top_{uco(E)}, \bot_{uco(E)}\rangle$ est un treilli complet avec :
- $\forall \rho, \mu \in uco(E),\ \rho \sqsubseteq \mu \Leftrightarrow \forall x \in E, \rho(x) \sqsubseteq_E \mu(x) \Leftrightarrow \mu(E) \subseteq \rho(E)$ ;
- $\forall \{\rho_i\}_{i\in I} \subseteq uco(E)$, $(\sqcap_{i\in I}\rho_i)(x) = \wedge_{i\in I}\rho_i(x)$. Le domaine abstrait $\sqcap_{i\in I}\rho_i(E)$ est plus concret que tous les domaines $\{\rho_i(E)\}_{i\in I}$. Le produit réduit $\sqcap$ sur $uco(E)$ est utilisé pour combiner des domaines abstraits connus pour concevoir de nouvelles abstractions plus précises.
- $\forall \{\rho_i\}_{i\in I} \subseteq uco(E)$, $\sqcup_{i\in I}\rho_i(x) = x \Leftrightarrow \forall i \in I,\ \rho_i(x) = x$. Le domaine abstrait $\sqcup_{i\in I}\rho_i(E)$ est la plus petite abstraction commune à tous les domaines $\{\rho_i(E)\}_{i\in I}$. L'opérateur $\sqcup$ fournit le domaine le plus concret dans $uco(E)$ qui est une abstraction de tous les domaines $\{\rho_i(E)\}_{i\in I}$ : $\sqcup_{i\in I}\rho_i(E) = \bigcap_{i\in I}\rho_i(E)$.
- $\top_{uco(E)} = \lambda x.\top_E$ ;
- $\bot_{uco(E)} = \lambda x.x$.

**Lien entre opérateur de clôture et correspondance de Galois** Les notions de correspondance de Galois et d'opérateur de clôture sont équivalentes. En effet, étant donnée une correspondance de Galois $(E, f, g, F)$, on peut démontrer que $g \circ f$ est une clôture sur $E$. Inversement, étant donné un ensemble partiellement ordonné po$\langle E, \sqsubseteq\rangle$, et une clôture $\rho$ sur $E$, alors $(E, \rho, \lambda x.x, \rho(E))$ définit une insertion de Galois.

Les propriétés d'un opérateur de clôture caractérisent bien le processus d'abstraction. Une clôture est un opérateur $\rho : E \to E$

- extensif : $\forall x \in E,\ x \sqsubseteq \rho(x)$. Cette propriété représente le fait qu'après approximation d'un élément concret $x$, on obtient un élément abstrait $\rho(x)$ qui contient au plus la même quantité d'information que l'élément initial.
- monotone : $\forall x, y \in E,\ x \sqsubseteq y \Rightarrow \rho(x) \sqsubseteq \rho(y)$. Cette propriété assure que le processus d'approximation préserve la relation de précision : si un élément concret $x$ contient plus d'information qu'un élément concret $y$, alors après approximation on est certain que $\rho(x)$ est plus précis que $\rho(y)$.
- idempotent : $\forall x \in E,\ \rho(\rho(x)) = \rho(x)$. Cette propriété représente le fait que le processus d'approximation ne perd d'information que lors de sa première application. Le processus peut donc être itéré sans perte supplémentaire de précision.

Soient $(E, f_1, g_1, F_1)$ et $(E, f_2, g_2, F_2)$ deux correspondances de Galois. Soient $\rho_1,\ \rho_2 \in uco(E)$ tels que $\rho_1(E) \cong F_1$ et $\rho_2(E) \cong F_2$. $F_1 \sqsubseteq F_2 \Leftrightarrow \rho_1 \sqsubseteq \rho_2 \Leftrightarrow \rho_2(E) \subseteq \rho_1(E)$.
Deux domaines abstraits $F_1$ et $F_2$ d'un même domaine concret $E$ peuvent donc être comparés en terme de degré de précision : le domaine $F_1$ est plus précis que le domaine $F_2$ si $\forall y_2 \in F_2, \exists y_1 \in F_1$ tel que $g_1(y_1) = g_2(y_2)$, c'est-à-dire si $g_2(F_2) \subseteq g_1(F_1)$. On note alors $F_1 \sqsubseteq F_2$. Soit $\mathcal{A}$ l'ensemble des domaines abstraits, $\langle \mathcal{A}, \sqsubseteq \rangle$ forme un treillis, appelé treillis des interprétations abstraites et $\langle uco(E), \sqsubseteq, \sqcup, \sqcap, \top_{uco(E)}, \bot_{uco(E)} \rangle$ est isomorphe au treillis $\langle \mathcal{A}, \sqsubseteq \rangle$ des interprétations abstraites.

**Pseudo-complément** On définit le pseudo-complément comme l'opérateur inverse du produit réduit. On a vu que le produit réduit $\sqcap$ sur le treillis $\langle uco(E), \sqsubseteq, \sqcup, \sqcap, \top_{uco(E)}, \bot_{uco(E)} \rangle$ est utilisé pour combiner des domaines abstraits en un domaine abstrait plus précis. Etant donné $\rho_1, \rho_2 \in uco(E)$, tels que $\rho_1 \sqsubseteq \rho_2$ (i.e. $\rho_2(E) \subseteq \rho_1(E)$), on définit le pseudo-complément $\rho_1 \ominus \rho_2$ de la manière suivante :

$$\rho_1 \ominus \rho_2 = \sqcup\{\rho \in uco(E) \mid \rho_2 \sqcap \rho = \rho_1\}$$

c'est-à-dire :

$$(\rho_1 \ominus \rho_2)(E) = \bigcap_{\rho_2 \sqcap \rho = \rho_1} \rho(E).$$

De manière équivalente, on peut formuler cette définition de la manière suivante : Etant donnés deux domaines abstraits $A_1 \sqsubseteq A_2$, le pseudo-complément de $A_2$ est le domaine le plus abstrait $A$ dont le produit réduit avec $A_2$ est exactement égal à $A_1$ : $A_1 \ominus A_2 = \sqcup\{A \in \mathcal{A} \mid A_2 \sqcap A = A_1\}$.

**Correction et complétude de l'approximation d'une opération concrète** Une opération sémantique concrète est formalisée par une fonction $f : C \to C$ sur le domaine concret $C$. Une opération sémantique concrète est approximée sur un domaine abstrait par une opération abstraite $f^{\#}$. Pour que l'approximation soit correcte, la condition suivante doit être vérifiée :

$$\begin{tikzcd}
f(x) \arrow[dr, "\alpha"] & f^{\#}(\alpha(x)) \\
 & \alpha(f(x)) \arrow[u, "f^{\#}"] \\
x \arrow[uu, "f"] \arrow[r, "\alpha"] & \alpha(x) \arrow[uu, dash]
\end{tikzcd}$$

$$\forall x \in C,\ \alpha(f(x)) \sqsubseteq_A f^{\#}(\alpha(x)).$$

La correction d'une approximation peut être définie de manière équivalente par la condition suivante :

$$\begin{tikzcd}
\gamma(f^{\#}(y)) & f^{\#}(y) \arrow[l, "\gamma"'] \\
f(\gamma(y)) & \\
\gamma(y) \arrow[u, "f"] & y \arrow[l, "\gamma"'] \arrow[uu, "f^{\#}"']
\end{tikzcd}$$

$$\forall y \in A,\ f(\gamma(y)) \sqsubseteq_C \gamma(f^{\#}(y)).$$

Etant donnée une opération concrète $f : C \to C$, on peut ordonner les approximations correctes de $f$ relativement à la correspondance de Galois $(C, \alpha, \gamma, A)$. Si $f_1^{\#}$ et $f_2^{\#}$ sont deux approximations correctes de $f$, alors $f_1^{\#}$ est une meilleure approximation que $f_2^{\#}$ si $\forall x \in C,\ f_1^{\#}(x) \sqsubseteq_C f_2^{\#}(x)$, c'est-à-dire si $f_1^{\#} \sqsubseteq f_2^{\#}$.

Etant données une fonction concrète $f : C \to C$ et une correspondance de Galois $(C, \alpha, \gamma, A)$, il existe une meilleure approximation $f^{\#}$ de $f$, donnée par : $f^A = \alpha \circ f \circ \gamma$. En effet, $\alpha \circ f \circ \gamma$ est une approximation correcte de $f$ sur $A$ et pour toute approximation correcte $f^{\#}$ de $f$, on a : $\forall y \in A,\ \alpha(f(\gamma(y))) \sqsubseteq_A f^{\#}(y)$, c'est-à-dire $\alpha \circ f \circ \gamma \sqsubseteq f^{\#}$.

$$\begin{tikzcd}
f(\gamma(y)) \arrow[dr, "\alpha"] & & f^{\#}(y) \\
 & \alpha(f(\gamma(y))) \arrow[ur, "f^{\#}"] & \\
\gamma(y) \arrow[uu, "f"] & & y \arrow[ll, "\gamma"] \arrow[uu, dash]
\end{tikzcd}$$

Une approximation est dite complète lorsqu'elle est correcte et qu'elle préserve la même précision. On distingue deux conditions distinctes pour exprimer la complétude d'une approximation :

**Définition D.1.29** *Soit $(C, \alpha, \gamma, A)$ une correspondance de Galois, $f : C \to C$ une fonction concrète et $f^{\#} : A \to A$ une fonction abstraite. L'approximation $f^{\#}$ est dite $B$-complète si la condition suivante est vérifiée :*

$$\forall x \in C,\ \alpha(f(x)) = f^{\#}(\alpha(x)).$$

*L'approximation $f^{\#}$ est dite $F$-complète si la condition suivante est vérifiée :*

$$\forall y \in A,\ f(\gamma(y)) = \gamma(f^{\#}(y)).$$

La $B$-complétude préserve la précision d'une approximation sur les sorties de l'opération concrète.

$$\begin{tikzcd}
f(x) \arrow[dr, "\alpha"] & & f^{\#}(\alpha(x)) \arrow[d, equal] \\
 & & \alpha(f(x)) \\
x \arrow[uu, "f"] \arrow[rr, "\alpha"] & & \alpha(x) \arrow[uu, "f^{\#}"]
\end{tikzcd}$$

La $F$-complétude préserve la précision d'une approximation sur ses entrées.

$$\begin{tikzcd}
\gamma(f^{\#}(y)) \arrow[d, equal] & & f^{\#}(y) \arrow[ll, "\gamma"] \\
f(\gamma(y)) & & \\
\gamma(y) \arrow[u, "f"] & & y \arrow[ll, "\gamma"] \arrow[uu, "f^{\#}"]
\end{tikzcd}$$

Si tout domaine abstrait induit l'existence d'une meilleure approximation correcte d'une opération sémantique concrète, tous les domaines abstraits n'induisent pas une abstraction $B$-complète ou $F$-complète. Cependant, s'il existe une approximation complète $f^{\#}$ pour $f$ sur le domaine abstrait $\alpha(C)$, alors $\alpha \circ f \circ \gamma$ définit une approximation complète pour $f$ sur $\alpha(C)$.

**Théorème de transfert de point fixe**

**Théorème D.1.1** *(Théorème de transfert de point fixe [CC79]). Etant donnée une correspondance de Galois* $(C, \alpha, \gamma, A)$ *et une fonction monotone* $f : C \to C$.

- *Si* $\alpha \circ f \sqsubseteq f^{\#} \circ \alpha$, *alors :*

$$\alpha(lfp^{\sqsubseteq_C} f) \sqsubseteq_A lfp^{\sqsubseteq_A} f^{\#} = lfp^{\sqsubseteq_A}(\alpha \circ f \circ \gamma).$$

  *Donc, le plus petit point fixe de la meilleure approximation correcte* $f^{\#} = \alpha \circ f \circ \gamma$ *de* $f$ *sur* $A$ *fournit une approximation correcte de* $lfp^{\sqsubseteq_C} f$ *sur* $A$.

- *Si* $\alpha \circ f = f^{\#} \circ \alpha$, *alors :*

$$\alpha(lfp^{\sqsubseteq_C} f) = lfp^{\sqsubseteq_A} f^{\#} = lfp^{\sqsubseteq_A}(\alpha \circ f \circ \gamma).$$

  *En d'autre termes, si l'approximation* $f^{\#}$ *est* $B$*-complète, alors le plus petit point fixe de la meilleure approximation correcte de* $f$ *sur* $A$ *fournit une approximation complète de* $lfp^{\sqsubseteq_C} f$ *sur* $A$.

Nous retrouvons donc la notion de correction qui, appliquée par exemple à l'analyse du flot de données par propagation de la constante, nous assure que notre abstraction des données manipulées par le programme ne va pas nous faire affirmer à tort qu'une variable est constante si ce n'est pas le cas. Nous ne sommes cependant pas assurés de détecter toutes les constantes.

## D.2 Modélisation (et hiérarchisation) des sémantiques d'un programme

Cette approche permet de rendre compte des actions effectuées par un attaquant lorsqu'il analyse statiquement un programme, et qu'il cherche à extraire le chemin d'exécution du programme d'un ensemble de chemins d'exécutions volontairement rendu inextricable.

L'analyse statique peut être formalisée dans le modèle de l'interprétation abstraite [CC77a, CC77b, CC79]. L'analyse par interprétation abstraite des sémantiques concrètes d'un programme est un formalisme utilisé par les concepteurs de compilateurs, lorsqu'ils souhaitent implémententer des transformations d'optimisation. De nombreuses méthodes d'analyse statique ont été développées sur cette base théorique.

**Système de transitions** Un système de transition $\tau = (\Sigma[\![P]\!], \Sigma_i[\![P]\!], T[\![P]\!])$ est caractérisé par :

- un ensemble des états $\Sigma[\![P]\!]$, défini comme l'ensemble des couples $\sigma = \langle \rho, C \rangle$, où $\rho \in \mathcal{E}[\![P]\!]$ représente l'environnement du programme $P$ et $C \in \mathbb{C}[\![P]\!]$ représente l'instruction du programme $P$ qui va s'exécuter dans l'environnement $\rho$ ;
- un ensemble d'états initial $\Sigma_i[\![P]\!] \subseteq \Sigma[\![P]\!]$ ;
- une relation $T[\![P]\!]$ sur l'ensemble des états $\Sigma[\![P]\!]$ du programme $P$ : $T[\![P]\!] \subseteq \Sigma[\![P]\!] \times \Sigma[\![P]\!]$.

À titre d'exemple, le programme :

```
x=0;
while (x<100) {
    x+=1;
}
```

peut être formalisé par le système de transition :

$$(\Sigma[\![P]\!], \Sigma_i[\![P]\!], T[\![P]\!]) = (\mathbb{Z}, \{0\}, \{\langle x, x' \rangle \mid x < 100 \wedge x' = x + 1\}).$$

**Langage de programmation.** Reprenons - adoptons - la syntaxe abstraite générale d'un programme impératif telle que définie par Cousot & Cousot dans [CC02] :

| | | syntaxe | description |
|---|---|---|---|
| $n$ | : | $\mathbb{Z}$ | nombres entiers |
| $X$ | : | $\mathbb{X}$ | variables du programme |
| $E$ | : | $\mathbb{E}$ | expressions arithmétiques |
| | := | $n \vert X \vert E_1 - E_2$ | (entier, variable ou différence) |
| $B$ | : | $\mathbb{B}$ | expression booléenne |
| | := | $E_1 < E_2 \vert B_1 \vee B_2 \vert \neg B_1 \vert true \vert false$ | (comparaison, disjonction, négation, vrai\|faux) |
| $A$ | : | $\mathbb{A}$ | actions du programme |
| | := | $(X := E) \vert (X :=?) \vert B$ | (affectation, affectation aléatoire, test) |
| $L$ | : | $\mathbb{L}$ | étiquettes |
| $l$ | $\notin$ | $\mathbb{L}$ | étiquette indéfinie |
| $C$ | : | $\mathbb{C}$ | commandes |
| | := | $L_1 : A \rightarrow L_2$ | (transitions ou instructions) |
| $P$ | : | $\mathbb{P}$ | programmes ($\mathbb{P} = \mathcal{P}(\mathbb{C})$) |

Le programme d'exemple précédent peut être représenté en utilisant uniquement des instructions de la forme : $L : A \rightarrow L'$ :

$$\begin{array}{l} L_1 : X := 0 \rightarrow L_2\,; \\ L_2 : (X < 100) \rightarrow L_3\,; \\ L_2 : \neg(X < 100) \rightarrow L_5\,; \\ L_3 : X := X + 1 \rightarrow L_4\,; \\ L_4 : true \rightarrow L_2\,; \text{ (skip)} \\ L_5 : true \rightarrow l\,; \text{ (stop)} \end{array}$$

Soit $var[\![D]\!]$, l'ensemble des variables d'une expression arithmétique, d'une expression booléenne ou d'une action $D \in \mathbb{E} \cup \mathbb{B} \cup \mathbb{A}$. On définit les sémantiques suivantes :

$$\begin{array}{ll} lab[\![L_1 : A \rightarrow L_2]\!] = L_1 & lab[\![P]\!] = \bigcup_{C \in P} lab[\![C]\!] \\ var[\![L_1 : A \rightarrow L_2]\!] = var[\![A]\!] & var[\![P]\!] = \bigcup_{C \in P} var[\![C]\!] \\ suc[\![L_1 : A \rightarrow L_2]\!] = L_2 & act[\![L_1 : A \rightarrow L_2]\!] = A \end{array}$$

Les commandes d'un programme $P$ agissent sur des variables $X \in var[\![P]\!]$. Les variables d'un programme prennent leurs valeurs dans le domaine sémantique $\mathcal{D}$. Un environnement $\rho \in \mathcal{E}$ est une fonction qui à une variable $X$ associe sa valeur $\rho(X)$. Les sémantiques $E[\![E]\!]\rho$ d'une expression arithmétique $E$ sont définies par induction de la manière suivante :

$$\begin{array}{ll} E[\![n]\!]\rho = n & E[\![X]\!]\rho = \rho(X) \\ E[\![E_1 - E_2]\!] = E[\![E_1]\!]\rho - E[\![E_2]\!]\rho & \end{array}$$

On définit de même les sémantiques d'une expression booléenne :

$$\begin{array}{ll} B[\![true]\!]\rho = true & B[\![false]\!]\rho = false \\ B[\![E_1 < E_2]\!]\rho = E[\![E_1]\!]\rho < E[\![E_2]\!]\rho & B[\![\neg B]\!]\rho = \neg B[\![B]\!]\rho \\ B[\![B_1 \vee B_2]\!]\rho = B[\![B_1]\!]\rho \vee B[\![B_2]\!]\rho & \end{array}$$

et les sémantiques d'une action, en terme d'effet de son exécution sur l'environnement $\rho$ :

$$\begin{array}{ll} A[\![B]\!]\rho = \{\rho' \mid B[\![B]\!]\rho' = true \wedge \rho' = \rho\} & A[\![X := E]\!]\rho = \{\rho(X := E[\![E]\!]\rho)\} \\ A[\![X :=?]\!]\rho = \{\rho' \mid \exists z \in \mathbb{Z} : \rho' = \rho(X := z)\} & A[\![true]\!]\rho = A[\![skip]\!]\rho = \{\rho\} \end{array}$$

Soit $D \in \mathbb{E} \cup \mathbb{B}$. Le résidu $R[\![D]\!]\rho$ de $D$ dans un environnement $\rho$ est l'expression résultant de la spécialisation de $D$ dans cet environnement. Lorsque $var[\![D]\!] \subseteq dom(p)$, l'expression $D$ est dite statique dans l'environnement $\rho$, sinon elle est dite dynamique.

Spécialisation d'une expression arithmétique :

$$\begin{array}{ll} R[\![n]\!]\rho = n & R[\![X]\!]\rho = \rho(X) \text{ si } X \in dom(\rho), X \text{ sinon} \\ R[\![E_1 - E_2]\!] = R[\![E_1]\!]\rho - R[\![E_2]\!]\rho & \end{array}$$

Spécialisation d'une expression booléenne :

$$\begin{array}{ll} R[\![true]\!]\rho = true & R[\![false]\!]\rho = false \\ R[\![E_1 < E_2]\!]\rho = B[\![E_1]\!]\rho < B[\![E_2]\!]\rho & R[\![\neg B]\!]\rho = \neg B[\![B]\!]\rho \\ R[\![B_1 \vee B_2]\!]\rho = R[\![B_1]\!]\rho \vee R[\![B_2]\!]\rho & \end{array}$$

Spécialisation d'une action :

$$R[\![B]\!]\rho = \langle \rho, R[\![B]\!]\rho \rangle \quad R[\![X :=?]\!]\rho = \langle \rho \backslash X, X :=? \rangle$$

$$R[\![X := E]\!]\rho = \begin{cases} \langle \rho \backslash X, X := R[\![E]\!]\rho \rangle & \text{si } E \text{ est dynamique} \\ \langle \rho(X = R[\![E]\!]\rho), skip \rangle & \text{si } E \text{ est statique} \end{cases}$$

**Hiérarchie des sémantiques, sémantique dénotationnelle, sémantique des traces, transfert du point fixe** Dans [Cou02], Cousot a défini une hierarchie des sémantiques, montrant que la plupart des sémantiques connues peuvent être dérivées par interprétation abstraite de la sémantique des traces maximales, aussi appelée sémantique concrète.

Il existe de nombreuses méthodes de descriptions de sémantiques (opérationnelle, dénotationnelle, axiomatique) qui sont équivalentes ou sont des approximations les unes des autres. Elles forment donc une hierarchie, organisée selon la précision de la description des comportements des programmes à l'exécution. L'idée clé de l'interprétation abstraite est que le comportement d'un programme à différents niveaux d'abstraction est une approximation de ses sémantiques concrètes.

Les sémantiques d'un programme peuvent généralement se définir par point fixe. Notons $\Sigma^+$ l'ensemble des traces maximales finies, $\Sigma^w$ l'ensemble des traces maximales infinies et $\Sigma^\infty = \Sigma^+ \cup \Sigma^w$ l'ensemble des traces maximales finies ou infinies.
Considérons l'ensemble $S^n[\![P]\!]$ des traces maximales de taille $n$ engendrées par un système de transitions $\tau$. $S^n[\![P]\!] \subseteq \Sigma^+$ est la réunion de :

- l'ensemble des traces finies de longueur 1 réduites à un état initial $\sigma_0$, sans transition possible : $\forall s, \langle \sigma_0, s \rangle \notin T[\![P]\!]$ ;
- l'ensemble des traces finies de longueur $n > 1$ commençant par une transition $\langle \sigma_0, \sigma_1 \rangle \in T[\![P]\!]$ prolongées par une trace finie $\sigma_1 \ldots \sigma_{n-1} \in S^{n-1}[\![P]\!]$.

Avec cette construction, on est certain qu'aucune trace finie n'est préfixe d'une trace plus longue. C'est la raison pour laquelle on parle de traces maximales. On définit l'ensemble des traces maximales finies $S^+[\![P]\!] \subseteq \Sigma^+$ engendrée par un système de transitions $\tau$ comme la réunion des traces maximales de taille $n$ :

$$S^+[\![P]\!] = \bigcup_{n \geq 0} S^n[\![P]\!].$$

Considérons la fonction $F^+[\![P]\!] : \mathcal{P}(\Sigma^+) \to \mathcal{P}(\Sigma^+)$ définie par :

$$\begin{array}{rcl} F^+[\![P]\!](X) & = & \{\sigma_0 \in \Sigma^+ \mid |\sigma_0| = 1 \wedge \forall s,\ \langle \sigma_0, s \rangle \notin T[\![P]\!]\} \\ & \cup & \{\sigma_0\sigma_1\sigma \in \Sigma^+ \mid \langle \sigma_0, \sigma_1 \rangle \in T[\![P]\!] \wedge \sigma_1\sigma \in X\}. \end{array}$$

On peut montrer que $F^+[\![P]\!](S^+[\![P]\!]) = S^+[\![P]\!]$ :

$$\begin{array}{rcl} F^+[\![P]\!](S^+[\![P]\!]) & = & F^+[\![P]\!](\bigcup_{n\geq 0} S^n[\![P]\!]) \\ & = & \{\sigma_0 | \sigma_0 \in \Sigma^+\} \cup \\ & & \{\sigma_0\sigma_1\sigma \in \Sigma^+ \,|\, \langle\sigma_0,\sigma_1\rangle \in T[\![P]\!] \wedge \sigma_1\sigma \in \bigcup_{n\geq 0} S^n[\![P]\!]\} \\ & = & \{\sigma_0 | \sigma_0 \in \Sigma^+\} \cup \\ & & \bigcup_{n\geq 0} \{\sigma_0\sigma_1\sigma \in \Sigma^+ \,|\, \langle\sigma_0,\sigma_1\rangle \in T[\![P]\!] \wedge \sigma_1\sigma \in S^n[\![P]\!]\} \\ & = & S^1[\![P]\!] \cup \bigcup_{n\geq 0} S^{n+1}[\![P]\!] \\ & = & \bigcup_{n\geq 1} S^n[\![P]\!] \\ & = & \bigcup_{n\geq 0} S^n[\![P]\!] \text{ car } S^0[\![P]\!] = \emptyset \\ & = & S^+[\![P]\!] \end{array}$$

□

De plus, si $Y$ est un autre point fixe de $F^+[\![P]\!]$, on peut démontrer par récurrence que $\forall n \geq 0$, $S^n[\![P]\!] \subseteq Y$. Donc $S^+[\![P]\!]$ est le plus petit point fixe de $F^+[\![P]\!]$ :

$$S^+[\![P]\!] = lfp^{\subseteq} F^+[\![P]\!] = \bigcup_{n\geq 0} F^+[\![P]\!]^n(\emptyset).$$

En effet, soit $Y$ un autre point fixe de $F^+[\![P]\!]$. Montrons par récurrence que $\forall n \geq 0$, $S^n[\![P]\!] \subseteq Y$. Pour $n = 0$, on a : $S^0[\![P]\!] = \emptyset \subseteq Y$. Pour $n = 1$, on a : $S^1[\![P]\!] = \{\sigma \,|\, \sigma \in \Sigma\} \subseteq F^+[\![P]\!](Y) = Y$. Supposons que $\sigma_0\sigma \in S^n[\![P]\!] \Rightarrow \sigma_0\sigma \in Y$. On a alors : $\{\sigma_0\sigma \,|\, \langle\sigma_0,\sigma_1\rangle \in T[\![P]\!] \wedge \sigma_1\sigma \in S^n[\![P]\!]\} \subseteq \{\sigma_0\sigma \,|\, \langle\sigma_0,\sigma_1\rangle \in T[\![P]\!] \wedge \sigma_1\sigma \in Y\}$ d'où : $S^{n+1}[\![P]\!] \subseteq F^+[\![P]\!](S^n[\![P]\!]) \subseteq F^+[\![P]\!](Y) = Y$.
□

L'ensemble des traces maximales infinies $\Sigma^w[\![P]\!] \subseteq \Sigma^w$ engendré par un système de transition $\tau$ est l'ensemble des traces infinies $\sigma$ commençant par une transition $\langle\sigma_0,\sigma_1\rangle \in T[\![P]\!]$ et prolongée par une trace infinie $\sigma_1\sigma_2\ldots$ de $S^w[\![P]\!]$.

Considérons la fonction $F^w[\![P]\!] : \mathcal{P}(\Sigma^w) \to \mathcal{P}(\Sigma^w)$ définie par :

$$F^w[\![P]\!](X) \quad = \quad \{\sigma_0\sigma_1\sigma \in \Sigma^w \,|\, \langle\sigma_0,\sigma_1\rangle \in T[\![P]\!] \wedge \sigma_1\sigma \in X\}.$$

On établit comme précédemment que $F^w[\![P]\!](S^w[\![P]\!]) = S^w[\![P]\!]$ et que $S^w[\![P]\!]$ est le plus grand point fixe de $F^w[\![P]\!]$ :

$$S^w[\![P]\!] = gfp^{\subseteq} F^w[\![P]\!] = \bigcap_{n\geq 0} F^w[\![P]\!]^n(\top).$$

Soit $ucp(\mathcal{P}(\Sigma^\infty)$ le treillis des sémantiques abstraites. Chaque clôture de $ucp(\mathcal{P}(\Sigma^\infty)$ peut être vue comme une abstraction de la sémantique des traces maximales.

La sémantique dénotationnelle $S_d$ préserve le comportement observable d'un programme, i.e. son comportement en termes d'entrées/sorties. Elle abstrait l'historique des calculs en n'observant que les relations d'entrées/sorties des traces finies :

$$\{\sigma \in \Sigma^+ \,|\, \exists\sigma' \in T[\![P]\!] \cap \Sigma^+ \; \sigma_0 = \sigma'_0 \text{ et } \sigma_F = \sigma'_F\}$$

et les entrées sur lesquelles les traces infinies divergent :

$$\{\sigma \in \Sigma^w \,|\, \exists\sigma' \in T[\![P]\!] \cap \Sigma^w \; \sigma_0 = \sigma'_0\}.$$

On a donc :

$$S_d(X) = \{\sigma \in \Sigma^+ \,|\, \exists\sigma' \in X \cap \Sigma^+ \; \sigma_0 = \sigma'_0 \text{ et } \sigma_F = \sigma'_F\} \cup \{\sigma \in \Sigma^w \,|\, \exists\sigma' \in X \cap \Sigma^w \; \sigma_0 = \sigma'_0\}.$$

La sémantique dénotationnelle $S_d$ peut être vue comme une abstraction de la sémantique des traces maximales (voir la hierarchie des sémantiques d'un système de transition [Cou02]).

Les sémantiques concrètes $S[\![P]\!]$ d'un programme $P$ sont usuellement données sous la forme du plus petit point fixe d'une fonction $F[\![P]\!]$ : $F[\![P]\!](S[\![P]\!]) = S[\![P]\!]$ et

$$S[\![P]\!] = lfp^{\sqsubseteq_C} F[\![P]\!] = \bigcup_{n \geq 0} F[\![P]\!]^n(\bot_C).$$

Considérons la correspondance de Galois $(C, \alpha, \beta, A)$. Lorsque la fonction $F[\![P]\!] : C \to C$ est monotone, le théorème de transfert du point fixe (théorème D.1.1) nous donne un moyen pour construire une sémantique abstraite du programme $P$ de la forme :

$$S^{\#}[\![P]\!] = lfp^{\sqsubseteq_A} F^{\#}[\![P]\!] = \bigcup_{n \geq 0} F^{\#}[\![P]\!]^n(\bot_A)$$

où la transformation sémantique abstraite $F^{\#}[\![P]\!] = (\gamma \circ F \circ \alpha)[\![P]\!]$ est conçue en utilisant la condition de commutation locale suivante :

$$\alpha \circ F[\![P]\!] \sqsubseteq_A F^{\#}[\![P]\!] \circ \alpha.$$

On est alors assuré de la correction de l'approximation $S^{\#}[\![P]\!]$ de $S[\![P]\!]$ sur $A$ :

$$\alpha(S[\![P]\!]) = \alpha(lfp^{\sqsubseteq_C} F[\![P]\!]) \sqsubseteq_A lfp^{\sqsubseteq_A} F^{\#}[\![P]\!] = S^{\#}[\![P]\!].$$

Lorsque le domaine abstrait $A$ est $B$-complet pour la fonction de transfert $F[\![P]\!]$, alors les sémantiques abstraites coincident avec l'abstraction des sémantiques concrètes. Donc, lorsque le domaine abstrait $A$ est $B$-complet pour la fonction de transfert $F[\![P]\!]$, le plus petit point fixe $lfp^{\sqsubseteq_A} F^{\#}[\![P]\!]$ de la meilleur approximation correcte $F^{\#}[\![P]\!] = \alpha \circ F[\![P]\!] \circ \gamma$ de $F[\![P]\!]$ sur $A$ fournit une approximation complète des sémantiques concrètes :

$$\alpha(S[\![P]\!]) = lfp^{\sqsubseteq_A}(\alpha \circ F \circ \gamma)[\![P]\!].$$

## D.3 Modélisation de l'analyse du flot de données, théorème du point fixe

Prenons l'exemple du problème d'analyse de flot de données consistant à déterminer la portée des définitions. La résolution d'un système d'équations de flot nous permet de déterminer quelles définitions atteignent quels noeuds du graphe CFG [Muc97].

La propagation du vecteur de bits à travers le graphe CFG peut être modélisé par le système d'équations récursives suivant :

$$\begin{cases} in(i) & = & \bigcap_{(j,i) \in E} out(j) \\ out(i) & = & gen(i) \cup (in(i) - kill(i)) \end{cases}$$

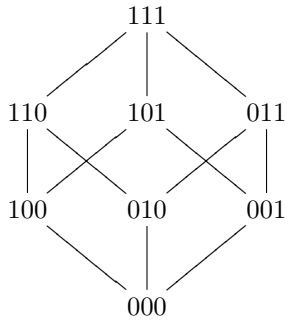
où $in(i)$ est l'ensemble des définitions en entrée du noeud $i$, $out(i)$ est l'ensemble des définitions en sortie du noeud $i$, $gen(i)$ est l'ensemble des définitions générées par le noeud $i$ et $kill(i)$ est l'ensemble des définitions supprimées par le noeud $i$.

Si tous les problèmes de l'analyse du flot de données ne peuvent pas être résolus par l'utilisation d'un simple vecteur de bits, de nombreux problème de l'analyse du flot de données peuvent être résolus par une telle approche équationelle. Ces problèmes sont alors formulés en utilisant un système d'équation de flot plus général :

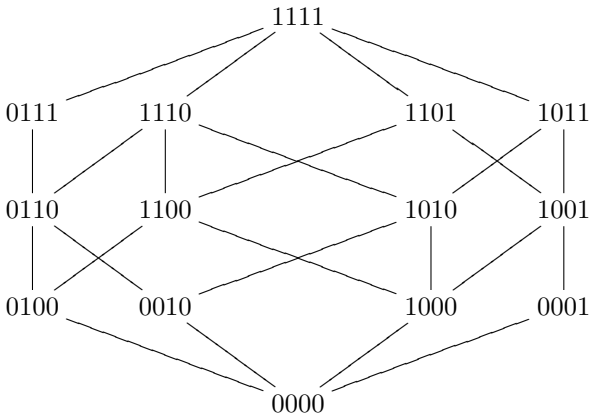
$$\begin{cases} in(i) & = & \bigsqcap_{(j,i) \in E} out(j) \\ out(i) & = & F_i(in(i)) \end{cases}$$

où $F_i$ est la fonction de transfert qui modifie les valeurs du flot de données lorsqu'il se propage à travers le noeuds $i$ et $\bigsqcap$ est l'opérateur de combinaison des valeurs de flots venants des différents prédécesseurs.

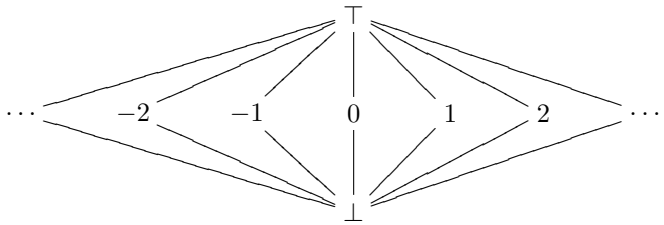
Dans le cas d'un programme comportant $n$ variables, le treillis $A^n$ est utilisé. À titre d'exemple, le domaine abstrait $A^3$ est le treillis suivant :




et le domaine abstrait $A^4$ est le treillis suivant :




Le théorème du point fixe nous permet de calculer le plus petit point fixe de la fonction de transfert vectorielle $F^{\#}[\![P]\!] : A^m \to A^m$ où $m$ représente le nombre de noeuds du graphe CFG du programme $P$ considéré, dès lors que toutes les composantes de $F^{\#}[\![P]\!]$ - les fonctions de transfert associées à chaque noeud du graphe - sont continues (théorème D.1.1). La solution fournie pas cet algorithme itératif est alors exactement la solution MFP - MFP$[\![P]\!] = lfp^{\sqsubseteq_A} F^{\#}[\![P]\!]$. Lorsque toutes les composantes de $F^{\#}[\![P]\!]$ sont distributives, l'algorithme itératif fournit également la solution MOP - MFP$[\![P]\!]$ =MOP$[\![P]\!]$. Dans le cas contraire, on a : MOP$[\![P]\!] \sqsubset lfp^{\sqsubseteq_A} F^{\#}[\![P]\!]$ =MFP$[\![P]\!]$ [Kil73, KU77]. Si de nombreux problèmes de l'analyse du flot de donnée sont distributifs, tous ne le sont pas, comme par exemple le problème de propagation de la constante, pour lequel le domaine abstrait suivant est généralement considéré :




L'analyse du flot de données par propagation de la constante a pour but de découvrir les valeurs qui restent constantes sur toutes les exécutions d'un programme et de propager la valeur de ces constantes aussi loin que possible dans le flot d'exécution du programme.
Cette technique d'analyse statique a plusieurs utilisations, dans le cadre de l'optimisation des programmes. Les transformations d'optimisation peuvent être appliquée à la compilation [Muc97] et à la décompilation des programmes [Cif96]. L'application principale dans le cadre de la rétroingénierie concerne la détection de code mort (jamais exécuté ou inatteignable), la détection des chemins d'exécution qui ne sont jamais empruntés (en identifiant les branchements conditionnels constants, i.e. conditionnés par un prédicat constant). L'analyse du flot de données

par propagation de la constante conditionne donc l'application de transformations d'optimisation du flot de contrôle, permettant d'élaguer le graphe de flot et peut s'appliquer éventuellement de manière itérée.

Bien que le problème de propagation de la constante soit généralement indécidable, il est possible de proposer des algorithmes approximatifs corrects (toutes les constantes ne sont pas nécessairement détectées, mais chaque constante détectée l'est pour toutes les exécutions possibles du programme).

Considérons par exemple le programme suivant [CC77b] :

```
a=1; c=0;
while( - ) do {
        b=2;
        d=a+b;
        e=b+c;
        c=4;
}
```

Les contextes abstraits associés aux points d'exécution du programme sont des vecteurs qui associent une valeur abstraite à chaque variable du programme. $\top$ représente la valeur d'une variable non-constante. $\bot$ représente la valeur d'une variable non initialisée. Les autres valeurs du treillis représentent les constantes.
Le système d'équation correspondant à notre exemple est le suivant :

$$\begin{cases} f_1(x_1,x_2,x_3,x_4,x_5) & = & (\top,\top,\top,\top,\top) \\ f_2(x_1,x_2,x_3,x_4,x_5) & = & (1, f_1(x_2), 0, f_1(x_4), f_1(x_5)) \\ f_3(x_1,x_2,x_3,x_4,x_5) & = & f_2(x_1,x_2,x_3,x_4,x_5) \sqcup f_7(x_1,x_2,x_3,x_4,x_5) \\ f_4(x_1,x_2,x_3,x_4,x_5) & = & (f_3(x_1), 2, f_3(x_3), f_3(x_4), f_3(x_5)) \\ f_5(x_1,x_2,x_3,x_4,x_5) & = & (f_4(x_1), f_4(x_2), f_4(x_3), f_4(x_1)+f_4(x_2), f_4(x_5)) \\ f_6(x_1,x_2,x_3,x_4,x_5) & = & (f_5(x_1), f_5(x_2), f_5(x_3), f_5(x_4), f_5(x_2)+f_5(x_3)) \\ f_7(x_1,x_2,x_3,x_4,x_5) & = & (f_6(x_1), f_6(x_2), 4, f_6(x_4), f_6(x_5)) \end{cases}$$

L'addition dans le domaine abstrait vérifie :

$$\forall x \in L,\ \bot + x = x + \bot = \bot \text{ et } \forall x \in L - \{\bot\},\ \top + x = x + \top = \top$$

Puisque toute chaîne strictement croissante de $L$ est de longueur au plus égale à 3, $(L^5)^7$ vérifie la condition ACC. La méthode de résolution du système d'équation itérative doit donc converger en au plus $3 \times 5 \times\ 7 + 1$ itérations successives. Les approximations successives sont les suivantes :

| | $a$ | $b$ | $c$ | $d$ | $e$ |
|---|---|---|---|---|---|
| $f_i$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| $f_1$ | $\top$ | $\top$ | $\top$ | $\top$ | $\top$ |
| $f_2$ | 1 | $\top$ | 0 | $\top$ | $\top$ |
| $f_2$ | 1 | $\top$ | 0 | $\top$ | $\top$ |
| $f_4$ | 1 | 2 | 0 | $\top$ | $\top$ |
| $f_5$ | 1 | 2 | 0 | 3 | $\top$ |
| $f_6$ | 1 | 2 | 0 | 3 | 2 |
| $f_7$ | 1 | 2 | 4 | 3 | 2 |
| $f_3$ | 1 | $\top$ | $\top$ | $\top$ | $\top$ |
| $f_4$ | 1 | 2 | $\top$ | $\top$ | $\top$ |
| $f_5$ | 1 | 2 | $\top$ | 3 | $\top$ |
| $f_6$ | 1 | 2 | $\top$ | 3 | $\top$ |
| $f_7$ | 1 | 2 | 4 | 3 | $\top$ |
| $f_3$ | 1 | $\top$ | $\top$ | $\top$ | $\top$ |
| $f_4$ | 1 | 2 | $\top$ | $\top$ | $\top$ |
| $f_5$ | 1 | 2 | $\top$ | 3 | $\top$ |
| $f_6$ | 1 | 2 | $\top$ | 3 | $\top$ |
| $f_7$ | 1 | 2 | 4 | 3 | $\top$ |

Nous pouvons constater que le système se stabilise bien après 3 itérations. Le résultat de cette analyse du flot de données par propagation de la constante est le suivant : la variable $a$ est constante égale à 1. Les variables $b$ et $d$ sont constantes dans la boucle, dès lors qu'elles ont été initialisées.

## D.4 Modélisation des transformations de programmes

Avant d'essayer de formaliser le glaive et le bouclier dans ce contexte, présentons brièvement le cadre formel développé par Cousot et Cousot [CC02] permettant de définir la notions de transformation appliquée à un programme par interprétation abstraite. On note :

- $\mathbb{P}$ un langage de programmation - ou de manière équivalente l'ensemble des programmes syntaxiquement corrects écrits dans ce langage ;
- $\mathtt{t} : \mathbb{P} \to \mathbb{P}$ une transformation syntaxique qui à un programme $P \in \mathbb{P}$ associe le programme transformé $\mathtt{t}(P) \in \mathbb{P}$ ;
- $po\langle \mathcal{D}, \sqsubseteq \rangle$ le domaine des sémantiques concrètes ;
- $S : \mathbb{P} \to po\langle \mathcal{D}, \sqsubseteq \rangle$ l'application qui à un programme $P \in \mathbb{P}$ associe ses sémantiques concrètes $S[\![P]\!] \in po\langle \mathcal{D}, \sqsubseteq \rangle$ ;
- $\mathtt{p} : po\langle \mathcal{D}, \sqsubseteq \rangle \to po\langle \mathbb{P}/\equiv, \sqsubseteq \rangle$ la transformation qui à $d \in \mathcal{D}$ associe le programme $\mathtt{p}[d]$ le plus simple dont les sémantiques concrètes sur-approximent $d \in \mathcal{D}$. Il est commode de définir Les éléments de $\mathbb{P}$ comme des classes d'équivalence, définis à une équivalence syntaxique près : $P \equiv Q \Leftrightarrow S[\![P]\!] = S[\![Q]\!]$. En effet, on peut avoir $\mathtt{p}(S[\![P]\!]) \sqsubset P$ - par exemple parce que du code mort est présent dans le programme $P$ mais absent des sémantiques concrètes $S[\![P]\!]$ du programme $P$ - alors que l'on a toujours $\mathtt{p}(S[\![P]\!]) \equiv P$.

En considérant les programmes comme des abstractions de leurs sémantiques concrètes, on obtient l'insertion de Galois :

$$po\langle \mathcal{D}, \sqsubseteq \rangle \underset{\mathtt{p}}{\overset{S}{\leftrightarrows}} po\langle \mathbb{P}/\equiv, \sqsubseteq \rangle.$$

Cette insertion de Galois formalise en terme d'interprétation abstraite la correspondance existant entre la syntaxe et les sémantiques concrètes des programmes.

Nous pouvons à présent explorer les liens existants entre transformations sémantiques et transformations syntaxiques. Considérons l'insertion de Galois :

$$po\langle \mathcal{D}, \sqsubseteq \rangle \underset{\mathtt{p}}{\overset{S}{\leftrightarrows}} po\langle \mathbb{P}/\equiv, \sqsubseteq \rangle.$$

On a : $\mathtt{p}(S[\![P]\!]) = P$ (par définition d'une insertion de Galois), $\mathtt{p}(t(S[\![P]\!])) \sqsubseteq \mathtt{t}(\mathtt{p}(S[\![P]\!]))$ et l'on peut observer que $\mathtt{p}(t(S[\![P]\!]))$ apparaît comme la décompilation des spécifications sémantiques $t(S[\![P]\!])$ de la transformation $t$ :

$$\begin{tikzcd}
t(S[\![P]\!]) \arrow[dr, "\mathtt{p}"] & \mathtt{t}(\mathtt{p}(S[\![P]\!])) \\
& \mathtt{p}(t(S[\![P]\!])) \arrow[u, "\mathtt{t}"'] \\
S[\![P]\!] \arrow[uu, "t"] \arrow[r, "\mathtt{p}"] & \mathtt{p}(S[\![P]\!]) \arrow[uu, dash]
\end{tikzcd}$$

On a : $t(S(P)) \sqsubseteq S(\mathtt{t}(P))$

$$\begin{tikzcd}
S[\![\mathtt{t}(P)]\!] & \mathtt{t}(P) \arrow[l, "S"] \\
t(S[\![P]\!]) & \\
S[\![P]\!] \arrow[u, "t"] & P \arrow[l, "S"] \arrow[uu, "\mathtt{t}"']
\end{tikzcd}$$

En introduisant une transformation d'optimisation, formalisée par la correspondance de Galois suivante :

$$po\langle \mathcal{D}, \sqsubseteq\rangle \underset{t}{\overset{\gamma_t}{\leftrightarrows}} po\langle \mathcal{D}, \sqsubseteq\rangle$$

et en composant cette transformation de Galois avec l'insertion de Galois $po\langle \mathcal{D}, \sqsubseteq\rangle \underset{\mathbb{p}}{\overset{S}{\leftrightarrows}} po\langle \mathbb{P}/\equiv, \sqsubseteq\rangle$, on obtient :

$$po\langle \mathbb{P}/\equiv, \sqsubseteq\rangle \underset{\mathbb{t}}{\overset{\gamma_{\mathbb{t}}}{\leftrightarrows}} po\langle \mathbb{P}/\equiv, \sqsubseteq\rangle.$$

En appliquant le théorème de transfert de point fixe successivement à l'insertion de Galois : $po\langle \mathcal{D}, \sqsubseteq\rangle \underset{t}{\overset{\gamma_t}{\leftrightarrows}} po\langle \mathcal{D}, \sqsubseteq\rangle$ et à la correspondance de Galois : $po\langle \mathcal{D}, \sqsubseteq\rangle \underset{\mathbb{p}}{\overset{S}{\leftrightarrows}} po\langle \mathbb{P}/\equiv, \sqsubseteq\rangle$, on obtient :

$$\begin{array}{rcl} \mathbb{p}(t(S[\![P]\!])) & = & \mathbb{p}(t(lfp^{\sqsubseteq}F[\![P]\!])) \\ & \sqsubseteq & \mathbb{p} \circ lfp^{\sqsubseteq}(t \circ F \circ \gamma_t)[\![P]\!] \\ & \sqsubseteq & lfp^{\sqsubseteq}(\mathbb{p} \circ t \circ F \circ \gamma_t \circ S)[\![P]\!] \end{array}$$

En posant $\mathbb{F}[\![P]\!] = \mathbb{p} \circ t \circ F \circ \gamma_t \circ S[\![P]\!]$ et $\mathbb{t}[\![P]\!] = lfp^{\sqsubseteq}\mathbb{F}[\![P]\!]$, on a donc : $\mathbb{p}(t(S[\![P]\!])) \sqsubseteq \mathbb{t}[\![P]\!]$. Si le point fixe $lfp^{\sqsubseteq}\mathbb{F}[\![P]\!]$ est définit sur un ensemble partiellement ordonné vérifiant la condition des chaînes ascendantes, on est alors en mesure de calculer $\mathbb{t}[\![P]\!]$ - en un nombre fini d'itérations - en appliquant l'algorithme itératif :

$$\mathbb{t}[\![P]\!] = lfp^{\sqsubseteq}\mathbb{F}[\![P]\!] = \bigvee_{n \geq 0} \mathbb{F}[\![P]\!]^n(\bot).$$

Remarquons à présent que si $t = S \circ \mathbb{t} \circ \mathbb{p}$ alors $S \circ \mathbb{t} = t \circ S$. En effet, si $t = S \circ \mathbb{t} \circ \mathbb{p}$, alors $t \circ S = S \circ \mathbb{t} \circ \mathbb{p} \circ S = S \circ \mathbb{t} \circ Id = S \circ \mathbb{t}$ Si $t = S \circ \mathbb{t} \circ \mathbb{p}$, alors $\mathbb{p} \circ t \circ S = \mathbb{p} \circ S \circ \mathbb{t} = \mathbb{t}$.
La condition $t = S \circ \mathbb{t} \circ \mathbb{p}$ induit donc la $F$-complétude de l'approximation $\mathbb{t}$ de $t$. Intuitivement, cette condition correspond à l'existence - pour la transformation sémantique $t$ - d'une transformation syntaxique dont les effets sur la sémantique des programmes sont exactement les mêmes que ceux produits par la transformation $t$. C'est donc sous cette hypothèse que le diagramme suivant commute :

$$\begin{array}{ccc} P & \xmapsto{\mathbb{t}} & \mathbb{t}[\![P]\!] = \mathbb{p}[t[S[\![P]\!]]] \\ S \downarrow \uparrow \mathbb{p} & & S \downarrow \uparrow \mathbb{p} \\ S[\![P]\!] & \xmapsto{t} & t[S[\![P]\!]] = S[\![\mathbb{t}[\![P]\!]]\!] \end{array}$$

Nous avons donc établi que sous l'hypothèse d'existence d'une transformation syntaxique dont les effets sur la sémantique des programmes sont exactement les mêmes que ceux produits par la transformation sémantique $t$, on a la correspondance suivante entre transformation syntaxique et transformation sémantique :

- une transformation sémantique $t$ appliquée aux sémantiques concrètes $S[\![P]\!] \in po\langle \mathcal{D}, \sqsubseteq\rangle$ d'un programme $P$ induit une transformation syntaxique sur $\mathbb{P}$ :

  $$\mathbb{t}[\![P]\!] = \mathbb{p}(t(S[\![P]\!])).$$

  Une transformation syntaxique peut donc être vue comme une abstraction de la transformation sémantique correspondante.
- une transformation syntaxique $\mathbb{t}$ appliquée à un programme $P$ induit une transformation sémantique $t$ sur ses sémantiques concrètes :

  $$t[S[\![P]\!]] = S[\![\mathbb{t}[\![\mathbb{p}(S[\![P]\!])]\!]\!]$$

  Une transformation sémantique peut donc être vue comme une concrétisation de la transformation syntaxique correspondante.

La condition $t[S[\![P]\!]] = S[\![\mathbb{t}[\![\mathbb{p}(S[\![P]\!])]\!]\!]$ correspond à la définition d'une transformation vue comme une compilation par un algorithme.

Introduisons à présent un domaine abstrait $po\langle \mathcal{D}_{\mathcal{O}}, \sqsubseteq_{\mathcal{O}} \rangle$ correspondant à l'observation du domaine concret $po\langle \mathcal{D}, \sqsubseteq \rangle$, et la correspondance de Galois correspondante :

$$po\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha_{\mathcal{O}}}{\overset{\gamma_{\mathcal{O}}}{\leftrightarrows}} po\langle \mathcal{D}_{\mathcal{O}}, \sqsubseteq_{\mathcal{O}} \rangle.$$

Une transformation syntaxique $\mathtt{t}$ est dite conservative vis-à-vis de cette observation - abstraction des sémantiques concrètes - si $\alpha_{\mathcal{O}}(S[\![P]\!]) = \alpha_{\mathcal{O}}(S[\![\mathtt{t}[\![P]\!]]\!])$, i.e. si les programmes $P$ et $\mathtt{t}[\![P]\!]$ sont équivalents vis-à-vis de cette observation : $P \equiv_{\mathcal{O}} \mathtt{t}[\![P]\!]$.

En d'autres termes, une transformation syntaxique est conservative vis-à-vis d'une observation si pour ce niveau d'abstraction, l'observation de l'exécution du programme est équivalente à celle du programme transformé.

$$\begin{tikzcd}
t(S[\![P]\!]) = S[\![\mathtt{t}[\![P]\!]]\!] \arrow[dr, "\alpha_{\mathcal{O}}"] & \\
& \alpha_{\mathcal{O}}(t(S[\![P]\!])) \arrow[d, equal] \\
S[\![P]\!] \arrow[u, "t"] \arrow[r, "\alpha_{\mathcal{O}}"] & \alpha_{\mathcal{O}}(S[\![P]\!])
\end{tikzcd}$$

Sous cette hypothèse, on a : $\alpha_{\mathcal{O}}(S[\![P]\!]) = \alpha_{\mathcal{O}}(S[\![\mathtt{t}[\![P]\!]]\!])$ et

$$\alpha_{\mathcal{O}}(S[\![P]\!]) \sqsubseteq \alpha_{\mathcal{O}}(t[S[\![P]\!]]) \sqsubseteq \alpha_{\mathcal{O}}(S[\![\mathtt{t}[\![P]\!]]\!]),$$

d'où par antisymétrie :

$$\alpha_{\mathcal{O}}(S[\![P]\!]) = \alpha_{\mathcal{O}}(t[S[\![P]\!]]) = \alpha_{\mathcal{O}}(S[\![\mathtt{t}[\![P]\!]]\!]).$$

Le diagramme commutatif suivant récapitule le lien existant entre transformations sémantiques et transformations syntaxiques d'un programme ainsi que la formalisation de la notion de transformation syntaxique conservative vis-à-vis d'un certain niveau d'abstraction :

$$\begin{tikzcd}
P \arrow[d, shift right, "S"'] \arrow[rr, mapsto, "\mathtt{t}"] & & \mathtt{t}[\![P]\!] \sqsupseteq \mathtt{p}[t[S[\![P]\!]]] \arrow[d, shift right, "S"'] \\
S[\![P]\!] \arrow[u, shift right, "\mathtt{p}"'] \arrow[rr, mapsto, "t"] \arrow[dr, shift left, "\alpha_{\mathcal{O}}"] & & t[S[\![P]\!]] \sqsubseteq S[\![\mathtt{t}[\![P]\!]]\!] \arrow[u, shift right, "\mathtt{p}"'] \arrow[dl, shift left, "\gamma_{\mathcal{O}}"] \\
& \alpha_{\mathcal{O}}(S[\![P]\!]) = \alpha_{\mathcal{O}}(t[S[\![P]\!]]) = \alpha_{\mathcal{O}}(S[\![\mathtt{t}[\![P]\!]]\!]) \arrow[ur, shift left, "\gamma_{\mathcal{O}}"] \arrow[ul, shift left, "\alpha_{\mathcal{O}}"] &
\end{tikzcd}$$

Annexe E

# Fondamentaux de la théorie de la complexité

L'objectif de cette annexe est de présenter les résultats théoriques fondamentaux en calculabilité, décidabilité et complexité utilisés en virologie informatique. L'essentiel de cette présentation est fondée sur les ouvrages [Pap94, BB87, RJ87, Soa87, Rey04] et le support de cours [Asa03].

Nous y présentons notamment :
- le modèle de machine à registre et différents types de machines de Turing (machine de Turing déterministe - définition E.4.1, non déterministe - définition E.4.2, avec oracle - définition E.4.5, avec oracle ensembliste définition - E.4.6, probabiliste - définition E.4.7) ;
- le codage d'une machine (nombre de Gödel, section E.5) ;
- les notions de décidabilité et d'indécidabilité d'un problème de décision (définitions E.6.5 à E.6.7) ;
- les classes de complexité (définition E.8.1) et la hiérarchie polynomiale (proposition 11) ;
- la thèse de church (thèse E.11.1), le problème de l'arrêt (théorème E.13.1) et le problème SAT (définition E.10.8).

Cet exposé n'a pas vocation a être complet. Il doit plus être vu comme un glossaire des concepts de la théorie de la complexité auxquels il est fait référence dans cette thèse.

## E.1 Analyse en pire et moyen cas

L'analyse théorique de l'efficacité d'un algorithme (en temps) se fait à une constante multiplicative $k$ près, dans le but de ne pas tenir compte des variations introduites par un changement d'implémentation, de langage de programmation ou d'ordinateur.

Une analyse en pire cas consiste à ne considérer, pour chaque taille $n$, que l'exemplaire (du problème à résoudre, considéré comme une collection éventuellement infinie d'exemplaires) de cette taille sur lequel l'algorithme requiert le plus de temps.

Une analyse en moyenne consiste à déterminer le temps d'exécution en moyenne sur tous les exemplaires de taille $n$.

**Définition E.1.1** *Soit une fonction $f : \mathbb{N} \to \mathbb{R}^+$, l'ordre de $f(n)$, noté $O(f(n))$ est l'ensemble des fonctions $g(n)$ bornées supérieurement par un multiple de $f(n)$, en autant que $n$ soit suffisamment grand.*

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists k \in \mathbb{R}^+ \exists n_0 \in \mathbb{N}, \forall n \geq n_0,\ g(n) \leq kf(n)\}$$

Nous dirons d'un algorithme qui prend un temps dans $O(1)$, $O(n)$, $O(n^2)$, $O(n^3)$, $O(n^k)$, ou $O(k^n)$, qu'il prend un temps constant, linéaire, quadratique, cubique, polynomial, ou exponentiel.

**Définition E.1.2** *Soit une fonction $f : \mathbb{N} \to \mathbb{R}^+$, l'oméga de $f(n)$, noté $\Omega(f(n))$ est l'ensemble des fonctions $g(n)$ bornées inférieurement par un multiple de $f(n)$, en autant que $n$ soit suffisamment grand.*

$$\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists k \in \mathbb{R}^+ \exists n_0 \in \mathbb{N}, \forall n \geq n_0,\, g(n) \geq kf(n)\}$$

**Définition E.1.3** *Soit une fonction $f : \mathbb{N} \to \mathbb{R}^+$, l'ordre exact de $f(n)$, noté $\theta(f(n))$ est l'ensemble des fonctions $g(n)$ bornées supérieurement et inférieurement par des multiples (réels positifs) de $f(n)$, en autant que $n$ soit suffisamment grand.*

$$\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

## E.2 Modèles de machine

Un dispositif de calcul peut être décrit indépendamment de toute réalisation physique comme un dispositif ayant les propriétés suivantes :

- À chaque instant, le dispositif se trouve dans un *état* bien précis et le nombre d'états possibles est fini. Un état particulier désigne le début du calcul.
- Le dispositif comporte une *mémoire* qui peut contenir un nombre donné éventuellement infini d'objets finis mais qui à chaque instant n'en contient qu'un nombre fini.
- Le dispositif est muni d'un *programme* qui est un objet fini contrôlant les changements d'états possibles et les modifications de la mémoire. À chaque instant, le programme choisit l'ensemble des actions à effectuer en fonction du contenu de la mémoire et de l'état dans lequel se trouve la machine. La machine s'arrête si cet état est vide.

On suppose que la représentation des données est obtenue par un codage standard.

#### Modèle de Machine

On appelle *modèle de machine* toute classe de dispositifs de calcul analogues dans le sens qu'ils peuvent être décrits mathématiquement de la même façon. Le mot *machine* est alors utilisé pour désigner un dispositif de calcul quelconque du modèle choisi.

#### Configuration d'une machine

La *configuration instantanée* d'une machine $M$ est un objet $C$ qui décrit complètement à un instant donné :

- l'état dans lequel se trouve la machine ;
- le contenu de la mémoire ;
- l'ensemble des actions que le programme peut déclencher à cet instant.

On appelle *configuration initiale* de la machine $M$ pour l'entrée $x$, la configuration instantanée où

- la mémoire ne contient que la représentation de $x$ ;
- la machine est dans l'état initial.

On appelle *configuration terminale* de la machine $M$, toute configuration instantanée pour laquelle la machine s'arrête. Lorsque la machine $M$ est dans cette configuration, le programme de $M$ ne peut déclencher aucune action.

#### Machine déterministe

Une machine est dite *déterministe*, si pour toute configuration $C$, le programme ne peut choisir qu'un seul ensemble d'actions à déclencher. Une telle machine est dite *séquentielle*, si pour toute configuration $C$, le programme ne peut déclencher au plus qu'une action. Dans le cas contraire, la machine est dite *parallèle*.
Soit $C$ et $C'$ deux configurations instantanées de la machine $M$, on dit que la machine $M$ calcule $C'$ à partir de $C$, si dans la configuration $C$, le programme peut déclencher des actions qui font passer la machine dans la configuration $C'$. On note alors :

$$C \underset{M}{\vdash} C'$$

On appelle calcul de la machine $M$ sur l'entrée $x$ toute suite finie de configurations instantanées $C_0 \dots C_n$ telles que :

$$C_0 \underset{M}{\vdash} C_1 \underset{M}{\vdash} \dots C_i \underset{M}{\vdash} C_{i+1} \dots \underset{M}{\vdash} C_n$$

où $C_0 = C_0(x)$ est la configuration initiale de la machine $M$ pour l'état $x$ et $C_n$ une configuration terminale de la machine $M$.

Soit $M$ une machine, on note $x \overset{M}{\rightsquigarrow} y$ la relation *$M$ calcule $y$ sur l'entrée $x$*. On peut toujours supposer que $\overset{M}{\rightsquigarrow}$ est une relation de $\mathbb{N}$ dans $\mathbb{N}$, dans la mesure où le codage et le décodage des données sont supposés extérieurs aux machines.

**Simulation**

Pour pouvoir comparer deux modèles de machines, on introduit la notion de simulation.

Soient $M$ et $M'$ deux machines de modèles éventuellement différents, on dit que $M'$ simule $M$ si $M'$ calcule $y$ à partir de $x$ si et seulement si $M$ calcule $y$ à partir de $x$, c'est-à-dire :

$$\forall x, y \, x \overset{M}{\rightsquigarrow} y \Longleftrightarrow x \overset{M'}{\rightsquigarrow} y.$$

On impose de plus que pour tout calcul de $M$ sur l'entrée $x$, il existe un calcul de $M'$ sur l'entrée $x$ effectivement constructible qui représente le calcul de $M$ sans ambiguité.

On dit alors que deux modèles de machines sont équivalents si toute machine $M$ de l'un des modèles peut être simulée par une machine $M'$ de l'autre modèle. On impose également que cette équivalence soit effective, c'est-à-dire que l'on puisse effectivement construire $M'$ à partir de $M$.

### E.2.1 Machines à registres

Les machines à registres ont été introduites par Shepherdson et Sturgis en 1963.

La *mémoire* d'une machine à registre est composée d'un nombre virtuellement infini de registres $(R_1, R_2, \ldots)$. Chaque registre contient un entier quelconque. Si l'entier contenu dans le registre $R_i$ est nul, on dit que le registre $R_i$ est *vide*. À un instant donné, il n'y a qu'un nombre fini de registres non vides.

Les seules opérations possibles sur les registres sont l'incrémentation (ajouter 1 au contenu du registre) et la décrémentation (soustraire 1 au contenu du registre si ce registre n'est pas vide).

Le *programme* d'une machine à registres est représenté par un graphe orienté fini $G$ qui contient deux sommets particuliers :
- l'état initial de la machine, de degré entrant nul et de degré sortant 1 ;
- l'état final de la machine, de degré sortant nul ;

qui correspondent respectivement au début et à la fin des calculs.

Les autres sommets sont de deux types :
- les sommets d'incrément sont de degré sortant 1. Ils correspondent à l'incrément du registre ;
- les sommets de décrément sont de degré sortant 2. Ils correspondent au décrément du registre.

En entrée, les entiers $x_1, \ldots, x_n$ sont placés dans les registres $R_1, \ldots, R_n$. Le résultat du calcul est par convention placé dans le registre $R_1$. Chaque sommet du graphe correspond à un état de la machine et le programme fait passer la machine dans l'état indiqué par l'arête après avoir effectué l'opération associée au sommet.

Certaines arêtes sont étiquetées par 0. Ces arêtes partent nécessairement d'un sommet de décrément. De tout sommet de décrément part une et une seule arête étiquetée par 0.

Dans le cas d'une opération de décrément, la machine passe dans l'état indiqué par l'arête étiquetée par 0 si et seulement si le registre est vide avant l'opération de décrément.

Les machines à registres sont séquentielles et déterministes, car le programme ne peut exécuter qu'au plus une opération de décrément et qu'il n'a qu'un seul choix possible.

## E.3 Fonctions calculables par une machine à registre

### Fonction calculable

Les machines à registres sont séquentielles et déterministes. Les calculs sont tous fonctionnels.

Soit $\mathcal{F}_*$ l'ensemble des fonctions de la forme $f : \mathbb{N}^k \longmapsto \mathbb{N}$, pour un entier $k$ quelconque. On note $\mathcal{F}_k$ l'ensemble des fonctions de $\mathcal{F}_*$ qui sont d'arité $k$. Ainsi,

$$\mathcal{F}_* = \bigcup_{k \in \mathbb{N}} \mathcal{F}_k$$

Une fonction $f \in \mathcal{F}_k$ est dite *calculable* par une machine à registres s'il existe une machine à registres $\mathcal{R}$ telle que, quelque soit $x \in \mathbb{N}^k$, $f(x) = \mathcal{R}(x)$.

Cette égalité recouvre deux cas :
- soit les deux membres sont définis et dans ce cas ils sont égaux ;
- soit les deux membres sont indéfinis.

### Fonction constante

La fonction $f(x) = 0$ est calculable par une machine à registre.

### Fonction successeur

La fonction $f(x) = x + 1$ est calculable par une machine à registre.

### Composition de fonctions calculables

Si la fonction $h(y) \in \mathcal{F}_m$ et si les $m$ fonctions $g_1(x), \ldots, g_m(x)$ de $\mathcal{F}_k$ sont calculables par une machine à registres alors la fonction *composée* $h(g_1(x), \ldots, g_m(x))$ est calculable par une machine à registre.

### Récursion primitive

Si les fonctions $g(x) \in \mathcal{F}_k$ et $h(y, z, x) \in \mathcal{F}_{k+2}$ sont calculables par une machine à registres, alors la fonction $f \in \mathcal{F}_{k+1}$ définie par $f(0, x) = g(x)$ et $f(y + 1, x) = h(y, f(y, x), x)$ est calculable par une machine à registre.

### Opérateur de minimisation

Soit $f \in \mathcal{F}_{k+1}$. Soit $x \in \mathbb{N}^k$, on définit l'ensemble éventuellement vide $\mathcal{A}(x)$ des $y \in \mathbb{N}$ tel que :

$$\begin{cases} f(y, x) = 0 \text{ et} \\ \forall z \leq y, f(z, x) \text{ est défini} \end{cases}$$

On définit la fonction $g(x)$ par :

$$g(x) = \begin{cases} \min\limits_{y \in \mathcal{A}(x)} y \text{ si } \mathcal{A}(x) \text{ est non vide ;} \\ \uparrow \text{ sinon} \end{cases}$$

Si la fonction $f(y, x)$ est calculable par une machine à registres, alors la fonction $g(x)$ est calculable par une machine à registre.

### Fonctions élémentaires

L'ensemble des fonctions élémentaires est la plus petite partie $\mathcal{F}_*^E$ de $\mathcal{F}_*$ qui vérifie les propriétés suivantes :
- La fonction constante $f(x) = 0$ appartient à $\mathcal{F}_*^E$.
- La fonction successeur $f(x) = x + 1$ appartient à $\mathcal{F}_*^E$.
- $\mathcal{F}_*^E$ est fermée pour la composition.

La classe des fonctions élémentaires correspond aux fonctions que l'on peut obtenir à partir des opérations arithmétiques simples. Ces fonctions sont partout définies et correspondent aux fonctions usuelles.

### Fonctions primitives récursives

L'ensemble des fonctions primitives récursives est la plus petite partie $\mathcal{F}_*^{PR}$ de $\mathcal{F}_*$ qui vérifie les propriétés suivantes :

- La fonction constante $f(x) = 0$ appartient à $\mathcal{F}_*^{PR}$.
- La fonction successeur $f(x) = x + 1$ appartient à $\mathcal{F}_*^{PR}$.
- $\mathcal{F}_*^{PR}$ est fermée pour la composition.
- $\mathcal{F}_*^{PR}$ est fermée par récursion primitive.

La classe des fonctions prmitives récursives est l'extension de la classe des fonctions élémentaires quand on utilise aussi la récursion primitive.

**Remarque** : intuitivement, on voit qu'une fonction primitive récursive est programmable en utilisant des boucles `for` et des appels non récursifs de fonctions. Ainsi, la fonction de récursion primitive peut être calculée par le programme suivant :

**En entrée** $(y, x)$
$r \leftarrow g(x)$
**Pour** $i$ **de** $0$ **à** $y - 1$ **faire**
  $r \leftarrow h(i, r, x)$
**Fin Pour**
**Retourner** $r$

Algorithme 6: Programmation de la récursion primitive

### Fonctions récursives

L'ensemble des fonctions récursives est la plus petite partie $\mathcal{F}_*^{R}$ de $\mathcal{F}_*$ qui vérifie les propriétés suivantes :

- La fonction constante $f(x) = 0$ appartient à $\mathcal{F}_*^{R}$.
- La fonction successeur $f(x) = x + 1$ appartient à $\mathcal{F}_*^{R}$.
- $\mathcal{F}_*^{R}$ est fermée pour la composition.
- $\mathcal{F}_*^{R}$ est fermée par récursion primitive.
- $\mathcal{F}_*^{R}$ est fermée par minimisation.

La classe des fonctions recursives est l'extension de la classe des fonctions primitives récursives quand on permet également l'emploi de l'opérateur de minimisation.

**Remarque** : intuitivement, on voit que pour pouvoir programmer les fonctions récursives totales, il faut utiliser des boucles `while`. Ainsi, la fonction de minimisation $\mu y.(f(y, x) = 0)$ (le plus petit $y$ tel que $f(y, x) = 0$) peut être calculée par le programme suivant :

**En entrée** $x$
$y \leftarrow 0$
**Tant que** $(f(y, x) \neq 0)$ **faire**
  $y \leftarrow y + 1$
**Fait**
**Retourner** $y$

Algorithme 7: Programmation de la minimisation non bornée

## E.4 Les machines de Turing

Les machines de Turing ont été introduites par Alan Mathison Turing en 1936.

Il y a plusieurs sortes de machines de Turing, qui sont toutes équivalentes entre elles dans le sens que toute machine d'une sorte donnée peut être simulée par une machine d'une autre sorte.

Soit $\mathcal{A}$ un alphabet. On définit un symbole spécial, le blanc noté $\square$. On note $\mathcal{A}^*$ l'ensemble des mots formés à partir de cet alphabet et $\widetilde{\mathcal{A}}$ l'ensemble $\mathcal{A} \cup \square$.

Une machine de Turing est composée de trois parties :
- Une bande découpée en cellules. Chacune de ces cellules peut contenir un caractère pris dans l'alphabet $\widetilde{\mathcal{A}}$. La bande est infinie dans les deux sens. Elle ne contient qu'un nombre fini de caractères non blancs.
- Une tête de lecture-écriture qui est toujours placée sur une cellule et qui peut remplacer le symbole de la cellule courante par un autre symbole de l'alphabet $\widetilde{\mathcal{A}}$, en se déplacant éventuellement d'une cellule vers la gauche ou vers la droite.
- Une unité centrale qui contrôle le comportement de la machine. À tout moment, le comportement dépend de l'état interne de la machine, ces états sont en nombre fini et notés à l'aide des éléments d'un ensemble fini $\mathcal{Q}$.

**Table des transitions**

Le comportement de la machine dépend alors de la table des transitions qui est un ensemble fini de quintuplets de la forme

$$(q, a, b, D, q') \text{ où } q, q' \in \mathcal{Q}, a, b \in \mathcal{A} \text{ et } D = R, L, N$$

Si la machine est dans l'état $q$ et si le symbole sous la tête de lecture-écriture est $a$, la tête de lecture-écriture écrit $b$ dans la cellule, puis se déplace à droite si $D = R$, à gauche si $D = L$ ou ne bouge pas si $D = N$. Enfin la machine passe dans l'état $q'$.

Si la machine est dans l'état $q$ avec comme symbole lu $a$ et qu'il n'y a pas d'instructions commençant par $(q, a)$, la machine s'arrête de fonctionner. Un tel état est appelé un état d'arrêt.

La table des transitions est donc le programme qu'exécute la machine. L'état initial de la machine est noté $q_0$. On définit un état particulier, appelé état acceptant et noté $q_{acc}$, qui marque un état d'arrêt de la machine correspondant à une acceptation des données initiales de la bande. Un état d'arrêt qui n'est pas acceptant est dit rejettant.

On suppose qu'au début des calculs, la tête de lecture est placée sur la lettre de l'alphabet $\mathcal{A}$ la plus à gauche de la bande.

**Configuration instantanée** On appelle configuration instantanée d'une machine de Turing tout triplet $(w_1, q, w_2)$ où $w_1$ et $w_2$ sont les mots de $\widetilde{\mathcal{A}}^*$ tels que
- $w_1 w_2$ est le contenu de la bande de la machine de Turing. Autrement dit il n'y a que des symboles blancs à gauche et à droite de ce mot ;
- le premier symbole du mot $w_1$ et le dernier symbole du mot $w_2$ sont différents du blanc ;
- dans l'état $q$ de la machine, le mot $w_1$ est le mot strictement à gauche de la position de la tête de lecture sur la bande et le mot $w_2$ est le mot à droite de la position de la tête de lecture. La tête est donc placée sur le premier symbole de $w_2$ si ce mot n'est pas vide. Si $w_2$ est vide, la tête de lecture est en fait sur un blanc.

Le mot $w_1 w_2$ est *la partie active de la bande.*

Notons $\underset{T}{\vdash}$ l'action de la machine $T$ et $\overset{*}{\underset{T}{\vdash}}$ la cloture transitive de cette relation. Considérons la transition $(q_1, a, b, D, q_2)$.
- si $D = R$, $(w_1, q_1, aw_2) \underset{T}{\vdash} (w_1 b, q_1, w_2)$ ;
- si $D = L$, $(w_1' c, q_1, aw_2) \underset{T}{\vdash} (w_1', q_1, cbw_2)$ ;
- si $D = N$, $(w_1, q_1, aw_2) \underset{T}{\vdash} (w_1, q_1, bw_2)$.

On dit que la machine de Turing $T$ calcule le mot $w'$ à partir du mot $w$, si :

$$(\varnothing, q_0, w_1) \overset{*}{\underset{T}{\vdash}} (w_1, q', w_2) \text{ où } w' = w_1 w_2 \text{ et } q' \text{ est un état d'arrêt.}$$

Ce qui signifie qu'il existe trois suites finies $u = (u_i)_{0 \leq i \leq n}, v = (v_i)_{0 \leq i \leq n}, q = (q_i)_{0 \leq i \leq n}$, où $u_i$ et $v_i$ sont des mots de $\widetilde{\mathcal{A}}^*$ et $q_i$ des symboles de l'alphabet $\mathcal{Q}$, telles que

$$(u_0, q_0, v_0) \underset{T}{\vdash} (u_1, q_1, v_1) \underset{T}{\vdash} \ldots \underset{T}{\vdash} (u_i, q_i, v_i) \underset{T}{\vdash} (u_{i+1}, q_{i+1}, v_{i+1}) \underset{T}{\vdash} \ldots \underset{T}{\vdash} (u_n, q_n, v_n)$$

où $u_0 = \varnothing$, $v_0 = w$, $u_n v_n = w'$ et $q'$ est un état d'arrêt.

**Définition formelle**

**Définition E.4.1 (Machine de Turing déterministe)** *Une machine de Turing (déterministe) $\mathcal{T}$ est un quintulet $(\mathcal{Q}, \mathcal{A}, \square, \delta, q_0)$ où*
- *$\mathcal{Q}$ et $\mathcal{A}$ sont des ensembles finis disjoints et $\square \notin \mathcal{A} \sqcup \mathcal{Q}$ ;*
- *$\delta$ est une fonction de $\mathcal{Q} \times \widetilde{\mathcal{A}}$ dans $\widetilde{\mathcal{A}} \times \{R, L, N\} \times \mathcal{Q}$.*

**Définition E.4.2 (Machine de Turing non déterministe)** *Une machine de Turing non déterministe $\mathcal{T}$ est un quintulet $(\mathcal{Q}, \mathcal{A}, \square, \delta, q_0)$ où*
- *$\mathcal{Q}$ et $\mathcal{A}$ sont des ensembles finis disjoints et $\square \notin \mathcal{A} \sqcup \mathcal{Q}$ ;*
- *$\delta$ est une partie quelconque de $\mathcal{Q} \times \widetilde{\mathcal{A}} \times \widetilde{\mathcal{A}} \times \{R, L, N\} \times \mathcal{Q}$.*

**Définition E.4.3 (Degré d'indétermination d'une machine de Turing non déterministe)** *Le degré d'indétermination d'une machine de Turing non déterministe est le nombre maximal de transitions possibles sur une même lettre et un même état.*

**Théorème E.4.1** *(Simulation d'une machine de Turing non déterministe). Toute machine de Turing non déterministe $\mathcal{T}$ peut être simulée par une machine de Turing déterministe $\mathcal{T}'$. Si la machine $\mathcal{T}$ accepte une entrée en $n$ transitions, la machine $\mathcal{T}'$ accepte cette entrée en $O(n^2 r^{2n})$ transitions, où $r$ est le degré d'indétermination de $\mathcal{T}$. La machine $\mathcal{T}'$ n'utilise qu'au plus $O(n)$ cellules supplémentaires.*

*En d'autres termes, la simulation d'une machine de Turing non déterministe par une machine de Turing déterministe se fait avec un accroissement linéaire de l'espace utilisé.*

**Définition E.4.4 (Machine de Turing multibande)** *Une machine de Turing multibande est une machine de Turing à laquelle on a ajouté des bandes ayant chacune leur propre tête de lecture-écriture. À chaque transition, la machine change d'état, écrit un symbole sur chacune des pistes et déplace éventuellement chacune des têtes. Au démarrage, seule la première bande contient effectivement des données. Une transition a donc la forme :*

$$(q, a_1, b_1, D_1, a_2, b_2, D_2 \ldots, a_k, b_k, D_k, q') \text{ où } D_i \in \{R, L, N\},$$

*qui signifie que si la tête 1 lit $a_1$, ..., si la tête $k$ lit $a_k$, alors $\forall i$, $1 \leq i \leq k$, la tête $i$ écrit $b_i$ et effectue le mouvement $D_i$.*

**Théorème E.4.2 (Simulation d'une machine de Turing multibande)** *Toute machine de Turing multibande $\mathcal{T}$ peut être simulée par une machine de Turing à une bande $\mathcal{T}'$. Si la machine $\mathcal{T}$ accepte une entrée en $n$ transitions, la machine $\mathcal{T}'$ accepte cette entrée en $O(n^2)$ transitions. Le nombre de cellules visitées par la nouvelle machine est égal au maximum des cellules visitées sur chacune des bandes de la machine $\mathcal{T}$.*

*En d'autres termes, la simulation d'une machine de Turing multibande par une machine de Turing ayant une seule bande se fait avec un accroissement quadratique du temps et dans un espace identique.*

**Définition E.4.5 (Machine de Turing avec oracle)** *Une machine de Turing avec oracle $\mathcal{T}^O$ est une machine de Turing déterministe ou non déterministe qui possède une bande spéciale, dite de requête, et deux états spéciaux $q_q$ et $q_r$. La machine $\mathcal{T}^O$ peut faire une requête à l'oracle de la manière suivante :*

- *elle écrit un mot $x$ sur la bande de requête et place la tête de lecture après ce mot ;*
- *elle passe dans l'état $q_q$ ;*
- *l'oracle efface la bande de requête, écrit un mot $y$ sur la bande de requête, place la tête de lecture sur le premier caractère de ce mot et fait passer la machine dans l'état $q_r$.*

*Le passage de l'état $q_q$ à l'état $q_r$ compte pour une seule transition. Si l'oracle calcule une fonction $f$ de $\mathcal{A}^*$ dans $\mathcal{A}^*$ (l'alphabet $\mathcal{A}$ étant un sous-alphabet de la machine de Turing), le mot placé par l'oracle dans la bande de requête est $y = f(x)$. La réalisation de la machine avec un oracle calculant la fonction $f$ est notée $\mathcal{T}^f$.*

**Définition E.4.6 (Machine de Turing avec oracle ensembliste)** *Une machine de Turing avec oracle ensembliste $\mathcal{T}^O$ est une machine de Turing déterministe ou non déterministe qui possède une bande de requête et trois états spéciaux $q_q$, $q_o$ et $q_n$. La machine $\mathcal{T}^O$ peut faire une requête à l'oracle de la manière suivante :*

- *elle écrit un mot $x$ sur la bande de requête et place la tête de lecture après ce mot ;*
- *elle passe dans l'état $q_q$ ;*
- *l'oracle fait passer la machine soit dans l'état $q_o$, soit dans l'état $q_n$ et vide la bande de requête.*

*Si la fonction calculée par l'oracle est la fonction caractéristique de l'ensemble $A$, on note $\mathcal{T}^A$ la machine de Turing avec oracle ensembliste lorsqu'elle est munie d'un oracle reconnaissant le langage $A$.*

**Définition E.4.7 (Machine de Turing probabliste)** *Une machine de Turing probabiliste $\mathcal{T}$ est une machine de Turing déterministe munie d'une bande supplémentaire appelée bande probabiliste et d'un générateur aléatoire de bits qui fournit des bits 0 ou 1 avec probabilité 1/2. Le fonctionnement de $\mathcal{T}$ est le suivant :*

- *sur l'entrée $x$, le générateur aléatoire écrit un bit sur la bande probabiliste ;*
- *si la machine peut exécuter une transition, elle l'effectue en déplaçant la tête de lecture sur la bande probabiliste d'une position vers la droite.*

*La configuration instantanée d'une machine de Turing probabiliste comporte le mot $\sigma \in \{0,1\}^*$ contenu sur la bande aléatoire, la tête de lecture étant placée après ce mot. La longueur de ce mot donne le nombre de transitions déjà effectuées. Dans une configuration terminale, la longeur du mot $|\sigma|$ est égale au temps de calcul.*

Une machine de Turing probabiliste $\mathcal{T}$ n'est donc pas très différente d'une machine de Turing non déterministe pour laquelle chaque transition est effectuée avec un degré d'incertitude $r = 2$. En fait, une machine de Turing probabiliste comporte deux fonctions de transition $\delta_0$ et $\delta_1$. À chaque étape du calcul, $\mathcal{T}$ choisit de manière aléatoire quelle fonction de transition appliquer, avec probabilité 1/2 d'appliquer la fonction de transition $\delta_0$ et probabilité 1/2 d'appliquer la fonction de transition $\delta_1$.

Nous dirons qu'une machine de Turing probabiliste $\mathcal{T}$ décide un langage $L$ en temps $T(n)$ si $\forall x \in \{0,1\}^*$, $\mathcal{T}$ s'arrête en $T(|x|)$ étapes, quelques soient les choix aléatoires effectués lors du calcul et si $p[\mathcal{T}(x) = L(x)] \geq 2/3$ ($L(x) = 1$ si $x \in L$, 0 sinon).

**Probabilité qu'une machine de Turing probabiliste s'arrête** Pour définir la probabilité qu'une machine de Turing probabiliste s'arrête, il faut considérer les suites de bits $\sigma \in \{0,1\}^\infty$ générées par le générateur aléatoire. On considère l'espace de probabilité $\{0,1\}^\infty$, muni d'une distribution uniforme. Un évenement $A$ de $\{0,1\}^\infty$ est mesurable si et seulement s'il existe un borélien $B$ de $[0,1]$ tel que $A = \varphi^{-1}(B)$, où $\varphi : \{0,1\}^\infty \to [0,1]$ est l'application surjective qui à un mot $\sigma \in \{0,1\}^\infty$ associe le nombre réel de $[0,1]$ dont développement en base 2 est $0,\sigma$ (cette application n'est pas injective car par exemple $0,1$ et $0,0111\ldots$ codent le même réel 1/2). Dans ce cas, $p(A) = \int_B dx$. Soit $L \subseteq \{0,1\}^*$. Quelle est la probabilité $p(L)$ de $L$ sur $\{0,1\}^\infty$ ? Soit $\sigma \in \{0,1\}^*$. On lui associe l'ensemble $A_\sigma$ de tous les mots $\sigma' \in \{0,1\}^\infty$ ayant $\sigma$ pour préfixe. On a :

$$\varphi(\sigma') = \sum_{n=1}^{|\sigma|} \varepsilon_n 2^{-n} + \sum_{n \geq |\sigma|+1} \varepsilon_n 2^{-n} \leq \varphi(\sigma) + \frac{1}{2^{|\sigma|}}.$$

Donc :

$$A_\sigma = \varphi^{-1}([0,\sigma;\, 0,\sigma + \frac{1}{2^{|\sigma|}}]).$$

On peut donc définir la probabilité :

$$p(A_\sigma) = \int_{[0,\sigma;\, 0,\sigma+\frac{1}{2^{|\sigma|}}]} dx = \frac{1}{2^{|\sigma|}}.$$

On en déduit que :

$$\bigcup_{\sigma\in L} A_\sigma = \varphi^{-1}(\bigcup_{\sigma\in L}[0,\sigma;\, 0,\sigma+\frac{1}{2^{|\sigma|}}])$$

admet une probabilité comme image inverse d'une union au plus dénombrable d'intervalles :

$$p(\bigcup_{\sigma\in L} A_\sigma)) = \int_{\bigcup_{\sigma\in L}[0,\sigma;\, 0,\sigma+\frac{1}{2^{|\sigma|}}]} dx.$$

On peut évaluer cette probabilité facilement dans le cas où $L_n \subseteq \{0,1\}^*$ est formé de mots ayant tous la même longueur $n$. En effet, dans ce cas, les intervalles $[0,\sigma;\, 0,\sigma+\frac{1}{2^n}[$ sont deux à deux disjoints donc :

$$p(\bigcup_{\sigma\in L_n} A_\sigma)) = \sum_{\sigma\in L_n}\int_{[0,\sigma;\, 0,\sigma+\frac{1}{2^n}]} dx = \frac{1}{2^n}.\sum_{\sigma\in L_n} = \frac{|L_n|}{2^n}$$

Soit $L \subseteq \{0,1\}^*$, $L = \bigsqcup_{n\geq 0} L_n$ ($L_n$ désigne l'ensemble des mots de longueur $n$ de $L$) donc :

$$p(\bigcup_{\sigma\in L} A_\sigma)) = \sum_{n\geq 0} p(\bigcup_{\sigma\in L_n} A_\sigma)) = \sum_{n\geq 0}\frac{|L_n|}{2^n}$$

D'où puisque les ensembles $(\bigcup_{\sigma\in L_n} A_\sigma,\, n\geq 0)$ sont disjoints deux à deux :

$$p(\bigcup_{\sigma\in L} A_\sigma) = \sum_{\sigma\in L}\frac{1}{2^{|\sigma|}}.$$

Soit $\sigma$ et $\sigma'$ deux mots distincts de la bande aléatoire tels que la machine de Turing probabiliste s'arrête sur l'entrée $x$, alors on est certain que le mot $\sigma$ n'est pas préfixe de $\sigma'$ car la machine de Turing est déterministe. Si $L_A(x)$, $L_T(x)$ et $L_R(x)$ désignent respectivement l'ensemble des mots de $\{0,1\}^\infty$ qui sont acceptant, terminaux et rejettants par la machine de Turing probabiliste sur l'entrée $x$, on a : $p(L_T(x)) = p(L_A(x)) + p(L_R(x))$ et :

$$p(L_A(x)) = \sum_{\sigma\in L_A(x)} = \frac{1}{2^{|\sigma|}}$$

$$p(L_R(x)) = \sum_{\sigma\in L_R(x)} = \frac{1}{2^{|\sigma|}}$$

$$p(L_T(x)) = \sum_{\sigma\in L_T(x)} = \frac{1}{2^{|\sigma|}}$$

Le temps de calcul sur l'entrée $x$ est infini si $p(L_T(x)) < 1$. Si $p(L_T(x)) = 1$, le temps de calcul sur l'entrée $x$ est défini par :

$$t(x) = max(|\sigma|, \sigma \in L_T(x)).$$

Le temps de calcul sur une entrée de longueur $n$ est donné par :

$$\mathbf{time}(n) = max(t(x),\, |x| = n).$$

On peut également définir le temps de calcul moyen sur l'entrée $x$ qui est infini si $p(L_T) < 1$. Si $p(L_T(x)) = 1$, le temps de calcul moyen sur l'entrée $x$ est définit par :

$$E[t(x)] = \sum_{\sigma\in L_T(x)} = \frac{|\sigma|}{2^{|\sigma|}}.$$

La moyenne des temps de calcul sur les entrées ne peut être défini que s'il on connaît la distribution des mots d'entrées. On peut cependant toujours définir le temps de calcul espéré par :

$$\mathbf{expect_time}(n) = max(E[t(x)],\, |x| = n).$$

### Temps et espace de calcul

On suppose que les données sont stockées en binaire sur la bande. Soit $x \in \mathbb{N}$. $x = \sum_{i=0}^{n-1} \varepsilon_i 2^i$, avec $\varepsilon_i \in \{0,1\}$. On a : $x \leq \sum_{i=0}^{n-1} 2^i = 2^n - 1 \leq 2^n$. Donc $n \geq log_2 x$. D'où $|x| = \lfloor log_2 x \rfloor + 1$ est une taille suffisante pour stocker $x$. En convenant que $\lfloor log_2(0) \rfloor = 0$, et en notant $|x|$ le nombre de symbole nécessaire pour stocker $x \in \mathbb{N}$ en binaire sur la bande, on a donc : $|x| = \lfloor log_2 x \rfloor + 1$. Pour stocker un élément $x = (x_1, \ldots, x_k) \in \mathbb{N}^k$ en binaire sur la bande, le nombre de symboles nécessaires est donc :

$$|x| = 2k - 1 + \sum_{i=1}^{k} \lfloor log_2 x_i \rfloor.$$

**Définition E.4.8 (Temps de calcul)** *Le temps de calcul d'une machine de Turing $\mathcal{T}$ sur l'entrée $w$, noté $t(w)$, est le nombre minimum de transitions effectuées lors d'un calcul fini de la machine. Si aucun calcul sur l'entrée $w$ n'est fini, on considère que le temps de calcul est infini.*
*Le temps de calcul de $\mathcal{T}$ sur les entrées de longueur $n$, noté $\mathbf{t}_{\mathcal{T}}(n)$, est le maximum des temps de calcul sur l'ensemble des entrées de longueur $n$.*

$$\mathbf{t}_{\mathcal{T}}(n) = \max_{|w|=n} t(w)$$

**Définition E.4.9 (Espace de calcul)** *L'espace de calcul d'une machine de Turing $\mathcal{T}$ sur l'entrée $w$, noté $s(w)$, est l'espace minimum visité sur les bandes de travail lors d'un calcul fini de la machine. Si aucun calcul sur l'entrée $w$ n'est fini, on considère que l'espace de calcul est infini.*
*L'espace de calcul de $\mathcal{T}$ sur les entrées de longueur $n$, noté $\mathbf{s}_{\mathcal{T}}(n)$, est le maximum des espaces de calcul sur l'ensemble des entrées de longueur $n$.*

$$\mathbf{s}_{\mathcal{T}}(n) = \max_{|w|=n} s(w)$$

**Définition E.4.10** *Soient $t$ et $s$ deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. On dit qu'une machine de Turing calcule en temps $t(n)$ et en espace $s(n)$ si, pour toute entrée de longueur $n$ pour laquelle elle s'arrête, elle effectue au plus $max(n+1, \lceil t(n) \rceil)$ transitions et elle visite au plus $max(1, \lceil s(n) \rceil)$ cellules* [1].

**Théorème E.4.3** *(Compression de la bande). Soit $c > 0$ un nombre réel. Pour toute machine de Turing $\mathcal{T}$ calculant en temps $t(n)$ et en espace $s(n)$, il existe une machine de Turing $\mathcal{T}'$ qui la simule en temps $t(n) + 1$ et en espace $c.s(n)$.*

## E.5 Codage d'une machine de Turing

### Système d'appariement

Le triplet $(f, g, h)$ est un *système d'appariement* si :
- $f$ est une fonction de $\mathbb{N}^2$ dan $\mathbb{N}$ ;
- $g$ et $h$ sont des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ ;
- pour tout $(x, y) \in \mathbb{N}^2$, $g(f(x, y)) = x$ et $h(f(x, y)) = y$ ;
- pour tout $x \in \mathbb{N}$, $f(h(x), g(x)) = x$ ;
- pour tout $x \in \mathbb{N}$, $g(x) \leq x$ et $h(x) \leq x$.

Dans ces conditions, $f$ définit une bijection de $\mathbb{N}^2$ sur $\mathbb{N}$ et $f^{-1} = g \times h$.

Considérons le système consistant à énumérer $\mathbb{N} \times \mathbb{N}$ en utilisant la relation d'ordre total sur les couples de $\mathbb{N}^2$ définie par :

$$(x, y) \preceq (x', y') \text{ ssi } (x - x') + (y - y') < 0 \text{ ou } ((x - x') + (y - y') = 0 \text{ et } x \leq x')$$

[1] On suppose que toute machine lit toujours intégralement son entrée, effectue au moins une transition et visite toujours au moins un symbole sur ses bandes de travail.

qui se traduit graphiquement lorsque $x > x'$ par le fait que $(x', y')$ reste cependant au dessous de la doite passant par $(x', y')$ de pente $-1$.

On définit $f(x, y)$ comme étant le nombre de couple de $\mathbb{N} \times \mathbb{N}$ strictement plus petits que $(x, y)$ :

$$f(x, y) = \frac{(x + y)(x + y + 1)}{2} + x$$

Soit $(f, g, h)$ un système d'appariement. Pour tout entier $k > 0$, on définit par récurrence les fonctions $f_k : \mathbb{N}^k \longrightarrow \mathbb{N}$ par :

$$\begin{cases} f_1(x) = x \\ f_{k+1}(x_1, \ldots, x_k, x_{k+1}) = f(f_k(x_1, \ldots, x_k), x_{k+1}) \end{cases}$$

Pour tout entier $k > 0$, $f_k$ est une bijection de $\mathbb{N}^k$ sur $\mathbb{N}$.

Pour toute suite d'entiers $x = (x_1, \ldots, x_k)$, on note :

$$\|x\| = f_{k+1}(x, k)$$

**Notation** :
On considère le système d'appariement $(\langle ., . \rangle, \pi_1, \pi_2)$
- $\langle ., . \rangle$ est une fonction de $\mathbb{N}^2$ dan $\mathbb{N}$ ;
- $\pi_1$ et $\pi_2$ sont des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ ;
- pour tout $(x, y) \in \mathbb{N}^2$, $\pi_1(\langle x, y \rangle) = x$ et $\pi_2(\langle x, y \rangle) = y$ ;
- pour tout $x \in \mathbb{N}$, $\langle h(x), g(x) \rangle = x$ ;
- pour tout $x \in \mathbb{N}$, $\pi_1(x) \leq x$ et $\pi_2(x) \leq x$.

$\langle x, y \rangle = \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$ est une fonction primitive récursive. On note : $\langle x_1, x_2, x_3 \rangle = \langle \langle x_1, x_2 \rangle, x_3 \rangle$ et de manière générale $\langle x_1, x_2, \ldots, x_n \rangle = \langle \ldots \langle \langle x_1, x_2 \rangle, x_3 \rangle, \ldots, x_n \rangle$.

### Nombre de Gödel

Les alphabets $\mathcal{A}$ et $\mathcal{Q}$ sont des ensembles finis et on peut supposer que ces ensembles sont totalement ordonnés de manière arbitraire.

On choisit le codage suivant, utilisant les nombres impaires :

| $R$ | $L$ | $N$ | $\square$ | $q_0$ | $a_0$ | $q_1$ | $a_1$ | $\ldots$ | $q_i$ | $a_i$ | $\ldots$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | $\ldots$ | $9 + 4i$ | $11 + 4i$ | $\ldots$ |

Exemple : calcul de $\|(a_2, q_1, a_3, a_1)\| = \|(19, 13, 23, 15)\| = f_5(19, 13, 23, 15, 4)$

```
calc <<END
x=19
y=13
z=(x+y)*(x+y+1)/2+x
x=z
y=23
z=(x+y)*(x+y+1)/2+x
x=z
y=15
z=(x+y)*(x+y+1)/2+x
x=z
y=4
z=(x+y)*(x+y+1)/2+x
z
END
88887111659994600815
```

À une transition $\tau = (q, a, b, D, q')$ d'une machine de Turing, on associe le nombre $\|\tau\| = \|(q, a, b, D, q')\|$.

Exemple : calcul de $\|\tau\| = \|(q_0, \square, \square, N, q_1)\| = \|(9, 7, 7, 5, 13)\| = f_6(9, 7, 7, 5, 13, 5)$

```
calc <<END
x=9
y=7
z=(x+y)*(x+y+1)/2+x
x=z
y=7
z=(x+y)*(x+y+1)/2+x
x=z
y=5
z=(x+y)*(x+y+1)/2+x
x=z
y=13
z=(x+y)*(x+y+1)/2+x
x=z
y=5
z=(x+y)*(x+y+1)/2+x
z
quit
END
289604357487500691777566058503 5
```

On prolonge cette représentation aux machines de Turing. Si $\mathcal{T}$ est une machine de Turing et si on choisit un ordre quelconque pour l'ensemble de ses transitions, par exemple : $\{\tau_1, \ldots, \tau_m\}$, le nombre $\|\mathcal{T}\| = \|(\tau_1, \ldots, \tau_m)\| = \|(\|\tau_1\| \ldots, \|\tau_m\|)\|$ est appelé un nombre de Gödel de la machine $\mathcal{T}$. La machine $\mathcal{T}$ admet donc $m!$ nombres de Gödel distincts.

Exemple : Considérons la machine de Turing $\mathcal{T}$ possédant l'unique transition $\|\tau\| = \|(q_0, \square, \square, N, q_1)\|$. Le nombre de Gödel de $\mathcal{T}$ vaut $\|\mathcal{T}\| = \|(\|\tau\|)\| = f_2(\|\tau\|, 1)$

```
calc <<END
x=9
y=7
z=(x+y)*(x+y+1)/2+x
x=z
y=7
z=(x+y)*(x+y+1)/2+x
x=z
y=5
z=(x+y)*(x+y+1)/2+x
x=z
y=13
z=(x+y)*(x+y+1)/2+x
x=z
y=5
z=(x+y)*(x+y+1)/2+x
x=z
y=1
z=(x+y)*(x+y+1)/2+x
z
quit
END
41935341937874048998125667639805632459779038759778583844382 01
```

## E.6 Décidabilité d'un problème de décision

**Calculabilité d'un prédicat**

**Définition E.6.1** *Soit $P(x)$ un prédicat sur $\mathbb{N}^k$. On dit que $P$ est calculable par une machine à registre si la fonction caractéristique de $P$ est calculable par une machine à registre. La fonction $1_P$ est partout définie.*

$$1_P(x) = \begin{cases} 1 \text{ si } P(x) \\ 0 \text{ sinon} \end{cases}$$

*On a donc : $\exists \mathcal{R} \forall x \, 1_P(x) = \mathcal{R}(x)$ et $\mathcal{R}(x) \downarrow$.*

**Définition E.6.2 (Prédicat partiellement calculable par une machine à registre)** *Soit $P(x)$ un prédicat sur $\mathbb{N}^k$. On dit que $P$ est partiellement calculable par une machine à registre si la fonction caractéristique partielle de $P$ est calculable par une machine à registre. La fonction partielle $1_P$ est définie par :*

$$1_P(x) = \begin{cases} 1 \text{ si } P(x) \\ \uparrow \text{ sinon} \end{cases}$$

*On a donc : $\exists \mathcal{R} \forall x \, 1_P(x) = \mathcal{R}(x)$ et $\mathcal{R}(x) \downarrow$ ou $\mathcal{R}(x) \uparrow$.*

**Définition E.6.3 (Problème de décision)** *On appelle problème de décision tout problème pour lequel il existe un prédicat $P(x)$ sur $\mathcal{A}^*$ tel que le problème consiste à répondre oui ou non à la question "est-ce que $P(x)$ est vrai ?". Résoudre effectivement un problème de décision revient à montrer que la fonction caractéristique du prédicat $P$ est effectivement calculable.*

Un problème de décision $\Pi$ peut être caractérisé par la donnée d'un langage et d'un prédicat :

1. le langage de ses instances qui est une partie $L(\Pi)$ de $A^*$, pour un alphabet $A$ convenablement choisi ;
2. un prédicat $P_\Pi$ défini sur $L(\Pi)$.

**Définition E.6.4 (Langage associé à un problème de décision)** *Pour caractériser les langages, on peut les définir comme représentant des problèmes de décision. On définit un langage $L$ comme étant l'ensemble des réponses positives à la question $w \in L$ ?*

**Définition E.6.5 (Décidabilité d'un problème de décision.)** *Soit $\Pi$ un problème de décision et $P_\Pi$ un prédicat défini sur le language $L(\Pi)$ des instances du problème $P_\Pi$. Un problème est décidable si les ensembles : $\{x | \, P_\Pi(x)\}$ et $\{x | \, \neg P_\Pi(x)\}$ sont calculables. Cela revient à dire que la fonction caractéristique $1_{P_\Pi}$ du prédicat $P_\Pi$ est récursive totale (pré-récursive).*

**Définition E.6.6 (Problème de décision partiellement décidable)** *Soit $\Pi$ un problème de décision et $P_\Pi$ un prédicat défini sur le language $L(\Pi)$ des instances du problème $P_\Pi$. Un problème est partiellement décidable si seul l'ensemble : $\{x | \, P_\Pi(x)\}$ est calculable. Cela revient à dire que la fonction caractéristique partielle $1_{P_\Pi}$ du prédicat $P_\Pi$ est récursive.*

**Définition E.6.7 (Problème de décision indécidable)** *Un problème de décision $\Pi$ est indécidable s'il n'est pas décidable.*

**Définition E.6.8** *Une partie $A \subseteq \mathbb{N}^k$ est récursivement énumérable si le prédicat $x \in A$ est partiellement décidable.*

On notera que les expressions $A$ est semi-décidable, semi-calculable, partiellement décidable et récursivement énumérable sont toutes synonymes.

Tout ensemble récursif est récursivement énumérable et un ensemble est récursif si et seulement si cet ensemble et son complémentaires sont récursivement énumérables.

L'ensemble $\mathbb{K} = \{x \, | \, x \in W_x\}$ est un ensemble récursivement énumérable qui n'est pas récursif et sont complémentaire $\mathbb{K}^c$ n'est pas récursivement énumérable.

L'ensemble $\mathbb{K}_x = \{y \, | \, y \in W_x\}$ est un ensemble récursivement énumérable qui n'est pas récursif et sont complémentaire $\mathbb{K}_x^c$ n'est pas récursivement énumérable.

## E.7 Langage accepté/reconnu par une machine de Turing

**Définition E.7.1 (Etat acceptant)** *Un état acceptant d'une machine de Turing $\mathcal{T}$ est un état d'arrêt de la machine de Turing qui correspond à une acceptation des données initiales de la bande. Un état d'arrêt qui n'est pas acceptant est dit rejettant.*

**Définition E.7.2 (Langage accepté)** *Soit $\mathcal{T}$ une machine de Turing ayant un état acceptant $q_a$. On note $L(\mathcal{T})$ le langage accepté par la machine $\mathcal{T}$, c'est-à-dire :*

$$L(\mathcal{T}) = \{w \in \mathcal{A}^* \mid \exists w_1, w_2 \in \widetilde{\mathcal{A}}^*,\ q_0 w \underset{\mathcal{T}}{\vdash} w_1 q_a w_2\}$$

*Une machine de Turing déterministe $\mathcal{T}$ accepte un langage $L$ si elle calcule la fonction caractéristique partielle de ce langage. La condition d'acceptation est donc : $w \in L$ si et seulement si la machine s'arrête.*

*Un langage $L$ est dit accepté par une machine de Turing non déterministe $\mathcal{T}$ si pour mot $w$ de $L$ il existe une suite de calculs qui se termine dans un état acceptant. On dit que la machine non déterministe $\mathcal{T}$ accepte l'entrée $x$ en $n$ transitions, si $n$ est la longueur du plus petit calcul de $\mathcal{T}$ qui accepte $x$. $n$ peut ne pas être défini pour certaines entrées.*

**Définition E.7.3 (Langage reconnu)** *Si la machine $\mathcal{T}$ s'arrête quelle que soit son entrée, le langage $L(\mathcal{T})$ est dit reconnu par la machine $\mathcal{T}$. Une machine de Turing déterministe $\mathcal{T}$ reconnaît un langage $L$ si elle calcule la fonction caractéristique totale de ce langage. La condition de reconnaissance est donc : $w \in L$ si et seulement si la machine s'arrête dans un état acceptant.*

## E.8 Classes de complexité et hiérarchie polynomiale

**Définition E.8.1 (Classes de complexité)** *Soient $t(n)$ et $s(n)$ deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. Soit $\mathcal{F}$ un ensemble de fonctions de $\mathbb{N}$ dans $\mathbb{R}$. On définit :*

1. $\mathbf{DTIME}(t(n))$ *désigne l'ensemble des langages reconnus par une machine de Turing déterministe en temps $t(n)$.*
   $\mathbf{P} = \bigcup_{k \geq 1} \mathbf{DTIME}(n^k)$ *est la classe des langages reconnus en temps déterministe polynomial. On note également* $\mathbf{P}$ *la classe des problèmes pouvant être résolus par un algorithme déterministe en temps polynomial.*
2. $\mathbf{NTIME}(t(n))$ *désigne l'ensemble des langages acceptés par une machine de Turing non déterministe en temps $t(n)$.*
   $\mathbf{NP} = \bigcup_{k \geq 1} \mathbf{NTIME}(n^k)$ *est la classe des langages acceptés en temps non déterministe polynomial. On note également* $\mathbf{NP}$ *la classe des problèmes pouvant être résolus par un algorithme non déterministe en temps polynomial.* $\mathbf{NP}$ *est aussi la classe des problèmes de décision pour lesquels pour lesquels il existe un système de preuves courtes et faciles à vérifier :* $\Pi \in \mathbf{NP}$ *si $\forall x \in \Pi$, il existe une preuve $q$, polynomiale relativement à la taille de $x$, telle que le système de preuve correspondant est dans* $\mathbf{P}$*, i.e. le système de preuve peut être évalué par un algorithme polynomial.*
3. $\mathbf{DSPACE}(s(n))$ *désigne l'ensemble des langages reconnus par une machine de Turing déterministe en espace $s(n)$.*
   $\mathbf{L} = \mathbf{DSPACE}(log_2(n))$ *est la classe des langages reconnus en espace déterministe logarithmique.*
   $\mathbf{PSPACE} = \bigcup_{k \geq 1} \mathbf{DSPACE}(n^k)$ *est la classe des langages reconnus en espace déterministe polynomial.*
4. $\mathbf{NSPACE}(s(n))$ *désigne l'ensemble des langages acceptés par une machine de Turing non déterministe en espace $s(n)$.* $\mathbf{NL} = \mathbf{NSPACE}(log_2(n))$ *est la classe des langages acceptés en espace non déterministe logarithmique.*
   $\mathbf{NSPACE} = \bigcup_{k \geq 1} \mathbf{NSPACE}(n^k)$ *est la classe des langages acceptés en espace non déterministe polynomial.*

Les machines de Turing à oracle ensembliste permettent de construire de nouvelles classes de complexité à partir des classes déjà introduites : Soit $L$ un langage et $\mathcal{C}$ une classe de langage, on note :

1. $\mathbf{P}(L)$ l'ensemble des langages qui sont reconnus par une machine de Turing déterministe munie d'un oracle pour $L$ et dont le temps de calcul est polynomial.
   On note $\mathbf{P}(\mathcal{C}) = \bigcup_{L \in \mathcal{C}} \mathbf{P}(L)$.
2. $\mathbf{NP}(L)$ l'ensemble des langages qui sont acceptés par une machine de Turing non déterministe munie d'un oracle pour $L$ et dont le temps de calcul est polynomial.
   On note $\mathbf{NP}(\mathcal{C}) = \bigcup_{L \in \mathcal{C}} \mathbf{NP}(L)$.

**Proposition E.8.1** *Soient $L$ et $L'$ deux langages, on a :*

1. $L \in \mathbf{P}(L)$
2. $\mathbf{P}(L) \subseteq \mathbf{NP}(L)$
3. $\mathbf{P}(L) = \mathbf{P}(\bar{L})$ *et* $\mathbf{NP}(L) = \mathbf{NP}(\bar{L})$
4. *Si* $L \in \mathbf{P}(L')$, *alors* $\mathbf{NP}(L) \subseteq \mathbf{NP}(L')$

**Preuve** :
$L \in \mathbf{P}(L)$ : Il s'agit de montrer qu'il existe une machine de Turing déterministe munie d'un oracle pour $L$ qui reconnait le langage $L$ en temps polynomial : $\exists \mathcal{T} \mid 1_L = \mathcal{T}^L$ et $\mathcal{T}^L$ effectue ses calculs en temps polynomial. Cela revient donc à montrer que $\leq_T^{\mathbf{P}}$ est réflexive.
$\mathbf{P}(L) \subseteq \mathbf{NP}(L)$ : une machine déterministe est un cas particulier de machine non déterministe.
$\mathbf{P}(L) = \mathbf{P}(\bar{L})$ et $\mathbf{NP}(L) = \mathbf{NP}(\bar{L})$ : on peut échanger les états $q_n$ et $q_o$ dans une machine à oracle sans modifier ni son temps de calcul, ni son comportement si on remplace l'oracle pour le langage $L$ par un oracle pour le langage $\bar{L}$.
Si $L \in \mathbf{P}(L')$, alors $\mathbf{NP}(L) \subseteq \mathbf{NP}(L')$ : Supposons qu'il existe une machine de Turing déterministe munie d'un oracle pour $L'$ et qui reconnait $L$ en temps polynomial : $1_L = \mathcal{T}_1^{L'}$ et $\mathcal{T}_1^{L'}$ effectue ses calculs en temps polynomial. Soit $L'' \in \mathbf{NP}(L)$, il existe une machine de Turing non déterministe munie d'un oracle pour $L$ qui accepte le langage $L''$ en temps polynomial : $1_{L''} = \mathcal{T}_2^L$. il s'agit de montrer que $L'' \in \mathbf{NP}(L')$, i.e. qu'il existe une machine de Turing non déterministe munie d'un oracle pour $L'$ qui accepte le langage $L''$ en temps polynomial : $1_{L''} = \mathcal{T}_3^{L'}$. La démonstration est très comparable à la démonstration de la transitivité de la relation $\leq_T^{\mathbf{P}}$ : $L'' \leq_T^{\mathbf{P}} L$ et $L \leq_T^{\mathbf{P}} L' \Longrightarrow L'' \leq_T^{\mathbf{P}} L'$. On construit la machine non déterministe $\mathcal{T}_3$ à partir de la machine non déterministe $\mathcal{T}_2$ et de la machine déterministe $\mathcal{T}_1$ en connectant la bande de requête de $\mathcal{T}_2$ à la bande d'entrée de $\mathcal{T}_1$. La machine $\mathcal{T}_3$ recopie la bande de sortie de $\mathcal{T}_1$ sur l'entrée de $\mathcal{T}_1$. De cette manière, on remplace l'oracle pour $L$ de $\mathcal{T}_2$ par la simulation de la machine déterministe $\mathcal{T}_1$, qui reconnait $L$ et est muni d'un oracle pour $L'$. Le déterminisme de $\mathcal{T}_1$ est indispensable pour garantir que la simulation de l'oracle pour $L$ donne toujours un résultat correct. □

Il résulte de la proposition précédente que pour toute classe de $\mathcal{C}$ de langage, on a : $\mathcal{C} \subseteq \mathbf{P}(\mathcal{C})$ et $\mathcal{C} \subseteq \mathbf{NP}(\mathcal{C})$. En effet, soit $L \in \mathcal{C}$, $L \in \mathbf{P}(L) \subseteq \mathbf{NP}(L)$.

En itérant les constructions $\mathbf{P}(\mathcal{C})$, $\mathbf{NP}(\mathcal{C})$ et $\mathbf{co\text{-}NP}(\mathcal{C})$ à partir de la classe $\mathbf{P}$, on obtient la hiérarchie polynomiale.

**Définition E.8.2** *Soit $k \geq 0$, les classes de complexité $\Sigma_k^{\mathbf{P}}$, $\Pi_k^{\mathbf{P}}$ et $\Delta_k^{\mathbf{P}}$ sont définies par :*

- $\Delta_0^{\mathbf{P}} = \Sigma_0^{\mathbf{P}} = \Pi_0^{\mathbf{P}} = \mathbf{P}$
- $\Sigma_{k+1}^{\mathbf{P}} = \mathbf{NP}(\Sigma_k^{\mathbf{P}})$
- $\Pi_{k+1}^{\mathbf{P}} = \mathbf{co\text{-}NP}(\Sigma_k^{\mathbf{P}})$
- $\Delta_{k+1}^{\mathbf{P}} = \mathbf{P}(\Sigma_k^{\mathbf{P}})$

**Proposition 10** $\Sigma_1^{\mathbf{P}} = \mathbf{NP}$, $\Delta_1^{\mathbf{P}} = \mathbf{P}$ *et* $\Pi_1^{\mathbf{P}} = \mathbf{co\text{-}NP}$.
$\Sigma_2^{\mathbf{P}} = \mathbf{NP}(\mathbf{NP})$, $\Delta_2^{\mathbf{P}} = \mathbf{P}(\mathbf{NP})$ *et* $\Pi_2^{\mathbf{P}} = \mathbf{co\text{-}NP}(\mathbf{NP})$.
$\Sigma_{k+1}^{\mathbf{P}} = \mathbf{NP}(\mathbf{NP}^k(\mathbf{NP}))$, $\Delta_{k+1}^{\mathbf{P}} = \mathbf{P}(\mathbf{NP}^k(\mathbf{NP}))$ *et* $\Pi_{k+1}^{\mathbf{P}} = \mathbf{co\text{-}NP}(\mathbf{NP}^k(\mathbf{NP}))$.

**Proposition 11 (Hiérarchie polynomiale)** *Pour tout $k \geq 0$, on a :*

$$\Sigma_k^{\mathbf{P}} \cup \Pi_k^{\mathbf{P}} \subseteq \Delta_{k+1}^{\mathbf{P}} \subseteq \Sigma_{k+1}^{\mathbf{P}} \cup \Pi_{k+1}^{\mathbf{P}} \subseteq \mathbf{PSPACE}$$

### Lien avec la logique du premier ordre

**Définition E.8.3** *(Formules $\Sigma_0$). La classe des formules $\Sigma_0$ peut être construite de la manière suivante :*

1. $\forall x, y, z$, $(x, y) = z \in \Sigma_0$, $\varphi_x(y) = z \in \Sigma_0, x = y \in \Sigma_0$, $x \leq y \in \Sigma_0$
2. *Si $F, G \in \Sigma_0$ alors $F \wedge G$, $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$, $\neg F$ appartiennent à $\Sigma_0$.*
3. *Si $F \in \Sigma_0$, alors les formules $\forall x \leq y : F$ et $\exists x \leq y : F$ sont aussi des formules $\Sigma_0$.*

**Proposition E.8.2** *Les fonctions calculables ou récursives totales correspondent aux relations fonctionnelles $\Sigma_0$. Les formules $\Sigma_0$ expriment les ensembles décidables. Tous les ensembles calculables sont des ensembles $\Sigma_0$.*

**Définition E.8.4** *(Formules, ensembles $\Sigma_n, \Pi_n$).*

1. *Soit $n > 0$. Un préfixe $\Sigma_n$ est un préfixe qui comence par $\exists$ et comporte $n-1$ alternances. Une formule $\Sigma_n$ est une formule de préfixe $\Sigma_n$. Un ensemble est $\Sigma_n$ s'il est exprimé par une formule $\Sigma_n$.*
2. *Soit $n > 0$. Un préfixe $\Pi_n$ est un préfixe qui comence par $\forall$ et comporte $n-1$ alternances. Une formule $\Pi_n$ est une formule de préfixe $\Pi_n$. Un ensemble est $\Pi_n$ s'il est exprimé par une formule $\Pi_n$.*

**Proposition E.8.3** *Les fonctions semi-calculables ou partielles récursives correspondent aux relations fonctionnelles $\Sigma_1$. Les formules $\Sigma_1$ expriment les ensembles récursivement énumérables ou semi-décidables. Tous les ensembles semi-calculables sont des ensembles $\Sigma_1$.*

**Proposition E.8.4** *Le prédicat $\varphi_z = \varphi_{z'}$ est un prédicat $\Sigma_1$.*

**Définition E.8.5** *(Ensemble $\Sigma_n$-complet, $\Pi_n$-complet). Soient $E$ et $F$ deux ensembles.*

- *$E \leq_r F$ s'il existe une fonction récursive totale $f$ telle que :*

$$\forall x,\ x \in E \Leftrightarrow f(x) \in F.$$

  *On dit que $f$ réduit $E$ à $F$.*
- *Un ensemble $F$ est $\Sigma_n$-complet s'il est $\Sigma_n$ et si pour tout ensemble $E$ de classe $\Sigma_n$, $E \leq_r F$.*
- *Un ensemble $F$ est $\Pi_n$-complet s'il est $\Pi_n$ et si pour tout ensemble $E$ de classe $\Pi_n$, $E \leq_r F$.*

**Proposition E.8.5** *Le prédicat '$\varphi_z$ est récursif total' est un prédicat $\Pi_2$. L'ensemble $\{z | \varphi_z$ est calculable $\}$ est un ensemble $\Pi_2$-complet.*

**Théorème E.8.1** $\Pi_0 = \Sigma_0$
$\forall P$, $P \in \Sigma_n \Leftrightarrow \neg P \in \Pi_n$
$\Sigma_n \cup \Pi_n \subset \Sigma_{n+1} \cup \Pi_{n+1}$

**Proposition E.8.6** *Nous pouvons établir les relations suivantes entre les classes de problèmes de la théorie de la comlexité et les classes de formules les exprimant :*

1. *La classe $\Sigma_0 = \Pi_0$ est la classe* **P** *des problèmes polynomiaux.*
2. *La classe $\Sigma_1$ est la classe* **NP** *des problèmes non polynomiaux.*
3. $\Pi_1 =$ **co-NP**.
4. $\Sigma_2 = \mathbf{NP}^{\mathbf{NP}}$.
5. $\Pi_2 = \mathbf{co\text{-}NP}^{\mathbf{NP}}$.
6. *Plus généralement, $\Sigma_{n+1} = \mathbf{NP}^{\Sigma_n}$ et $\Pi_{n+1} = \mathbf{co\text{-}NP}^{\Sigma_n}$.*

Nous rappelons ici les correspondances existant entre différentes notations de la théorie de la complexité :
- $\Pi_0 = \Sigma_0 =$ **P**, la classe des problèmes polynomiaux (c'est-à-dire pouvant être résolus par un algorithme polynomial)
- $\Sigma_1 =$ **NP**, la classe des problèmes de décision pour lesquels il existe un système de preuves courtes et faciles à vérifier
- $\Pi_1 =$ **co-NP**, la classe des problèmes de décision dont le complémentaire est dans **NP**
- $\forall i \geq 1$, $\Pi_{i+1} = \mathbf{NP}^{\Sigma_i}$
- $\forall i \geq 1$, $\Sigma_{i+1} = \mathbf{co\text{-}NP}^{\Sigma_i}$

## E.9 Réduction et complétude

**Définition E.9.1 (Réduction)** *On appelle réduction toute relation $\leq$ reflexive et transitive sur les langages. Autrement dit, une réduction $\leq$ vérifie, pour tout les langages $L \subseteq \mathcal{A}^*$, $L' \subseteq \mathcal{B}^*$ et $L'' \subseteq \mathcal{C}^*$,*

1. *$L \leq L$ et :*
2. *$L \leq L'$ et $L' \leq L'' \Longrightarrow L \leq L''$*

**Définition E.9.2 (Classe de langage initiale pour une réduction, Langage $\leq$-difficile, Langage $\leq$-complet)** *Soit $\mathcal{C}$ une classe de langage et soit $\leq$ une réduction. On dit que $\mathcal{C}$ est initiale pour $\leq$ si, pour tous les langages $L$ et $L'$ :*

$$L \leq L' \text{ et } L' \in \mathcal{C} \Rightarrow L \in \mathcal{C}.$$

*On dit qu'un langage $L$ est $\leq$-difficile pour $\mathcal{C}$ si, pour tout langage $L'$,*

$$L' \in \mathcal{C} \Rightarrow L' \leq L.$$

*Si de plus $L \in \mathcal{C}$, on dit que $L$ est $\leq$-complet pour $\mathcal{C}$.*

**Définition E.9.3 (Réduction de Karp)** *Pour tout langage $L \subseteq \mathcal{A}^*$, $L' \subseteq \mathcal{B}^*$, on a $L \leq_m^R L'$ s'il existe une fonction récursive $f : \mathcal{A}^* \longrightarrow \mathcal{B}^*$ telle que : $w \in L \Longleftrightarrow f(w) \in L'$.*

On particularise cette réduction en ajoutant des contraintes sur $f$.

**Définition E.9.4 (Réduction de Karp $\leq_m^{\mathbf{P}}$, $\leq_m^{\mathbf{L}}$)** *Soient $L \subseteq \mathcal{A}^*$ et $L' \subseteq \mathcal{B}^*$ deux langages et $f : \mathcal{A}^* \longrightarrow \mathcal{B}^*$ une fonction telle que : $w \in L \Longleftrightarrow f(w) \in L'$.*

*On dit que $L$ se réduit à $L'$ au sens de Karp en temps polynomial, si $f$ peut être calculée en temps déterministe polynomial. Dans ce cas, on note $L \leq_m^{\mathbf{P}} L'$. Si $L \leq_m^{\mathbf{P}} L'$ et $L' \leq_m^{\mathbf{P}} L$, on dit que $L$ et $L'$ sont équivalents pour les réduction de Karp polynomiales. On note cette propriété $L \equiv_m^{\mathbf{P}} L'$.*

*On dit que $L$ se réduit à $L'$ au sens de Karp en espace logarithmique, si $f$ peut être calculée en espace déterministe logarithmique. Dans ce cas, on note $L \leq_m^{\mathbf{L}} L'$. Si $L \leq_m^{\mathbf{L}} L'$ et $L' \leq_m^{\mathbf{L}} L$, on dit que $L$ et $L'$ sont équivalents pour les réduction de Karp logarithmiques. On note cette propriété $L \equiv_m^{\mathbf{L}} L'$.*

**Définition E.9.5 (Langage $\mathcal{C}$-complet)** *Soit $\mathcal{C}$ une classe de complexité et soit $L$ un langage. On dit que $L$ est $\mathcal{C}$-complet, si $L$ est $\leq_m^{\mathbf{P}}$-complet pour la classe $\mathcal{C}$.*

**Définition E.9.6 (Problème $\mathcal{C}$-complet)** *Si $L$ est le langage associé à un problème de décision et si ce langage est $\mathcal{C}$-complet, le problème est dit $\mathcal{C}$-complet.*

Pour montrer q'un problème de décision $\Pi$ est dans **NP**, on utilise très souvent le théorème suivant :

**Théorème E.9.1** *Un langage $L$ est dans* **NP** *si et seulement s'il existe un langage $L'$ dans* **P** *et un entier $n$ tels que :*

$$x \in L \Longleftrightarrow (\exists y,\, y \leq |x|^n)(\|x, y\| \in L').$$

**Preuve** : Supposons $L \in \mathbf{N}P$. Il existe donc une machine de Turing non déterministe $\mathcal{T}$ qui accepte le langage $L$ en temps polynomial : $\mathcal{T} = \mathbf{1}_L$. On peut donc supposer qu'il existe un entier $n$ tel que sur l'entrée $x \in L$, la machine $\mathcal{T}$ accepte ou rejette $x$ en au plus $|x|^n$ transitions.

Soit $r$ le degré d'indétermination de la machine $\mathcal{T}$. On ajoute $r$ nouveaux symboles à l'alphabet $\mathcal{A}$ de la machine $\mathcal{T}$, représentant les choix possibles en cas d'indétermination. Nous allons construire sur ce nouvel alphabet une machine de Turing $\mathcal{T}'$ déterministe qui vérifie par construction l'équivalence :

$$x \in L \Longleftrightarrow (\exists y,\, y \leq |x|^n)(\|x, y\| \in L')$$

et reconnait le langage correspondant en temps polynomial.

On construit cette machine de Turing $\mathcal{T}'$ de la manière suivante : sur toute entrée $w$ de la forme $\|x, y\|$, où $x \in \mathcal{A}^*$ et $y$ est un mot constitué exclusivement de à l'aide des nouveaux symboles tel que $|y| \leq |x|^n$, la machine $\mathcal{T}'$ simule la machine $\mathcal{T}$ de manière déterministe pour au plus $|y|$ transitions. À chaque transition, elle s'aide du symbole correspondant de $y$ pour choisir la transition de $\mathcal{T}$ à effectuer.

Si lors de cette simulation, la machine $\mathcal{T}$ accepte $x$, la machine $\mathcal{T}'$ accepte $w$. Sinon elle rejette le mot $w$. Par construction, le langage $L' = \{\|x, y\|,\, x \in L,\, y = b_1, \ldots, b_r\}$ reconnu par $\mathcal{T}'$ est bien dans $\mathbf{P}$ et :

$$x \in L \Longleftrightarrow (\exists y,\, y \leq |x|^n)(\|x, y\| \in L').$$

Réciproquement, supposons qu'il existe un langage $L' \in \mathbf{P}$ tel que l'équivalence précédente soit satisfaite. Soit $\mathcal{T}'$ une machine de Turing déterminste qui reconnait le langage $L'$ : $\mathcal{T} = \mathbf{1}_L$. Il s'agit de montrer que $L \in \mathbf{N}P$. On construit une machine non déterministe $\mathcal{T}$ qui accepte en temps polynomial le langage $L$ de la manière suivante :

- dans un premier temps, la machine $\mathcal{T}$, sur l'entrée $x \in \mathcal{A}^*$, génère de manière non déterministe un mot $y$ tel que $|y| \leq |x|^n$ ;
- puis elle forme le mot $\|x, y\|$ sur la bande de travail ;
- ensuite, la machine $\mathcal{T}$ simule la machine $\mathcal{T}'$ à partir de la bande de travail.

La machine $\mathcal{T}$ accepte le mot $x$ si et seulement si la simulation de $\mathcal{T}'$ aboutit à un état acceptant. La machine $\mathcal{T}$ accepte donc le langage $L$ en temps polynomial d'ou $L \in \mathbf{N}P$. $\square$

Ce théorème permet de simplifier les preuves d'appartenance à **NP**. Il suffit de construire un algorithme acceptant le langage $L(\Pi)$, et utilisant une instruction spécifique **deviner** qui traduit la partie non déterministe de l'algorithme. Cette instruction correspond à : $(\exists y,\, y \leq |x|^n)(\|x, y\| \in L')$ et permet de produire une valeur $y$. L'algorithme conserve la charge de vérifier que $y$ est bien de taille polynomiale en fonction de la taille de l'entrée : $y \leq |x|^n$. Le plus souvent, c'est immédiat par le choix de l'ensemble dans lequel $y$ est pris. Le seul point à préciser reste donc que les autres opérations de l'algorithme peuvent être faites en temps déterministe polynomial.

Pour montrer qu'un problème $\Pi$ est **NP**-complet, on montre que le langage associé au problème $\Pi$ est $\leq_m^{\mathbf{P}}$-complet pour la classe **NP**. Il s'agit donc de démontrer :

1. $L(\Pi) \in \mathbf{NP}$, i.e. que l'on on est capable de décrire un algorithme non déterministe reconnaissant le langage $L(\Pi)$ dont le temps d'exécution est un polynome fonction de la taille de l'entrée. Pour montrer que $\Pi$ est dans **NP**, il suffit de montrer que l'on peut décrire un algorithme acceptant le langage $L(\Pi)$, disposant d'une instruction supplémentaire de la forme **deviner** qui traduit la partie non déterministe de l'algorithme et que cet algorithme s'exécute en temps polynomial relativement à la taille de l'entrée. L'instruction **deviner** retourne une valeur qui doit être de taille polynomiale en fonction de la taille de l'entrée. Cette vérification est à la charge de l'algorithme.
2. $L(\Pi)$ est $\leq_m^{\mathbf{P}}$-difficile pour la classe **NP**, i.e. que tout langage de **NP** se réduit à $L(\Pi)$ au sens de Karp en temps polynomial. Pour montrer ceci, il suffit souvent de montrer qu'un langage **NP**-complet bien connu se réduit à $L(\Pi)$ au sens de Karp en temps polynomial. Pour démontrer cette réduction, il suffit de décrire un algorithme reconnaissant le langage $L(\Pi)$, disposant d'un accès par oracle à la solution du problème $L$ et s'exécutant en temps polynomial, en autant que soit compté à coût unitaire les décisions de l'oracle.

## E.10 Les problèmes SAT, $k$-SAT

### Expressions/formules booléennes

**Définition E.10.1 (Expression booléenne)** *Soit $\mathcal{X}$ un ensemble de symboles appelés variables booléennes. On appelle expression booléenne sur $\mathcal{X}$ toute expression formée en utilisant les symboles* 0 *et* 1*, les symboles de $\mathcal{X}$, les opérateurs binaires $\wedge$ et $\vee$, l'opérateur unaire $\neg$ et les parenthèses*[2]*, en respectant les règles de construction suivantes :*

1. *toute variable $x \in \mathcal{X}$ est une expression booléenne ; les symboles* 0 *et* 1 *sont des expressions booléennes ;*

[2] Pour simplifier les notations, on suppose que les opérateurs sont classés par priorité croissante : $\wedge$, $\vee$, $\neg$, ce qui permet de supprimer les parenthèses redondantes.

2. *si $F$ est une expression booléenne, alors $\neg F$ est une expression booléenne ;*
3. *si $F$ et $F'$ sont des expressions booléennes, alors $(F \vee F')$ et $(F \wedge F')$ sont des expressions booléennes.*

**Définition E.10.2 (Interprétation d'une expression booléenne)** *On appelle interprétation toute application $\tau$ de $\mathcal{X}$ dans $\{0,1\}$. On prolonge $\tau$ à l'ensemble des expressions booléennes de la manière suivante :*
- $\tau(0) = 0$, $\tau(1) = 1$, $\tau(\neg F) = 1 - \tau(F)$
- $\tau(F \vee F') = \max(\tau(F), \tau(F'))$, $\tau(F \wedge F') = \min(\tau(F), \tau(F'))$

*Deux expressions booléennes $F$ et $F'$ sont dites équivalentes si pour toute interprétation $\tau$, on a : $\tau(F) = \tau(F')$. On note dans ce cas $F \equiv F'$.*

**Définition E.10.3 (Satisfaction et consistance d'un ensemble d'expressions booléennes)** *Soit $\mathcal{F}$ un ensemble d'expressions booléennes sur $\mathcal{X}$. On dit que l'interprétation $\tau$ satisfait $\mathcal{F}$, si pour tout $F \in \mathcal{F}$ on a $\tau(F) = 1$. On dit que $\mathcal{F}$ est consistant s'il existe une interprétation qui satisfait $\mathcal{F}$. On dit que $\mathcal{F}$ est inconsistant si aucune interprétation ne satisfait $\mathcal{F}$.*

**Définition E.10.4 (Littéral, disjonction élémentaire, conjonction élémentaire)** *On appelle littéral, toute expression de la forme $x$ ou $\overline{x}$, où $x \in \mathcal{X}$.*
*On appelle disjonction élémentaire, toute expression de la forme $l_1 \vee \ldots \vee l_n$ où $l_1, \ldots, l_n$ sont des littéraux distincts.*
*On appelle conjonction élémentaire, toute expression de la forme $l_1 \wedge \ldots \vee l_n$ où $l_1, \ldots, l_n$ sont des littéraux distincts.*

**Définition E.10.5 (Clause, clause minimale)** *Soit $F$ une expression booléenne non constante. On dit qu'une disjonction élémentaire $C$ est une clause de $F$ si, pour toute interprétation $\tau$, $\tau(C) = 0 \Rightarrow \tau(F) = 0$.*
*Une clause $C$ de $F$ est dite minimale si aucune disjonction élémentaire formée à partir d'un sous-ensemble strict des littéraux de $C$ n'est une clause de $F$.*

**Définition E.10.6 (Impliquant, impliquant minimal)** *Soit $F$ une expression booléenne non constante. On dit qu'une conjonction élémentaire $I$ est un impliquant de $F$ si, pour toute interprétation $\tau$, $\tau(I) = 1 \Rightarrow \tau(F) = 1$.*
*Un impliquant $I$ de $F$ est dit minimal si aucune conjonction élémentaire formée à partir d'un sous-ensemble strict des littéraux de $I$ n'est un impliquant de $F$.*

**Définition E.10.7 (Forme normale disjonctive , Forme normale conjonctive)** *Soit $F$ une expression booléenne. On appelle forme normale conjonctive de $F$ la conjonction de ses clauses minimales.*
*On appelle forme normale disjonctive de $F$ la disjonction de ses impliquants minimaux.*

**Définition E.10.8 (Problème SAT)** *Une instance du problème SAT est une expression booléenne $F$ sur l'ensemble fini de variables $\mathcal{X}$. La question est de savoir s'il existe une interprétation $\tau$ qui satisfasse $F$.*

**Théorème E.10.1** *Le problème SAT est* **NP***-complet.*

**Définition E.10.9 (Problème $k$-SAT)** *Une instance du problème $k$-SAT est un ensemble de clauses $\mathcal{C}$ sur l'ensemble fini de variables $\mathcal{X}$ où chaque clause est formée d'exactement $k$ littéraux. La question est de savoir s'il existe une interprétation $\tau$ qui satisfasse toutes les clauses de $\mathcal{C}$.*

**Théorème E.10.2** *Pour $k \geq 3$, le problème $k$-SAT est* **NP***-complet.*

## E.11 La thèse de Church

**Thèse E.11.1** *(Thèse de Church). La classe des fonctions récursives partielle est égale à la classe des fonctions calculables par un algorithme quelconque.*

La classe des fonctions récursives partielles (definition E.3) est définie formellement comme la plus petite classe de fonctions partielles qui contient les fonctions de bases et qui est fermée par les constructeurs composition, recursion primitive (definition E.3) et minimisation non bornée (définition E.3). En revanche, la notion de fonction calculable par un algorithme quelconque n'est pas formellement définie. Il s'agit d'une notion intuitive et informelle.

Pour se convaincre que chaque fonction récursive partielle est calculable par un algorithme, il suffit de voir que l'on peut facilement programmer les fonctions de base et les constructeurs de cette classe de fonctions :

```
Fonction fog(x) :
    Retourner f(g(x))
Fin
Fonction Rec(g,h)(y,x) :
    r ← g(x)
    Pour i de 0 à y − 1 faire
        r ← h(i,r,x)
    Fin Pour
    Retourner r
Fin
Fonction μy.(f(y,.) = 0)(x) :
    y ← 0
    Tant que (f(y,x) ≠ 0) faire
        y ← y + 1
    Fait
    Retourner y
Fin
```

Algorithme 8: Les fonctions récursives partielles sont calculables

La réciproque est admise, dans la mesure où l'on observe que toutes les définitions formelles de fonctions calculables sont prouvées équivalentes à la récursivité partielle. Donc si il existe des fonctions calculables par un algorithme qui ne sont pas récursives partielles, on ne les a pas encore découvertes.

## E.12 La machine universelle de Turing

Le théorème E.12.1 est dû à Turing. Il est connu sous le nom de *théorème de la machine universelle*. Il montre l'existence de machines de Turing capables de simuler n'importe quelle autre machine de Turing.

**Théorème E.12.1** *(Théorème de la machine universelle). Pour tout $k \geq 1$, il existe un entier $u_k$ tel que :*

$$\varphi_{u_k}^{(k+1)}(z,x) = \varphi_z^{(k)}(x).$$

**Preuve** : Soit $k \geq 1$, la fonction $\lambda zx\varphi_z^{(k)}(x)$ est récursive. Donc cette fonction d'arité $(k+1)$ possède un indice $u_k$. □

#### Le théorème de récursion

Le théorème E.12.2 est dû à Kleene. Il est connu sous le nom de *théorème du paramétrage* ou encore théorème $s_n^m$.

**Théorème E.12.2** *(Théorème $s_n^m$). Pour tous les entiers $m$, $n$, il existe une fonction élémentaire $s_n^m(z,x)$ d'arité $m+1$ telle que :*

$$\varphi_z^{(m+n)}(x,y) = \varphi_{s_n^m(z,x)}^{(n)}(y).$$

**Preuve informelle** : on définit la fonction $s_n^m(z,x)$ comme étant un nombre de Gödel d'une machine de Turing. Cette machine de Turing a en entrée $y$. Elle ajoute $x$ devant $y$ puis exécute le code de la machine de Turing dont le nombre de Gödel est $z$. Une preuve explicite de ce résultat est fournie dans [Rey04].

Nous sommes donc capables de construire un programme qui lit son propre code :

$$\varphi_z^{(m+n)}(z,y) = \varphi_{s_n^m(z,z)}^{(n)}(y).$$

Le théorème E.12.3 est dû à Kleene. Il est connu sous le nom de *premier théorème de récursion de Kleene* ou encore sous le nom de *théorème du point fixe de Kleene*.

**Théorème E.12.3** *Soit $H$ un opérateur récursif. Il existe une fonction récursive $f_H$ telle que :*

1. $H(f_H) = f_H$ ;
2. $\forall g$ *telle que* $H(g) = g, f_H \subseteq g$.

Le théorème E.12.4 est dû à Kleene. Il est connu sous le nom de *second théorème de récursion de Kleene*.

**Théorème E.12.4** *(Second théorème de récursion de Kleene). Pour toute fonction semi-calculable $f(y,x)$, il existe un programme $z$ tel que pour tout $x$,*

$$\varphi_z(x) = f(z,x).$$

**Preuve** : soit $f(y,x)$ une fonction semi-calculable. D'après le théorème $s_n^m$ , il existe une fonction élémentaire $s$ telle que :

$$\varphi_z(y,x) = \varphi_{s(z,y)}(x).$$

La fonction $g(y,x) = f(s(y,y),x)$ est semi-calculable, comme composée de fonctions semi-calculables, donc il existe un programme $z_0$ calculant la fonction $g$ :

$$g(y,x) = \varphi_{z_0}(y,x).$$

On a donc :

$$\varphi_{s(z_0,z_0)}(x) = \varphi_{z_0}(z_0,x) = g(z_0,x) = f(s(z_0,z_0),x).$$

Nous avons donc établit l'existence d'un programme $z_1 = s(z_0,z_0)$ qui vérifie :

$$\varphi_{z_1}(x) = f(z_1,x).\square$$

**Remarque** : le programme $z_1$ vérifiant $\varphi_{z_1}(x) = f(z_1,x)$ est appelé point fixe de Kleene de la fonction $f$. La fonction $f$ est semi-calculable. Soit z un programme la calculant.

$$f(z_1,x) = \varphi_z(z_1,x).$$

D'après le théorème $s_n^m$, il existe une fonction élémentaire $s$ telle que :

$$\varphi_z(z_1,x) = \varphi_{s(z,z_1)}(x).$$

Si $z_1$ est un point fixe de Kleene de la fonction $f$, on a donc :

$$\varphi_{z_1}(x) = f(z_1,x) = \varphi_z(z_1,x) = \varphi_{s(z,z_1)}(x).$$

La fonction $h(y) = S(z,y)$ vérifie donc :

$$\varphi_{z_1}(x) = \varphi_{h(z_1)}(x).$$

Nous allons voir que $z_1$ est alors un point fixe de Rogers la fonction $h$.

Le théorème E.12.5 est dû à Rogers. Il est connu sous le nom de *théorème du point fixe de Rogers*.

**Théorème E.12.5** *(Théorème du point fixe) Pour toute fonction calculable $f$, il existe un programme $z$ tel que pour tout $x$,*

$$\varphi_z(x) = \varphi_{f(z)}(x).$$

**Preuve** : soit $f$ une fonction calculable. D'après le théorème $s_n^m$ , il existe une fonction élémentaire $s$ telle que :

$$\varphi_z(y, x) = \varphi_{s(z,y)}(x).$$

Considérons la fonction $g(y, x) = \varphi_{\varphi_y(y)}(x)$. La fonction $g(y, x)$ est semi-calculable, comme composée de fonctions semi-calculables. Donc il existe un programme $z_0$ la calculant :

$$g(y, x) = \varphi_{z_0}(y, x).$$

Considérons la fonction $h(y) = f(s(z_0, y))$. La fonction $h(y)$ est calculable, comme composée de fonctions calculables. Donc il existe un programme $z_1$ la calculant :

$$h(y) = \varphi_{z_1}(y).$$

On a donc :

$$\varphi_{s(z_0,z_1)}(x) = \varphi_{z_0}(z_1, x) = g(z_1, x) = \varphi_{\varphi_{z_1}(z_1)}(x) = \varphi_{h(z_1)}(x) = \varphi_{f(s(z_0,z_1))}(x).$$

Nous avons donc établit l'existence d'un programme $z_2 = s(z_0, z_1)$ qui vérifie :

$$\varphi_{z_2}(x) = \varphi_{f(z_2)}(x).$$

□

## E.13 Le problème de l'arrêt

On dispose d'un programme $\varphi_z$ qui effectue un calcul sur un nombre $x$. La question est de savoir si ce programme fonctionne correctement : le programme s'arrêtera-t-il après qu'on lui a donné le nombre, ou se perdra-t-il indéfiniment en calcul ? Considérons l'ensemble $W_x = \{z | \, \varphi_z(x) \downarrow\} = \mathrm{Dom}(\varphi_x)$ de tous les programmes qui s'arrêtent pour une entrée fixée $x$. La question est donc de savoir si le prédicat $z \in W_x$ est décidable, c'est-à-dire si la fonction caractéristique de ce prédicat est pré-récursive. Le théorème suivant montre que ce prédicat n'est pas décidable, en d'autre terme qu'il n'existe pas de programme qui s'arrête toujours et qui donne le résultat OUI si $\varphi_z(x) \downarrow$ et NON sinon.

**Théorème E.13.1 (Problème de l'arrêt)** *Le prédicat $z \in W_x$ est indécidable.*

**Preuve** : Supposons que le prédicat $x \in W_x$ soit décidable, donc que la fonction caractéristique $\psi(x) = 1_{x \in W_x}$ soit pré-récursive. Elle est donnée par :

$$\psi(x) = \begin{cases} 1 & \text{si } \varphi_x(x) \downarrow \\ 0 & \text{sinon.} \end{cases}$$

Dans ce cas, $\psi(x)$ est toujours défini et, en particulier, le prédicat $\psi(x) = 0$ est récursif, donc la fonction caractéristique de ce prédicat

$$\phi(x) = 1_{\psi(x)=0} = \begin{cases} 1 & \text{si } \psi(x) = 0 \\ \uparrow & \text{sinon.} \end{cases}$$

est récursive, donc $\exists z_0 | \, \phi(x) = \varphi_{z_0}(x)$.
$\varphi_{z_0}(z_0) \downarrow \Longleftrightarrow \phi(z_0) \downarrow \Longleftrightarrow \psi(z_0) = 0 \Longleftrightarrow \varphi_{z_0}(z_0) \uparrow$. Ceci est contradictoire donc le prédicat $x \in W_x$ n'est pas décidable. Si le prédicat $z \in W_x$ était décidable, il en serait de même du prédicat $x \in W_x$. On en déduit donc que le prédicat $z \in W_x$ n'est pas décidable.
□

# Table des matières


**I Analyse de la menace virale** **11**

**1 Fondements théoriques de la virologie informatique** **12**
1.1 Introduction . . . 12
1.2 Définition formelle d'un virus informatique . . . 12
1.3 Le problème de la détection virale . . . 18
1.4 Problèmes ouverts de la virologie informatique . . . 20
1.5 Conclusion . . . 22

**2 Cryptovirologie boîte blanche** **23**
2.1 Introduction . . . 23
2.2 Cadre théorique . . . 25
2.3 Utilisation de la cryptographie par un virus à des fins d'extorsion . . . 27
2.4 Exemples d'utilisation en contexte viral . . . 30
2.5 Implémentations boîte blanche du DES et de l'AES . . . 32
2.6 Evaluation . . . 35
2.7 Conclusion . . . 39

**3 Protection logicielle par embarquement de problèmes difficiles** **42**
3.1 Introduction . . . 42
3.2 Solution et mécanismes de protection logicielle . . . 44
3.3 Introduction aux techniques et méthodes de rétro-ingénierie . . . 53
3.4 Évaluation d'une solution de protection . . . 73
3.5 Conclusion . . . 77

**II Lutte anti-virale** **79**

**4 Modélisation de la rétro-ingénierie par interprétation abstraite** **80**
4.1 Introduction . . . 80
4.2 Formalisation du glaive et du bouclier par interprétation abstraite . . . 81
4.3 Extraction d'un schéma de détection . . . 87
4.4 Vers une modélisation de l'analyse dynamique par interprétation abstraite . . . 89
4.5 Conclusion . . . 91

**5 Unpacking** **92**
5.1 Introduction . . . 92
5.2 Unpacking sécurisé et avancé . . . 93
5.3 Vers un outil d'analyse fiable . . . 103
5.4 Conclusion . . . 115

**6 Analyse des virus furtifs** **117**
6.1 Introduction . . . 117
6.2 Autres travaux . . . 119
6.3 Conception de notre solution . . . 121
6.4 Evaluation . . . 132
6.5 Conclusion . . . 135

**7 Modélisation statistique du problème de la détection virale** **137**
7.1 Introduction . . . 137
7.2 Modélisation statistique de la détection antivirale . . . 138
7.3 Application à l'analyse spectrale . . . 142
7.4 Analyse spectrale par approche bayésienne . . . 145
7.5 Analyse spectrale par modèle Markovien . . . 150
7.6 Conclusion . . . 162

**III Évaluation** **164**

**8 Évaluation d'un produit anti-virus** **165**
8.1 Introduction . . . 166
8.2 Énoncé de la problématique de sécurité . . . 168
8.3 Pertinence de la conception du moteur de détection . . . 171
8.4 Conformité et robustesse de l'implémentation . . . 176
8.5 Plate-forme de tests . . . 177
8.6 Conclusion . . . 181

**A Tests de conformité et d'efficacité des fonctions de sécurité** **188**
A.1 Installation et paramétrage du produit . . . 188
A.2 Test de conformité des fonctions de sécurité . . . 190

**B Critères de sécurité boîte noire et génération aléatoire de S-Boxes vérifiant ces critères** **201**
B.1 Critères de sécurité boîte noire . . . 201
B.2 Génération aléatoire de S-Boxes . . . 214

**C Implémentation et cryptanalyse du DES et de l'AES boîte blanche** **224**
C.1 Implémentation boîte blanche de l'algorithme DES . . . 224
C.2 Cryptanalyse de l'algorithme DES boîte blanche . . . 225
C.3 Cryptanalyse de l'algorithme AES boîte blanche . . . 229

**D Modélisation des transformations de programme par interprétation abstraite** **233**
D.1 Rappels théorique . . . 233
D.2 Modélisation (et hiérarchisation) des sémantiques d'un programme . . . 240
D.3 Modélisation de l'analyse du flot de données, théorème du point fixe . . . 244
D.4 Modélisation des transformations de programmes . . . 247

**E Fondamentaux de la théorie de la complexité** **250**
E.1 Analyse en pire et moyen cas . . . 250
E.2 Modèles de machine . . . 251
E.3 Fonctions calculables par une machine à registre . . . 253
E.4 Les machines de Turing . . . 255
E.5 Codage d'une machine de Turing . . . 259
E.6 Décidabilité d'un problème de décision . . . 262
E.7 Langage accepté/reconnu par une machine de Turing . . . 263
E.8 Classes de complexité et hiérarchie polynomiale . . . 263
E.9 Réduction et complétude . . . 266

E.10 Les problèmes SAT, $k$-SAT . . . . . 267
E.11 La thèse de Church . . . . . 268
E.12 La machine universelle de Turing . . . . . 269
E.13 Le problème de l'arrêt . . . . . 271

# Table des figures


1 Moteur de détection : de l'extraction d'information au schéma de détection . . . 3
2 Moteur de détection : la fonction de détection . . . 4

2.1 Un tour du DES et son équivalent en boîte blanche . . . 33
2.2 $R_j^r$, $j = 0, \ldots, 3$, $r = 1, \ldots, 9$ . . . 38

3.1 Synoptique de la chaîne de compilation CSS . . . 44
3.2 Invariants d'un saut . . . 56
3.3 Graphes VCG de `call eax` sous Ollygraph et IDA Pro . . . 63
3.4 Graphe VCG de `call eax` sous IDA Pro et VxStripper . . . 63
3.5 Suppression des sauts inutiles sous `VxStripper` . . . 66
3.6 Arbre CFG d'un programme $F$ calculant 4 ! . . . 69
3.7 control flow flattening du programme $F$ . . . 70
3.8 control flow flattening avec *junk code* et prédicats opaques . . . 71

6.1 Rootkit from outside the Matrix . . . 120
6.2 Spectre d'auto-corrélation . . . 131
6.3 Spectre d'auto-corrélation avec masquage des interruptions . . . 132
6.4 Spectre d'auto-corrélation (VMware) . . . 133
6.5 Spectre d'auto-corrélation (VMware) avec masquage des interruptions . . . 134

7.1 Modélisation statistique de la détection virale . . . 139

8.1 Cible d'évaluation et son environnement . . . 169

B.1 Transformée de Fourier et spectre de Walsh de la fonction *f9a* . . . 210
B.2 Transformée de Fourier et spectre de Walsh de la fonction *f9b* . . . 211
B.3 Transformée de Fourier et spectre de Walsh de la fonction *f11* . . . 212
B.4 Non Linéarité des S-Boxes $(6, 4)$ générées aléatoirement . . . 220
B.5 Non Linéarité des S-Boxes $(8, 8)$ générées aléatoirement . . . 220
B.6 Valeur xor des S-Boxes $(6, 4)$ générées aléatoirement . . . 221
B.7 Valeur xor des S-Boxes $(8, 8)$ générées aléatoirement . . . 221
B.8 Distance DBIC(1,2) des S-Boxes $(6, 4)$ générées aléatoirement . . . 222
B.9 Distance DBIC(1,2) des S-Boxes $(8, 8)$ générées aléatoirement . . . 222

# Liste des tableaux


2.1 Complexité du problème de détection de la production $L(G)$ d'une grammaire $G$, ou de manière équivalente du problème : est-ce que $x$ appartient à $L(G)$ ? . . . 31
2.2 espace mémoire nécessaire au stockage des lookup tables . . . 34
2.3 Diversité des lookup tables de WB-AES [CEJvO02b] . . . 36
2.4 Ambiguïté des *lookup tables* de WB-AES [CEJvO02b] . . . 37

3.1 Complexité des problèmes **May Constants** et **Must Constants** dans le cas où le flot de contrôle ne comporte pas de boucle [MOR01] . . . 59
3.2 Complexité des problèmes **May Constants** et **Must Constants** dans le cas où le flot de contrôle peut comporter des boucles [MOR01] . . . 59
3.3 Complexité des problèmes **May Alias** et **Must Alias** dans le cas dans le cas de l'analyse intra-procédurale et sensible au flot d'exécution [Lan92, Hor97] . . . 62

5.1 Packers les plus utilisés par les virus de la collection Vx Heavens, mars 2006 . . . 93
5.2 Récapitulatif des différentes solutions . . . 97

B.1 Caractéristiques des fonctions de rétroaction du système de chiffrement COSvd $(2, 128)$ . . . 209
B.2 Critères de sécurité BBAC du DES, S-Box 1 à 8 . . . 218
B.3 Critères de sécurité BBAC de l'AES . . . 218
B.4 Non-linéarités des fonctions booléennes générées aléatoirement . . . 219
B.5 Caractéristiques des S-Boxes générées aléatoirement . . . 219

# Bibliographie


[AC75] A.V. Aho and M.J. Corasick. Efficient string matching : an aid to bibliographic search. In *Communications of the ACM*, volume 18, pages 333–340. ACM Press New York, NY, USA, 1975.

[ACD+06] B. Anckaert, J. Cappaert, B. DeBus, K. DeBosschere, M. Madou, and B. Preneel. On the Effectiveness of Source Code Transformations for Binary Obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*, 2006.

[AD97] M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 284–293. ACM New York, NY, USA, 1997.

[Ada97] C.M. Adams. Constructing Symmetric Ciphers Using the CAST Design Procedure. *Designs, Codes and Cryptography*, 12(3) :283–316, 1997.

[AdJ06] J. Aycock, R. deGraaf, and M. Jacobson. Anti-disassembly using Cryptographic Hash Functions. In *Journal in Computer Virology*, volume 2, pages 79–85. Springer, 2006.

[Adl90] L. Adleman. An abstract theory of computer viruses. In *Lecture Notes in Computer Science*, volume 403, pages 109–115. Springer, 1990.

[AHH+99] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S.J. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. In *Proceedings of LISA*. Springer, 1999.

[AM+00] C.M. Adams, S.J.M. Mister, et al. Practical S box design, us patent 6,031,911, 2000.

[AMD06] B. Anckaert, M. Madou, and K. DeBosschere. A model for self-modifying code. In *Proceedings of the 8th Information Hiding Conference*. Springer, 2006.

[AS08] McGhee S. Ataluri, S. and M. Stamp. Hunting for metamorphic engines. In *Journal in Computer Virology*. Springer, 2008.

[Asa03] E. Asarin. Calculabilité. Support de cours et de TD, université de Grenoble, 2003.

[Asp08] Aspack. Aspack/Asprotect. http ://www.aspack.com, 2008.

[AT93] C.M. Adams and S.E. Tavares. Designing S-boxes for ciphers resistant to differential cryptanalysis. In *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography, Rome, Italy*, pages 181–190, 1993.

[Bak99] D.W. Baker. The Development of a Common Enumeration of Vulnerabilities and Exposures. *Recent Advances in Intrusion Detection*, 7 :9, 1999.

[Bal06] F. Ballard. Qemu project home page. http ://bellard.org/qemu/, 2006.

[Bay05] U. Bayer. Ttanalyze : A tool for analyzing malware. Master's thesis, Technical University of Vienna, 2005.

[BB87] G. Brassard and P. Bratley. *Algorithmique, conception et analyse*. Masson, 1987.

[BB97] A.H. Baker and A. Baker. *The Windows NT Device Driver Book : A Guide for Programmers*. Prentice Hall, 1997.

[BB02] E. Barkan and E. Biham. In how many ways can you write Rijndael. In *Proceedings of Asiacrypt*. Springer, 2002.

[Bel05] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[Bet06] C. Betz. Memparser, 2006.

[BF07] P. Beaucamps and E. Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. In *Journal in Computer Virology*, volume 3, pages 3–21. Springer, 2007.

[BGEC04] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography*, volume 3357, pages 227–240. Springer, 2004.

[BGI+01] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *Lecture Notes in Computer Science*, volume 2139, pages 19–23. Springer, 2001.

[BH99] D. Brubacher and G. Hunt. Detours : Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1999.

[BH04] J. Butler and G. Hoglund. VICE-Catch the Hookers. *Black Hat USA*, pages 10–17, 2004.

[Bil04] J.A. Bilmes. What HMMs can't do. *Invited paper and lecture, ATR Workshop*, 2004.

[Bil06] J.A. Bilmes. What HMMs Can Do. In *Proceedings of IEICE Transactions on Information and Systems*, pages 869–891. IEICE, 2006.

[BKK06] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze : A Tool for Analyzing Malware. *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[BKM05] G. Bonfante, M. Kaczmarek, and J.Y. Marion. Toward an abstract computer virology. In *Lecture Notes in Computer Science*, volume 3722, pages 579–593. Springer, 2005.

[BKM06] G. Bonfante, M. Kaczmarek, and J.Y. Marion. On Abstract Computer Virology from a Recursion Theoretic Perspective. In *Journal in Computer Virology*, volume 1, pages 45–54. Springer, 2006.

[BKM08] G. Bonfante, M. Kaczmarek, and J.Y. Marion. An implementation of morphological malware detection. In *Proceedings of the 17th EICAR Conference, Laval, France, May 3-6*, 2008.

[Bla04] B. Blanchon. Winpooch watchdog. winpooch.free.fr, 2004.

[Blu67] M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2) :322–336, 1967.

[BM06] T. Brosch and M. Morgenstern. Runtime packers : The hidden problem. *Black Hat 2006 Conference*, 2006.

[BM08] J.-M. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal of Computer Virology*, 2008.

[BMM06] D. Bruschi, L. Martignoni, and M. Monga. Detecting Self-Mutating Malware Using Control Flow Graph Matching. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), IEEE Computer Society*. Springer, 2006.

[Bos06] H. Bos. A personal view on the future of Zero-day Worm Containment (slides), 2006.

[Bra06] A. Braeken. *Cryptographic properties of Boolean functions and S-boxes*. PhD thesis, Katholieke Universiteit Leuven, March, 2006.

[Bru04] N. Brulez. Scan of the Month 33 : Anti Reverse Engineering Uncovered. the Honeynet Project, Scan of the Month 33, 2004.

[Bru05] N. Brulez. Anti Reverse Engineering Uncovered. 2005.

[Bur05] M. Burdach. idetect, procenum and wmft tools. http ://forensic.seccure.net, 2005.

[Bur06a] M. Burdach. An introduction to Windows memory forensic. 2006.

[Bur06b] M. Burdach. Digital forensics of the physical memory. 2006.

[Bur06c] M. Burdach. Digital investigation. http ://forensic.seccure.net, 2006.

[Bur06d] M. Burdach. Finding Digital Evidence In Physical Memory, 2006.

[But06] J. Butler. Raide : Rootkit analysis identification elimination. http ://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Silberman-Butler.pdf, 2006.

[CAF07] CAFE. Mcafee virusscan 8.5i and epolicy orchestrator 3.6.1 validation report, 22 june 2007, 2007.

[Can01] A. Canteaut. Cryptographic functions and design criteria for block ciphers. In *Proceedings of INDOCRYPT*. Springer, 2001.

[Can02] A. Canteaut. Cryptanalyse des chiffrements à clef secrète par blocs. *Journal de la sécurité informatique MISC, (2), March*, 2002.

[Car06a] H. Carvey. Ramdump, lsproc, lspm, ReadPE tools. http ://sourceforge.net/projects/windowsir, 2006.

[Car06b] H. Carvey. Reassembling an image file from a memory dump. http ://sourceforge.net/projects/windowsir, 2006.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press New York, NY, USA, 1977.

[CC77b] P. Cousot and R. Cousot. Towards a universal model for static analysis of programs. 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM New York, NY, USA, 1979.

[CC01] P. Cousot and R. Cousot. A Case Study in Abstract Interpretation Based Program Transformation Blocking Command Elimination. In *Electronic Notes in Theoretical Computer Science*, volume 45, pages 41–64. Elsevier, 2001.

[CC02] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 178–190. ACM New York, NY, USA, 2002.

[CCE05] CCEVS. Us government protection profile anti-virus applications for workstations in basic robustness environments, version 1.0, 6 january 2005. http ://niap.nist.gov/cc-scheme/pp/PP_VID10053-PP.html, 2005.

[CCE06] CCEVS. Us government protection profile anti-virus applications for workstations in basic robustness environments, version 1.1, 4 april 2006, 2006.

[CCE07] CCEVS. Us government protection profile anti-virus applications for workstations in basic robustness environments, version 1.2, 25 july 2007, 2007.

[CEJvO02a] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot. A White-Box DES Implementation for DRM Applications. In *Proceedings of Digital Rights Management Workshop*, pages 1–15. Springer, 2002.

[CEJvO02b] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot. White-Box Cryptography and an AES Implementation. In *Proceedings of the Ninth Workshop on Selected Areas in Cryptography (SAC 2002)*. Springer, 2002.

[CGH04] R. Canetti, O. Goldreich, and S. Halevi. The Random Oracle Methodology, Revisited. *Journal of the ACM*, 51(4) :557–594, 2004.

[Cho56] N. Chomsky. Three models for the description of language. *Information Theory, IEEE Transactions on*, 2(3) :113–124, 1956.

[Cif96] C. Cifuentes. Interprocedural dataflow decompilation. *Journal of Programming Languages*, 4(2) :77–99, 1996.

[Cin75] E. Cinlar. Introduction to Stochastic Processes. *Englewood Cliffs*, 1975.

[CJ03] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

[CJ04] M. Christodorescu and S. Jha. Testing malware detectors. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 34–44. ACM Press New York, NY, USA, 2004.

[CJK+07] M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith. Software transformations to improve malware detection. In *Journal in Computer Virology*, volume 3, pages 253–265. Springer, 2007.

[CJS+05] M. Christodorescu, S. Jha, SA Seshia, D. Song, and RE Bryant. Semantics-aware malware detection. *Security and Privacy, 2005 IEEE Symposium on Security & Privacy*, pages 32–46, 2005.

[CKJ+05] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical report, (1539), University of Wisconsin, Madison, Wisconsin, USA, November, 2005.

[Clo07] Cloakware. Cloakware Security Suite. http ://www.cloakware.com, 2007.

[CMH03] C. Collberg, G. Mylesm, and A. Huntwork. Sandmark : A Tool for Software Protection Research. IEEE Computer Society, 2003.

[Coh86] F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, January, 1986.

[Cou02] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In *Theoretical Computer Science*, volume 277, pages 47–103. Elsevier, 2002.

[CP02] P. Charpin and E. Pasalic. On propagation characteristics of resilient functions. In *Selected Areas in Cryptography*, volume 2595, pages 175–195. Springer, 2002.

[CR06] B. Cogswell and M. Russinovich. SysInternals RootkitRevealer, 2006.

[CT91] T.M. Cover and J.A. Thomas. *Elements of information theory*. John Wiley & Sons, New York, 1991.

[CTL97] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. *University of Auckland Technical Report*, 170, 1997.

[CW00] D.M. Chess and S.R. White. An undetectable computer virus. In *Proceedings of the 2000 Virus Bulletin Conference (VB2000)*, 2000.

[Dat05] DataRescue. Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executables. http ://www.datarescue.com/idabase/, 2005.

[Dat06] DataRescue. Using the IDA debugger to unpack an hostile PE executable. http ://www.datarescue.com/idabase/, 2006.

[Dat07] DataRescue. IDA Pro. http ://www.datarescue.com/, 2007.

[DD03] E. Detoisien and E. Dotan. Cheval de Troie furtif sous Windows : mécanismes d'injection de code. *Journal de la sécurité informatique MISC*, (10), 2003.

[DJ07] G. Dabosville and S. Josse. Protection des logiciels contre la rétro-ingénierie. In *Proceedings of the C&ESAR 2007 Conference, Chateaugiron, France, November 6-8*, 2007.

[DPCJD07] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388. ACM Press New York, NY, USA, 2007.

[DPG05] M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Lecture notes in computer science*, pages 1325–1336. Springer, 2005.

[DPMDG06] M. Dalla Preda, M. Madou, K. DeBosschere, and R. Giacobazzi. Opaque Predicates Detection by Abstract Interpretation. In *Algebraic Methodology and Software Technology*, pages 81–95. Springer, 2006.

[DR05] T. Dullien and R. Rolles. Graph-Based Comparison of Executable Objects. *Symposium sur la Securite des Technologies de l'Information et des Communications (SSTIC'05)*, 2005.

[DT91] M.H. Dawson and S.E. Tavares. An expanded set of S-box design criteria based on information theory and its relation to differential-like attacks. In *Proceedings of EUROCRYPT'91 : Advances in Cryptology*, pages 352–367, 1991.

[Duf07] L. Duflot. *Contribution à la sécurité des systèmes d'exploitation et des microprocesseurs*. PhD thesis, Université de Paris XI, 18 octobre 2007, 2007.

[DVCD07] B. DeBus, L. VanPut, D. Chanet, and B. DeSutter. Diablo Is A Better Link-time Optimizer. 2007.

[EIC06] EICAR. European Institute for Computer Anti-Virus Research . http ://www.eicar.org/, 2006.

[Eli05] Elias. Detect if your program is running inside a Virtual Machine. http ://lgwm.org (Elias homepage), 2005.

[Elk97] C. Elkan. Naive Bayesian Learning. *Technical Report CS 97-557, Department of Computer Science and Engineering, University of California, San Diego, USA*, 1997.

[Ern03] M.D. Ernst. Static and dynamic analysis : synergy and duality. *WODA 2003 : ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[eSa06] eSafe. eSafe test page. http ://www.esafe.com/home/csrt/eSafe_Demo/TestPage.asp, 2006.

[FEG+07] E. Filiol, P. Evrard, G. Geffard, G. Guilleminot, G. Jacob, S. Josse, and D. Quenez. Evaluation de l'antivirus OneCare : quand avant l'heure ce n'est pas l'heure ! *Journal de la sécurité informatique MISC, Juillet*, (32) :42–51, 2007.

[Fer06] P. Ferrie. Attacks on Virtual Machine Emulators. In *Proceedings of the 2006 AVAR Conference, Auckland, New Zealand, December 3-5*, 2006.

[FFJ04] E. Filiol, C. Fontaine, and S. Josse. The COSvd Ciphers. In *Proceedings of the ECRYPT Conference on The State of the Art of Stream Ciphers, Bruges, October 13-15*, volume 59, 2004.

[FGJ+08] E. Filiol, G. Geffard, G. Jacob, S. Josse, and D. Quenez. Evaluation de l'antivirus Dr Web : l'antivirus qui venait du froid. *Journal de la sécurité informatique MISC, Mai*, (37), 2008.

[FHZ06] E. Filiol, M. Helenius, and S. Zanero. Open Problems in Computer Virology. In *Journal in Computer Virology*, volume 1, pages 55–66. Springer, 2006.

[Fil01] E. Filiol. *Techniques de reconstruction en cryptologie et théorie des codes*. PhD thesis, Ecole polytechnique, 2001.

[Fil04] E. Filiol. *Les virus informatiques : theorie, pratique et applications*. Springer-Verlag France, 2004.

[Fil05a] E. Filiol. *Computer Viruses : From Theory to Applications*. Springer, 2005.

[Fil05b] E. Filiol. La simulabilité des tests statistiques. *Journal de la sécurité informatique MISC*, (32), 2005.

[Fil05c] E. Filiol. Strong cryptography armoured computer viruses forbidding code analysis. In *Proceedings of the 14th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, pages 216–227, 2005.

[Fil06] E. Filiol. Malware Pattern Scanning Schemes Secure Against Black-box Analysis. In *Journal in Computer Virology*, volume 2, pages 35–50. Springer, 2006.

[Fil07a] E. Filiol. Formal model proposal for (malware) program stealth. In *Proceedings of the Virus bulletin 2007 Conference*, pages 179–185, 2007.

[Fil07b] E. Filiol. Metamorphism, Formal Grammars and Undecidable Code Mutation. *International Journal of Computer Science*, 2(2) :70–75, 2007.

[Fil07c] E. Filiol. *Techniques virales avancées*. Springer, 2007.

[FJ07] E. Filiol and S. Josse. A statistical model for undecidable viral detection. In *Proceedings of the 16th EICAR Conference, Budapest, Hungary, May 5 - 8, 2007, & In Journal in Computer Virology*, volume 3, pages 65–74. Springer, 2007.

[FJL07] E. Filiol, G. Jacob, and M.L. Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. In *Journal in Computer Virology*, volume 3, pages 23–37. Springer, 2007.

[FO04] P. Frej and M. Ogorkiewicz. Analysis of Buffer Overflow Attacks. 2004.

[Foc06] Security Focus. Bugtraq . 2006.

[Fon98] C. Fontaine. *Contribution à la recherche de fonctions booléennes hautement non linéaires et au marquage d'images en vue de la protection des droits d'auteur*. PhD thesis, 1998.

[Fou02] D. Fourdrinier. Statistique inférentielle. *Sciences Sup, Dunod*, 2002.

[FS06] F-Secure. BlackLight. http ://www.f-secure.com/blacklight/, 2006.

[fuz06] fuzen_op. Fu rootkit. https ://www.rootkit.com/vault/fuzen_op/, 2006.

[Gar06] G. Garner. Forensic Acquisition Utilities, 2006.

[Gaz08] A. Gazet. Comparative analysis of various ransomware virii. In *Proceedings of the 17th EICAR Conference, 4-6 May 2008, ESIEA, Laval, France*, 2008.

[GCS06] A. Grocholewska-CzuryJo and J. Stoklosa. Random Generation of S-Boxes for Block Ciphers. Springer, 2006.

[GG09] Y. Guillot and A. Gazet. Désobfuscation automatique de binaire - The Barbarian Sublimation. *Symposium sur la Securite des Technologies de l'Information et des Communications (SSTIC'09)*, 2009.

[Gha98] Z. Ghahramani. Learning dynamic Bayesian networks. In *Adaptive Processing of Sequences and Data Structures*, volume 1387, pages 168–197. Springer, 1998.

[GM06] G. Garner and R. Mora. Kntlist tool. http ://www.dfrws.org/2005/challenge/kntlist.html, 2006.

[GMQ07] L. Goubin, J.M. Masereel, and M. Quisquater. Cryptanalysis of white box DES implementations. In *Proceedings of the 14th Annual Workshop on Selected Areas in Cryptography*. Springer, 2007.

[Gri06] GriYo. Entry-Point Obscuring. http ://vx.netlux.org/lib/vgy01.html, 2006.

[Gui07] Y. Guillot. Metasm. *Symposium sur la Securite des Technologies de l'Information et des Communications (SSTIC'07)*, 2007.

[Haz03] P. Hazel. PCRE : Perl Compatible Regular Expressions. http ://www.pcre.org, 2003.

[Hea06a] J. Heasman. Implementing and detecting a PCI rootkit, 2006.

[Hea06b] J. Heasman. Implementing and Detecting an ACPI BIOS Root Kit. 2006.

[Hea08] VX Heavens. Virus Collection . http ://vx.netlux.org/, 2008.

[Her03] D.S. Herrmann. *Using the Common Criteria for IT Security Evaluation*. Auerbach Pub, 2003.

[Hog05] G. Hoglund. *Rootkits : Subverting the Windows Kernel*. Addison-Wesley, ISBN 0-321-29431-9, 2005.

[Hog06] G. Hoglund. NT Rootkit. https ://www.rootkit.com/vault/hoglund/, 2006.

[Hor97] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 19, pages 1–6. ACM Press New York, NY, USA, 1997.

[Ice06] IceSword. IceSword. http ://xfocus.net/tools/200509/1085.html, 2006.

[JBF03] M. Jacob, D. Boneh, and E. Felten. Attacking an obfuscated cipher by injecting faults. In *Proceedings of ACM CCS-9 Workshop (LNCS 2696) : Digital Rights Management*, pages 16–31. Springer-Verlag, 2003.

[JFD08] G. Jacob, E. Filiol, and H. Debar. Functional Polymorphic Engines : Formalisation, Implementation and Uses Cases. In *Proceedings of the EICAR 2008 Conference, Laval, 3-6 Mai*, 2008.

[Jos05] S. Josse. Techniques d'obfuscation de code : chiffrer du clair avec du clair. *Journal de la sécurité informatique MISC*, (20) :32–42, 2005.

[Jos06a] S. Josse. How to Assess the Effectiveness of your Anti-virus ? In *Proceedings of the 15th EICAR Conference, Hamburg, Germany, April 29-May 3, 2006, & In Journal in Computer Virology*, volume 2, pages 51–65. Springer, 2006.

[Jos06b] S. Josse. Secure and advanced unpacking using computer emulation. In *Proceedings of the AVAR 2006 Conference, Auckland, New Zealand, December 3-5*, pages 174–190, 2006. & In *Journal in Computer Virology*, volume 3, pages 221-236. Springer, 2007.

[Jos07a] S. Josse. Rootkit detection from outside the Matrix. In *Proceedings of the 16th EICAR Conference, Budapest, Hungary, May 5 - 8, 2007, & In Journal in Computer Virology*, volume 3, pages 113–123. Springer, 2007.

[Jos07b] S. Josse. VxStripper, Virus (and other armored software) reverse engineering tool. http ://pagesperso-orange.fr/vxstripper/, 2007.

[Jos08] S. Josse. White Box Attack Context Cryptovirology. In *Proceedings of the 17th EICAR Conference, Laval, France, May 3-6*, 2008.

[K+07] T. Kojm et al. Clam AntiVirus. 2007.

[Kil73] G.A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press New York, NY, USA, 1973.

[KPY07] M.G. Kang, P. Poosankam, and H. Yin. Renovo : a hidden code extractor for packed executables. *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53, 2007.

[KSS97] Y. Kaneko, F. Sano, and K. Sakurai. On provable security against differential and linear cryptanalysis in generalized Feistel ciphers with multiple random functions. *Proc. of SAC'97*, pages 185–199, 1997.

[KU77] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. In *Acta Informatica*, volume 7, pages 305–317. Springer, 1977.

[KW03] C. Karlof and D. Wagner. Hidden Markov Model Cryptanalysis. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*, volume 2779, pages 17–34. Springer-Verlag LNCS, 2003.

[L+01] K. Lawton et al. Plex86. 2001.

[lab06a] Virus Bulletin lab. http ://www.virusbtn.com/, 2006.

[Lab06b] ICSA Labs. International Computer Security Association Labs . https ://www.icsalabs.com/, 2006.

[Lab06c] West Coast Labs. . http ://www.westcoastlabs.org/, 2006.

[Lan92] W.A. Landi. Interprocedural aliasing in the presence of pointers. Rutgers University New Brunswick, NJ, USA, 1992.

[Lau08] C. Lauradoux. Detecting Virtual Rootkits with Cover Channels. In *Proceedings of the 17th EICAR Conference, Laval, France, May 3-6*, 2008.

[Law07] K. Lawton. Bochs : The Open Source IA-32 Emulation Project. http ://bochs.sourceforge.net, 2007.

[LC96] H.H.C. Law and C. Chan. N-thorder Ergodic Multigram HMM for modeling of languages without marked word boundaries. In *Proceedings of the 16th conference on Computational linguistics-Volume 1*, pages 204–209. Association for Computational Linguistics Morristown, NJ, USA, 1996.

[LD03] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM Press New York, NY, USA, 2003.

[LJC07] S. Lee, I. Jeong, and S. Choi. Dynamically Weighted Hidden Markov Model for Spam Deobfuscation. *Int'l Conf. Artificial Neural Networks (IJCAI-2007), Hyderabad, India, January 6-12*, pages 2523–2529, 2007.

[LN05a] H. Lee and A.Y. Ng. Spam Deobfuscation using a Hidden Markov Model. *Conference on Email and Anti-Spam*, 2005.

[LN05b] HE Link and WD Neumann. Clarifying obfuscation : improving the security of white-box DES. *International Conference on Information Technology : Coding and Computing, 2005. ITCC 2005.*, 1, 2005.

[LPS04] B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *Proceedings of Eurocrypt*, volume 2004. Springer, 2004.

[LSD02] LSD. The Last Stage of Delirium Research Group, Win32 Assembly Components, 2002.

[Luc00] S. Lucas. Observable Semantics and Dynamic Analysis of Computational Processes. 2000.

[MA96] S. Mister and C. Adams. Practical S-Box Design. *Workshop on Selected Areas in Cryptography, SAC*, 96 :61–76, 1996.

[Mac06] MackT. ImportREC. http ://mackt.cjb.net/, 2006.

[MADD05] M. Madou, B. Anckaert, B. DeSutter, and K. DeBosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital rights management*, pages 75–82. ACM Press New York, NY, USA, 2005.

[MAM+05] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. DeSutter, and K. DeBosschere. Software protection through dynamic code mutation. In *Proceedings of the 6th International Workshop on Information Security Applications (WISA*, pages 194–206. Springer, 2005.

[McA07] McAfee. Virusscan 8.5i and epolicy orchestrator 3.6.1 security target, version 6, june 2007, 2007.

[MDD+08] J.N. Matthews, E.M. Dow, T. Deshane, W. Hu, J. Bongio, P.F. Wilbur, and B. Johnson. *Running Xen : A Hands-On Guide to the Art of Virtualization.* Prentice Hall PTR Upper Saddle River, NJ, USA, 2008.

[MDWZ04] Y. Miretskiy, A. Das, C.P. Wright, and E. Zadok. Avfs : An On-Access Anti-Virus File System. In *Proceedings of the 13th USENIX Security Symposium (Security 2004)*, pages 73–88, 2004.

[Mic06a] Microsoft. IFSKit, Installable File System Kit. http ://www.microsoft.com/whdc/devtools/ifskit/, 2006.

[Mic06b] Microsoft. Kernel Patch Protection : Frequently Asked Questions. http ://www.microsoft.com/whdc/driver/kernel/64bitpatch_FAQ.mspx, 2006.

[Mic06c] Microsoft. Microsoft Portable Executable and Common Object File Format Specification, revision 8.0. http ://msdn.microsoft.com/, 2006.

[Mis03] P. Mishra. A taxonomy of software uniqueness transformations. Master's thesis, Department of Computer Science, San Jose State University, 2003.

[MKK07] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. *Proc. IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[MM07] B. Morin and L. Mé. Intrusion detection and virology : an analysis of differences, similarities and complementariness. In *Journal in Computer Virology*, volume 3, pages 39–49. Springer, 2007.

[MN98] A. McCallum and K. Nigam. A comparison of event models for naive bayes text classification. *AAAI-98 Workshop on Learning for Text Categorization*, 752, 1998.

[MOR01] M. Muller-Olm and O. Ruthing. The Complexity of Constant Propagation. In *Proceedings of the 10th European Symposium on Programming (ESOP)*, pages 190–205. Springer, 2001.

[Muc97] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Mur97] F. Muri. *Comparaison d'algorithmes d'identification de chaînes de Markov cachées et application à la détection de régions homogènes dans les séquences d'ADN*. PhD thesis, 1997.

[MVD06a] M. Madou, L. VanPut, and K. DeBosschere. LOCO : an interactive code (De) obfuscation tool. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 140–144. ACM Press New York, NY, USA, 2006.

[MVD06b] M. Madou, L. VanPut, and K. DeBosschere. Understanding Obfuscated Code. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)-Volume 00*, pages 268–274. IEEE Computer Society Washington, DC, USA, 2006.

[Neb00] G. Nebbett. *Windows Nt/2000 Native Api Reference*. Sams, 2000.

[New06] J. Newbigin. Dd for Windows. http ://uranus.it.swin.edu.au/ jn/linux/rawwrite/dd.htm, 2006.

[NIS01] NIST. FIPS 197 : Announcing the advanced encryption standard (AES). Technical report, Technical report, National Institute of Standards and Technology (NIST), 2001.

[Nor07] Norman. SandBox Malware analyzer. http ://www.norman.com/microsites/malwareanalyzer/, 2007.

[OM06] M.F. Oberhumer and L. Molnar. The Ultimate Packer for eXecutables (UPX). 2006.

[Ope07] org. Openrce. Ollydbg Plugins . http ://www.openrce.org/download/browse/OllydbgPlugins/, 2007.

[OSSM02] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis. *3rd Workshop on Information Security Applications (WISA 2002), Korea, August*, 2002.

[OSSM03] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software Obfuscation on a Theoretical Basis and Its Implementation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 86(1) :176–186, 2003.

[Pap94] C.H. Papadimitriou. *Computational complexity*. Addison-Wesley Reading, Mass, 1994.

[PEi07] PEiD. PE iDentifier. http ://peid.tk, 2007.

[Pen02a] A. Pennell. Minidumps tool, 2002.

[Pen02b] A. Pennell. Post-Mortem Debugging Your Application with Minidumps and Visual Studio .NET, 2002.

[PJ99] E. Pasalic and T. Johansson. Further results on the relation between nonlinearity and resiliency for Boolean functions. In *Proceedings of IMA Conference on Cryptography and Coding*, pages 35–45. Springer, 1999.

[PJFMA04] N.L. Petroni Jr, T. Fraser, J. Molina, and W.A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13 table of contents*, pages 13–13. USENIX Association Berkeley, CA, USA, 2004.

[Pla01] V. Plagnol. Detection de Génes par Modéles de Chaînes de Markov Cachées. Master's thesis, 2001.

[Por04] G. Portokalidis. *Zero Hour Worm Detection and Containment using Honeypots*. PhD thesis, Leiden University, 2004.

[PP] Protection profile.

[PSB06a] G. Portokalidis, A. Slowinska, and H. Bos. Argos : an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of the 2006 EuroSys conference*, pages 15–27. ACM Press New York, NY, USA, 2006.

[PSB06b] G. Portokalidis, A. Slowinska, and H. Bos. Argos project. https ://gforce.cs.vu.nl/projects/argos/, http ://www.few.vu.nl/argos/, 2006.

[PSB06c] G. Portokalidis, A. Slowinska, and H. Bos. Howto setting up argos the 0day shellcode catcher. http ://www.few.vu.nl/argos/, 2006.

[PW05] B. Preneel and B. Wyseur. Condensed white-box implementations. In *Proceedings of the 26th Symposium on Information Theory in the Benelux, Brussels,Belgium*, pages 296–301, 2005.

[Qoz99] Qozah. Polymorphism and grammars. 29A E-zine, http ://www.29a.net/., 1999.

[Rab89] LR Rabiner. A tutorial on hidden Markov models and selected applications inspeech recognition. In *Proceedings of the IEEE*, volume 77, pages 257–286, 1989.

[Rey04] J.F. Rey. Calculabilité, complexité et approximation. 2004.

[RHJ01] I. Rish, J. Hellerstein, and T. Jayram. An analysis of data characteristics that affect naive Bayes performance. In *Proceedings of the Eighteenth Conference on Machine Learning-ICML2001*. Morgan Kaufmann, 2001.

[RI00] J.S. Robin and C.E. 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 table of contents*, pages 10–10. USENIX Association Berkeley, CA, USA, 2000.

[RJ87] H. Rogers Jr. *Theory of recursive functions and effective computability*. MIT Press Cambridge, MA, USA, 1987.

[RKLC03] J.C. Rabek, R.I. Khazan, S.M. Lewandowski, and R.K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 76–82. ACM New York, NY, USA, 2003.

[roo07] rootkit.com. BootKit. http ://www.rootkit.com/vault/vipinkumar/, 2007.

[RS98] J. Riordan and B. Schneier. Environmental Key Generation Towards Clueless Agents. In *Mobile Agents and Security*, pages 15–24. Springer, 1998.

[RS05] M.E. Russinovich and D.A. Solomon. *Microsoft Windows internals : Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, Redmond, Wash., 2005.

[Run08] Runtime's Disk Explorer for NTFS. http ://www.runtime.org/diskexpl.htm, 2008.

[Rut05a] J. Rutkowska. Detecting Windows Server Compromises with Patchfinder 2. http ://www.invisiblethings.org/papers/rootkits_detection_with_patchfinder2.pdf, 2005.

[Rut05b] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. 2005.

[Rut05c] J. Rutkowska. System virginity verifier : Defining the roadmap for malware detection on windows systems. *Hack In The Box Security Conference, September*, 2005.

[Rut06] J. Rutkowska. Subverting Vista Kernel For Fun And Profit. *Blackhat 2006*, 2006.

[SAB07a] SABRE. BinDiff Users Manual. sabre-security.com, 2007.

[SAB07b] SABRE. Using SABRE BinDiff for Malware analysis. sabre-security.com, 2007.

[Sar02] P. Sarkar. Cross-Correlation Analysis of Cryptographically Useful Boolean Functions and S-Boxes. In *Theory of Computing Systems*, volume 35, pages 39–57. Springer, 2002.

[SC07] Y. Smaragdakis and C. Csallner. Combining Static and Dynamic Reasoning for Bug Detection. In *Proceedings of the 1st International Conference on Tests And Proofs (TAP)*, pages 1–16. Springer, 2007.

[Sch06a] A. Schuster. Acquisition : dd. Personal weblog http ://computer.forensikblog.de/, 2006.

[Sch06b] A. Schuster. Adapting PTfinder to other Versions of Microsoft Windows. Personal weblog http ://computer.forensikblog.de/, 2006.

[Sch06c] A. Schuster. Converting Virtual into Physical Addresses. Personal weblog http ://computer.forensikblog.de/, 2006.

[Sch06d] A. Schuster. Improving list-walkers. Personal weblog http ://computer.forensikblog.de/, 2006.

[Sch06e] A. Schuster. More on Processes and Threads. Personal weblog http ://computer.forensikblog.de/, 2006.

[Sch06f] A. Schuster. Reconstructing a Binary (parts 1 and 2). Personal weblog http ://computer.forensikblog.de/, 2006.

[Sch06g] A. Schuster. Searching for Processes and Threads. Personal weblog http ://computer.forensikblog.de/, 2006.

[Sch06h] A. Schuster. Tool MemDump.PL (PERL script). Personal weblog http ://computer.forensikblog.de/, 2006.

[Sch06i] A. Schuster. Tool PTFinder.PL (Find Processes and Threads in a Microsoft Windows memory dump, PERL script). Personal weblog http ://computer.forensikblog.de/, 2006.

[SEZS01] M.G. Schultz, E. Eskin, E. Zadok, and S.J. Stolfo. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, volume 49. Los Alamitos, CA, 2001.

[Sha49] C.E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4) :656–715, 1949.

[SHG04] A. Shafieinezhad, F. Hendessi, and TA Gulliver. A structure for fast data encryption. *Journal of Contemp. Math. Sciences*, 2(29-32) :1401–1424, 2004.

[SIG06] SIG∧2. KprocCheck. http ://www.security.org.sg/, 2006.

[Soa87] R.I. Soare. *Recursively Enumerable Sets and Degrees : A Study of Computable Functions and Computably Generated Sets*. Springer, 1987.

[SOL07] SOLAR. Software Optimization at Link-Time and Run-Time. http ://www.cs.arizona.edu/solar/, 2007.

[SP05] D. Soeder and R. Permeh. eEye BootRoot : A Basis for Bootstrap-Based Windows Kernel Code. *BlackHat USA*, 2005.

[Spi03] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. *Information Theory, IEEE Transactions on*, 49(1) :280–284, 2003.

[SR00] D.A. Solomon and M. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press Redmond, WA, USA, 2000.

[SRT08] Digital River Silicon Realms Toolworks. Armadillo SoftwarePassport. http ://www.siliconrealms.com, 2008.

[Sta04] M. Stamp. A Revealing Introduction to Hidden Markov Models. *Department of Computer Science San Jose State University*, 2004.

[Ste05] A.E. Stepan. Defeating Polymorphism : Beyond Emulation. In *Proceedings of the Virus Bulletin International Conference*, 2005.

[SW49] C. Shannon and W. Weaver. The mathematical theory of communication. University of Illinois. *Urbana*, 117, 1949.

[SWT01] D.X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *Proceedings of the 10th conference on USENIX Security Symposium, Washington DC, USA*. USENIX Association Berkeley, CA, USA, 2001.

[SXCM04] AH Sung, J. Xu, P. Chavez, and S. Mukkamala. Static analyzer of vicious executables (SAVE). *Computer Security Applications Conference, 2004. 20th Annual*, pages 326–334, 2004.

[Szö05] P. Ször. *The Art of Computer Virus Research and Defense*. Addison Wesley for Symantec Press, 2005.

[tea06] AV-Test team. Av-test.org project. http ://www.av-test.org/, 2006.

[Tro04] J. Troger. *Specification-Driven Dynamic Binary Translation*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2004.

[Tsa04] T.T. Tsai. Hidden Markov Model for text analysis, 2004.

[UDM05] SK Udupa, SK Debray, and M. Madou. Deobfuscation : Reverse Engineering Obfuscated Code. *Reverse Engineering, 12th Working Conference on*, pages 45–54, 2005.

[UNR+05] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kägi, F.H. Leung, and L. Smith. Intel Virtualization Technology. IEEE Computer Society, 2005.

[VDM+05] L. VanPut, B. DeSutter, M. Madou, B. DeBus, D. Chanet, K. Smits, and K. DeBosschere. LANCET : a nifty code editing tool. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering*, pages 75–81. ACM Press New York, NY, USA, 2005.

[Vid05] M. Videau. *Critères de sécurité des algorithmes de chiffrement à clé secrète*. PhD thesis, Université de Paris 6, 10 Nov., 2005.

[Vig06a] Vigil@nce. Incorrect analysis of LHA files. BUGTRAQ-10243, CVE-2004-0234, CVE-2004-0235. http ://vigilance.aql.fr/, 2006.

[Vig06b] Vigil@nce. Incorrect analysis of MIME messages. BUGTRAQ-9650, CVE-2004-2088. http ://vigilance.aql.fr/, 2006.

[Vig06c] Vigil@nce. Incorrect analysis of RAR files. BUGTRAQ-13416, CVE-2005-1346. http ://vigilance.aql.fr/, 2006.

[Vig06d] Vigil@nce. Incorrect analysis of the data integrated into a URI. BUGTRAQ-12269, CVE-2005-0218. http ://vigilance.aql.fr/, 2006.

[Vig06e] Vigil@nce. Incorrect analysis of ZIP files when they are protected by a password or have several levels of overlap. BUGTRAQ-11600, BUGTRAQ-11669, BUGTRAQ-11732, CVE-2004-2220, CVE-2004-2442. http ://vigilance.aql.fr/, 2006.

[Vig06f] Vigil@nce. Incorrect management of the files containing ANSI escape characters (these sequences can disturb display during the consultation of the audit files by the administrator). BUGTRAQ-12793. http ://vigilance.aql.fr/, 2006.

[Vig06g] Vigil@nce. Incorrect Unicode support. BUGTRAQ-10164. http ://vigilance.aql.fr/, 2006.

[Vig06h] Vigil@nce. No disinfection of ZIP file. BUGTRAQ-11448, CVE-2004-0932-0937, CVE-2004-1096. http ://vigilance.aql.fr/, 2006.

[Vig06i] Vigil@nce. Outlook accepts messages whose format does not respect the RFC 822 (Standard for the format of ARPA Internet text messages). BUGTRAQ-5259, CVE-2002-0637. http ://vigilance.aql.fr/, 2006.

[Vis01] E. Visser. A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57(2), 2001.

[VMw06] VMware. VMware. http ://www.vmware.com/, 2006.

[VMw07] VMware. VMware ACE. http ://www.vmware.com/products/ace/, 2007.

[vO03] P.C. van Oorschot. Revisiting software protection. In *Proceedings of the 6th International Information Security Conference (ISC)*. Springer, 2003.

[VY06] A. Vasudevan and R. Yerraballi. Cobra : Fine-grained Malware Analysis using Stealth Localized-Executions. *IEEE Symposium on Security and Privacy*, 2006.

[WBV+05] Y.M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. *Proc. DSN*, 2005.

[WDHK01] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. *International Conference of Dependable Systems and Networks*, 2001.

[Wea03a] Weariless. MDump tool. http ://www.codeproject.com/, 2003.

[Wea03b] Weariless. Performing a hex dump of another process's memory. http ://www.codeproject.com/, 2003.

[WHF07] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. In *IEEE Security & Privacy*, pages 32–39. IEEE Computer Society, 2007.

[WMGP07] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In *Proceedings of the 14th Annual Workshop on Selected Areas in Cryptography*. Springer, 2007.

[Won06] W. Wong. Analysis and detection of metamorphic computer viruses. Master's thesis, Department of Computer Science, San Jose State University, 2006.

[WS06] W. Wong and M. Stamp. Hunting for metamorphic engines. In *Journal in Computer Virology*, volume 2, pages 211–229. Springer, 2006.

[WT86] AF Webster and SE Tavares. On the design of S-boxes. In *Lecture notes in computer sciences on Advances in cryptology—CRYPTO'85*, volume 218, pages 523–534. Springer-Verlag New York, Inc. New York, NY, USA, 1986.

[XNHA05] Y. Xie, M. Naik, B. Hackett, and A. Aiken. Soundness and its Role in Bug Detection Systems. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools. Computer Science Department of the Stanford University, ACM Press, June*, 2005.

[Y0d06] Y0da. Y0da's packer. http ://y0da.cjb.net/, 2006.

[Y0d07] Y0da. Y0da's LordPE. http ://y0da.cjb.net/, 2007.

[Yus07] O. Yuschuk. Ollydbg. http ://www.ollydbg.de/, 2007.

[YY96] A. Young and M. Yung. Cryptovirology : Extortion-Based Security Threats and Countermeasures. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 129–141. IEEE Computer Society Press, 1996.

[YY04] A. Young and M. Yung. *Malicious cryptography : exposing cryptovirology*. John Wiley & Sons, 2004.

[Z0m00a] Z0mbie. About Permutation (RPME). http ://vx.netlux.org, 2000.

[Z0m00b] Z0mbie. Automated reverse engineering : Mistfall engine. http ://vx.netlux.org, 2000.

[Z0m00c] Z0mbie. VMWare has you. http ://vx.netlux.org, 2000.

[Zei05] A. Zeichick. Coming Soon to VMware, Microsoft, and Xen : AMD Virtualization Technology Solves Virtualization Challenges. http ://www.devx.com/amd/Article/30186/, 2005.

[Zov06] D.D. Zovi. Hardware Virtualization Based Rootkits. *Black Hat USA, Aug*, 2006.

[ZR94] S. Zhang and B.G. Ryder. Complexity of single level function pointer aliasing analysis. Technical report, Technical Report LCSR-TR-233, Laboratory of Computer Science Research, Rutgers University, October 1994, 1994.

[ZZ04] Z. Zuo and M. Zhou. Some Further Theoretical Results about Computer Viruses. In *The Computer Journal*, volume 47, pages 627–633. Br Computer Soc, 2004.

[ZZZ05] Z. Zuo, Q. Zhu, and M. Zhou. On the time complexity of computer viruses. *Information Theory, IEEE Transactions on*, 51(8) :2962–2966, 2005.

# Glossaire

**ACPI** La norme ACPI (*Advanced Configuration and Power Interface*) succède à la norme APM. Elle permet une gestion efficace de la consommation d'un ordinateur, en limitant, par exemple, l'alimentation des composants inactifs. . 128

**Adresse logique** Tout programme logiciel s'exécutant en mode protégé sur une machine x86 ne manipule exclusivement que des adresses logiques. Ces adresses logiques sont traduites par les mécanismes de segmentation et de pagination successivement en adresses virtuelles puis en adresses physiques. Ces dernières sont manipulées par l'ensemble des composants de la carte mère. . 123

**Adresse physique** Les adresses physiques sont manipulées par l'ensemble des composants de la carte mère (processeur, chipset, périphérique) mais pas par les composants logiciels qui s'exécutent sur le processeur. L'espace d'adressage physique comprend la mémoire principale ainsi que l'ensemble de la mémoire des périphériques accessible en MMIO. . 123

**API** Une interface de programmation (*Application Programming Interface*) est un ensemble de bibliothèques de fonctions utilisables pour développer des applications. À titre d'exemple, l'API Windows est constituée d'un ensemble de bibliothèques et fonctions fournies par le système d'exploitation et permettant la communication avec le noyau. . 46

**APM** La norme APM (*Advanced Power Management*) permet une gestion de l'alimentation du système par le BIOS ou le système d'exploitation. Bien que cette norme soit toujours supportée par la plupart des systèmes d'exploitation, elle est progressivement remplacée par la norme ACPI. . 128

**BIOS** Le BIOS (*Basic input/Output System*) est un microprogramme (firmware) qui s'interface directement avec le matériel pour exécuter certaines tâches d'entrée/sortie, ainsi que d'autres fonctions de bas niveau. le BIOS est le composant responsable de la configuration initiale du système. À titre d'exemple, la routine POST (*Power On Self Test*) du BIOS est le premier composant à s'exécuter lors du démarrage du système. Une fois que le POST a configuré les différents services souhaités, le BIOS passe la main au chargeur de démarrage qui permettra de sélectionner le système d'exploitation à démarrer. Historiquement, les systèmes d'exploitation faisaient appel à des routines du BIOS en cours de fonctionnement pour configurer des périphériques ou communiquer avec eux, mais ce n'est aujourd'hui plus le cas dans les machines modernes. Le BIOS ne sert généralement qu'au démarrage du système. . 123

**Chipset** Le chipset est un composant de la carte mère responsable de la gestion mémoire au niveau système, et de la gestion des périphériques. Il est à ce titre le correspondant privilégié du processeur. Il a par exemple la charge (via son sous-composant *northbridge*) d'effectuer la correspondance entre une adresse reçue et le périphérique MMIO auquel cette adresse est associée. Il est responsable (via son sous-composant *southbridge*) de la gestion des principaux bus du système : bus PCI, IDE, LPC, AGP, PCI-express, etc. . 107

**Commutation de tâches** En mode protégé, l'exécution d'un programme se fait à l'intérieur du contexte d'une tâche (la tâche courante). Le sélecteur du segment TSS de la tâche courante est stocké dans le registre TR. Une commutation de tâche survient typiquement lors d'un appel ou d'un saut vers une nouvelle tâche. Lors d'une commutation de tâche, le processeur effectue les opérations suivantes:

1. il stocke l'état de la tâche courante dans le TSS courant;

2. il charge le sélecteur du segment de la nouvelle tâche dans le registre TR;
3. il accède au nouveau TSS via un descripteur de segment de la table GDT;
4. il charge l'état de la nouvelle tâche à partir du nouveau segment TSS dans les registres généraux, les registres de segment, le registre LDTR, le registre de contrôle CR3, le registre EFLAGS, et le registre EIP;
5. il exécute la nouvelle tâche.

. 274

**Complétude** Un schéma de détection est dit complet s'il permet de détecter toutes les variantes d'un programme viral. Remarquons qu'un schéma de détection peut être à la fois complet et incorrect. . 4

**Confluence** Un ensemble de règles de transformation pour un programme induit une relation de réécriture sur les programmes. On s'intéresse aux transformations de programme vues comme l'application successive de réécritures $\rightarrow$ sur un objet syntaxique tel qu'un mot, un terme ou un graphe. Un système de réécriture est un ensemble de règles de réécritures s'appliquant à l'objet syntaxique. Un système de réécriture de mots s'applique aux objets syntaxiques que sont les mots construits à partir d'un alphabet. Un terme est une représentation symbolique de la structure qui est transformée. Un système de réécriture de termes s'applique aux objets syntaxiques que sont les expressions construites à partir d'une signature constituée de constantes et de symboles d'opérations. Si on considère la relation d'équivalence $\equiv^*$ associée au pré-ordre (relation réflexive et transitive, mais pas antisymétrique) $\rightarrow^*$, l'ensemble quotient $Sobj/\equiv^*$ représente les différents états atteignables lors de la transformation. Les formes normales de la relation $\rightarrow$ définissent des classes d'équivalence réduites à des singletons.

Une relation de réécriture $\rightarrow$ est dite confluente lorsque :

$$t \rightarrow^* t' \text{ et } t \rightarrow^* t'' \Rightarrow \exists s \text{ tel que } t' \rightarrow^* s \text{ et } t'' \rightarrow^* s.$$

. 2

**Conformité** Le contrôle de conformité d'une fonction (ou d'un mécanisme) de sécurité consiste à vérifier qu'elle (il) se comporte suivant ses spécifications. Ce contrôle est mené en repassant les tests du développeur et en mettant en oeuvre des tests complémentaires. L'objectif est de valider la démarche de test du développeur.
Le contrôle de conformité de fournitures vise à vérifier ou confirmer qu'elles satisfont aux exigences sur le contenu et la présentation des éléments de preuve. . 8

**Correction** Un schéma de détection est dit correct s'il ne détecte que les variantes d'un programme viral. Il ne considérera donc pas à tort un programme bénin comme malicieux. Remarquons qu'un schéma de détection peut être correct mais incomplet. . 4

**CPL** Le CPL (*Current Processor Level*) est le niveau de privilège du programme en cours d'exécution. Le CPL est stocké dans les deux premiers bits des registres de segment CS et SS. . 53

**CR3** Le registre de contrôle 3 (CR3) contient l'adresse physique de l'annuaire de page du processus courant, ainsi qu'un certain nombre de flags indiquant entre autres le type de page. . 123

**CSIM** Une CSIM (*Complete Software Interpreter Machine*) ou émulateur utilise uniquement l'interprétation logicielle (toutes les instructions CPU sont émulées par un programme logiciel). . 106

**DKOM** Le noyau utilise des objets pour représenter les différentes ressources. Il est possible, en modifiant directement en mémoire certains de ces objets (*Direct Kernel Object Manipulation*), de modifier le comportement du système. Par exemple, le système maintient plusieurs listes chaînées qu'il est possible de modifier afin de cacher des processus, des drivers ou autres ressources. Un rootkit peut retirer une ou plusieurs entrées de ces listes en modifiant le chaînage. Le système continue à fonctionner normalement, mais certaines méthodes documentées pour afficher les processus ou les drivers n'affichent plus les noeuds cachés. Il est aussi possible d'élever les privilèges d'un ou plusieurs processus en modifiant directement l'objet décrivant le processus (structure EPROCESS). En effet, cette structure contient entre autres le jeton du processus regroupant les privilèges associés. . 125

**DPL** Le DPL (*Descriptor Privilege Level*) est le niveau de privilège d'un segment ou d'une porte (gate). Le DPL est stocké dans le descripteur de segment ou de porte. . 124

**Efficacité, robustesse** Le contrôle de robustesse ou d'efficacité d'une fonction (ou d'un mécanisme) de sécurité consiste à effectuer des tests d'intrusion mettant en oeuvre des menaces ou enfreignant la politique de sécurité en exploitant des vulnérabilités. L'objectif est de mener une analyse indépendante de vulnérabilité permettant de confirmer, par une analyse indépendante, le potentiel minimum d'attaque nécessaire pour mettre en oeuvre les menaces / enfreindre les règles de politique de sécurité, tel que l'a estimé le développeur. . 1, 6

**ELF** Le format ELF (*Execute and Link Format*) est le format binaire utilisé par la plupart des systèmes Linux et UNIX aujourd'hui. . 77

**Exception** Evènement synchrone qui résulte de l'exécution d'une instruction particulière. Exécuter le programme une seconde fois avec les mêmes données sous les mêmes conditions peut reproduire l'exception. Les exceptions sont typiquement déclenchées par un accès illicite en mémoire, certaines instructions de déboguage ou une erreur comme la division par zéro. Les exceptions peuvent être logicielles ou matérielles. . 108

**Forensique** L'analyse forensique désigne l'analyse d'un système après incident. Elle est effectuée par exemple après l'attaque d'un virus ou après intrusion d'un rootkit. Les méthodes et techniques employées évitent dans la mesure du possible l'utilisation des API haut niveau du système compromis. . 121, 122

**Forme Algébrique Normale** L'ANF (*Algebraïc Normal Form*) désigne la représentation d'une fonction booléenne sous sa forme algébrique normale, c'est-à-dire de manière formelle sous la forme d'un polynôme à plusieurs variables. . 41

**GDT** La GDT (table des descripteurs de segment) est une structure mémoire dont l'adresse de base est spécifiée dans le registre processeur GDTR. Elle décrit des zones de la mémoire logique (les segments) et leurs propriétés (adresse de base en mémoire, taille, restrictions d'accès en fonction des privilèges processeurs, restrictions d'accès en lecture, écriture ou exécution). Cette table des descripteurs de segment peut également contenir des Task Gates ou des Selecteurs de tâche (TSS) qui peuvent permettre un ordonnancement des tâches au niveau matériel, et des Call Gates qui peuvent permettre un changement de privilèges processeur si elles sont utilisées. . 123

**Graphe CFG** Le graphe CFG (*Control Flow Graph*) est la représentation sous forme d'un graphe orienté $G = (N, E)$ des blocs de base (*basic blocks*) d'une procédure. . 7, 54

**Graphe PCG** Le graphe PCG (*Program Control Graph*) d'un programme, appelé parfois aussi graphe ICFG (*Inter-Procedural Control Graph*) ou simplement graphe des appels, est un graphe orienté comportant autant de noeuds que de procédures. Les noeuds du graphe PCG sont appelés graphes (*Control Flow Graph*). Le graphe PCG est donc un graphe de graphe. . 8, 54

**HAL** La couche d'abstraction du matériel (*Hardware Abstraction Layer* - HAL) est une couche de code isolant le noyau, les pilotes de périphériques et le reste de l'exécutif Windows des différences matérielles sous jacentes liées à l'architecture matérielle. . 123

**HVM** Une HVM (*Hybrid VM*) interprète chaque instruction privilégiée par le biais du logiciel. . 106, 274

**IAT** La table des imports (*import Address Table*) est une table de structures (une structure par DLL à importer). Chaque structure contient le nom de la DLL ainsi qu'une table de pointeurs de fonctions. Chaque fonction importée possède une entrée dans cette table qui sera remplie par le chargeur d'exécutable du système d'exploitation avec l'adresse virtuelle de la fonction en mémoire. Une fois qu'un module est chargé, l'IAT contient les adresses qui seront utilisées pour appeler les fonctions externes. Quel que soit le nombre d'appels à une fonction externe, tous ceux-ci passeront par un seul pointeur dans l'IAT. . 94

**IDMEF** Le format IDMEF (*Intrusion Detection Message Exchange Format*)... . 109

**IDS** Un système IDS (*Intrusion Detection System*) tire partie des données journalisées par le réseau, le système ou par les applications pour surveiller l'activité des processus ou des utilisateurs. Dans le premier cas (les données utilisées proviennent du trafic réseau), on parle de NIDS (*Network-based IDS*). Dans les deux autres cas (les données utilisées proviennent du système hôte ou des applications), on parle d'HIDS (*Host-based IDS*). Les sources de données sont généralement capturées de manière passive sur le réseau ou sur le système. Lorsque le système de détection d'intrusion est capable de bloquer le trafic réseau ou les flux applicatifs, on parle de NIPS ou de HIPS respectivement (*Network/Host-based Intrusion Prevention System*). Deux méthodes de détection

d'intrusion sont généralement utilisées [MM07]: la détection comportementale (*behavior* ou *anomaly detection*) consiste à modéliser les activités normale, tandis que la détection basée sur la connaissance (*knowledge* ou *misuse detection*) consiste à modéliser les activités anormales. . 6, 117

**IDT** La table des descripteurs d'interruption (*Interrupt Descriptor Table*) stocke l'ensemble des adresses des routines de traitement des interruptions. Cette table de 256 entrées contient l'adresse des gestionnaires d'interruption et d'exception. En cas d'interruption (matérielle ou logicielle), elle permet au processeur d'appeler la routine appropriée gérant l'exception. Les 32 premières entrées (de 0 à 31) sont réservées par Intel pour les exceptions générées par le processeur (erreur de protection générale, erreur de page, division par zéro, etc.). Lors d'une interruption, le processeur reçoit un vecteur d'interruption qui est un index dans l'IDT. Le gestionnaire associé à l'interruption est alors exécuté. Par exemple, lorsqu'un programme ring 3 accède à une zone mémoire de ring 0, une erreur de protection mémoire est générée. Le gestionnaire du système d'exploitation Windows pour cette exception informe l'utilisateur d'une telle erreur et termine le programme. Un autre exemple, l'interruption `0x2E`, était utilisée pour permettre le passage du mode utilisateur (ring 3) au mode superviseur (ring 0). . 123

**Interruption logicielle** Une interruption logicielle (synchrone) est déclenchée par la couche logicielle du système. On exécute une interruption logicielle dans un programme à l'aide d'une instruction assembleur `int`. Lors de la réception de l'instruction `int` suivi du numéro d'interruption demandé, le processeur va vérifier dans la table IDT si les privilèges processeurs de la tâche qui a demandé l'interruption permette effectivement le lancement de l'interruption. . 49

**IRQ, Interruption matérielle** Le mécanisme d'interruption matérielle permet à un périphérique de communiquer au noyau du système d'exploitation qu'un événement qui lui est relatif vient de se produire (réception d'un paquet réseau, frappe clavier, etc.). Le périphérique concerné émet une interruption à destination du processeur. Après réception de cette interruption, le processeur va lancer la routine de traitement qui aura été définie préalablement au moyen de la table IDT. . 49

**LDT** Table locale des descripteurs de segment. Un descripteur de segment est une structure de 8 octets donnant l'adresse de base du segment, sa taille, le DPL, etc. Une LDT est dans le principe analogue à la GDT. Dans la philosophie globale cependant, la GDT est une table système unique, alors qu'il est possible de spécifier une LDT différente par application, voire plusieurs, et de laisser à l'application une certaine liberté de modification des segments de données accessibles en ring 3. . 123

**LT** Une *lookup table* est un tableau à une dimension. Ce tableau comporte $2^n$ indices et chaque élément du tableau est de taille $m$ bits. c'est donc une des représentations d'une fonction vectorielle de $\mathbb{F}_2^n \to \mathbb{F}_2^m$. Le principe d'une *lookup table* est de remplacer une fonction $g$ prenant en entrée des mots de $n$ bits et retournant des mots de $m$ bits. Cette représentation permet de lire instantanément la sortie de $g$ appliquée à $w \in \{0,1\}^n$ en lisant la valeur se trouvant dans sa lookup table à l'indice $w$. . 26

**MMIO** Le mode MMIO (*Memory Mapped I/O*) est l'un des deux modes principaux de communication entre le bloc processeur/mémoire principal et périphériques. Le chipset projette la mémoire partagée ou les registres de configuration des périphériques MMIO dans l'espace d'adressage physique. En d'autres termes, du point de vue du processeur, on accède à ces zones mémoires comme à la RAM à l'aide d'instruction assembleur `mov`. Comme ces zones mémoires sont gérées comme de la RAM par l'ensemble des composants du système, il est possible d'utiliser les mêmes mécanismes pour protéger ou gérer efficacement ces zones mémoires (segmentation, pagination) que pour la RAM. . 274

**Mode protégé** Le mode protégé est l'un des quatre modes de fonctionnement d'un processeur x86. Ce mode est le seul mode de fonctionnement 32 bits du processeur. Il s'agit du mode nominal de fonctionnement où tous les mécanismes de contrôle d'accès à la mémoire et aux périphériques (privilèges processeur, segmentation, pagination, privilèges d'entrées-sorties) sont utilisables. En mode protégé, les programmes sont donc soumis à des règles précises permettant l'exécution stable d'un système multi-tâches, et l'adressage de $2^{32} = 4$ Go. . 274

**Mode réel** Le mode réel d'un processeur x86 est un mode de compatibilité. Il s'agit d'un mode 16 bits. Lorsque le processeur est placé dans ce mode, il se comporte comme un processeur 8086 (processeur 16 bits de la famille x86). Ce mode est celui dans lequel le processeur est commuté au démarrage, et se limite à l'adressage de $2^{16} = 64$ Ko. . 127

**Mode System Management** Le mode *System Management* d'un processeur x86 est l'un des quatre modes de fonctionnement d'un processeur x86. Il s'agit d'un mode 16 bits de maintenance. . 274

**Mode virtuel 8086** Le mode réel d'un processeur x86 est un mode de compatibilité. Il s'agit d'un mode 16 bits. Lorsque le processeur est placé dans ce mode, il se comporte comme un processeur 8086 (processeur 16 bits de la famille x86). À la différence du mode réel, ce mode permet d'exécuter du code 16 bits avec des privilèges restreints. . 274

**Mémoires**
- Mortes:
  - ROM (Read Only Memory), programmée lors de sa création à l'aide d'un procédé proche de la photogravure.
  - PROM (Programmable Read Only Memory).
  - EPROM (Erasable Programmable Read Only Memory).
  - EEPROM (Electrically Erasable Programmable Read Only Memory).
  - Flash
- Vives:
  - RAM (Random Access Memory)
  - DRAM (Dynamic Random Access Memory).
  - SRAM (Static Random Access Memory), utilisée pour le cache du processeur et des périphériques.
  - DDRRAM (Double Data Rate Random Access Memory), utilisant un bus de mémoire de fréquence élevée.
  - CMOS RAM (Complementary Metal-Oxide Semiconductor Random Access Memory), utilisée pour sauvegarder la configuration utilisateur du BIOS.

. 8

**Niveaux de privilège, rings** Le mécanisme de protection des segments du processeur reconnaît 4 niveaux de privilège, numérotés de 0 à 3, souvent appelés rings. Le ring 0 est réservé pour les segments contenant le code, les données et les piles les plus privilégiés. Il est donc typiquement utilisé par le noyau du système d'exploitation.

Le processeur utilise des niveaux de privilège pour empêcher un programme ou une tâche opérant à un moindre niveau de privilège d'accéder à un segment de niveau de privilège plus élevé, sauf dans certaines situations bien contrôlées. Lorsque le processeur détecte une violation du niveau de privilège, il génère une exception de protection générale #GP (General Protection). Pour effectuer les vérifications de niveau de privilège entre segments de code et segments de données, le processeur reconnaît 3 types de niveaux de privilège:

- Le CPL (Current Processor Level) est le niveau de privilège du programme en cours d'exécution. Le CPL est stocké dans les deux premiers bits des registres de segment CS et SS. Normalement, le CPL est égal au niveau de privilège du segment de code auquel les instructions en cours d'exécution appartiennent. Le processeur change le CPL lorsque le contrôle du programme est transféré à un segment de code avec un niveau de privilège différent. Cependant, le CPL n'est pas modifié lorsque le contrôle du programme est transféré à un segment de code avec un niveau plus privilégié: Si CPL>DPL, alors le CPL n'est pas modifié. Par exemple, si un programme s'exécutant au CPL 3 accède à un segment de DPL 0, le CPL reste égal à 3.
- Le DPL (Descriptor Privilege Level) est le niveau de privilège d'un segment ou d'une porte (gate). Le DPL est stocké dans le descripteur de segment ou de porte. Lorsque le segment de code en cours d'exécution tente d'accéder à un autre segment, le DPL du segment cible est interprété différemment en fonction de son type:
  - s'il s'agit d'un segment de données, le DPL indique le niveau de privilège minimum requis pour pouvoir accéder au segment. Par exemple, si le DPL d'un segment de données vaut 1, seuls les programmes s'exécutant au CPL 0 ou 1 pourront y accéder.
  - s'il s'agit d'une porte (call gate), le DPL indique le niveau de privilège minimum requis pour pouvoir accéder au segment. Par exemple, si le DPL d'une porte vaut 1, seuls les programmes s'exécutant au CPL 0 ou 1 pourront y accéder.
  - s'il s'agit d'un TSS, le DPL indique le niveau de privilège minimum requis pour pouvoir accéder au segment. Par exemple, si le DPL d'un TSS vaut 1, seuls les programmes s'exécutant au CPL 0 ou 1 pourront y accéder.
  - s'il s'agit d'un segment de code non-conformant (non-conforming code segment), sans utilisation d'un call gate, le DPL indique le niveau de privilège requis exactement pour pouvoir accéder au segment. Par

exemple, si le DPL d'un segment de code non-conformant vaut 1, seuls les programmes s'exécutant au CPL 1 pourront y accéder.
  - s'il s'agit d'un segment de code conformant (conforming code segment) ou d'un segment de code non-conformant mais via un call gate, le DPL indique le niveau de privilège maximum requis pour pouvoir accéder au segment. Par exemple, si le DPL d'un segment de code conformant vaut 2, seuls les programmes s'exécutant au CPL 2 ou 3 pourront y accéder.
- Le RPL (Requested Privilege Level) est un niveau de privilège supplémentaire assigné aux sélecteurs de segments. Le processeur contrôle conjointement le CPL et le RPL pour décider si l'accès à un segment est autorisé. Même si le CPL du programme est suffisant pour accéder à un segment, l'accès lui est refusé si le RPL est insuffisant. Par exemple, si le RPL d'un descripteur de segment vaut 3 et que le CPL du programme vaut 0, le RPL remplace le CPL.

Les niveaux de privilèges sont contrôlés lorsque le sélecteur de segment d'un descripteur de segment est chargé dans un registre de segment. . 117

**Noyau** Un système d'exploitation est un ensemble de composants logiciels. Parmi ces composants, le noyau est celui qui possède les privilèges les plus importants. Il est responsable de l'ordonnancement des différentes tâches sur la machine et est garant de la politique de sécurité du système. . 117

**Packer** Un packer est un programme qui prend en entrée un exécutable, le chiffre ou le compresse, puis forge un nouvel exécutable composé d'une routine d'unpacking et d'un ou plusieurs blocs de données. La fonction d'unpacking implémente une partie des fonctionnalités d'un *loader* d'exécutable PE. Un packer est pour cette raison souvent appelé loader de protection. Au moment de l'exécution, l'exécutable d'origine est dynamiquement reconstruit en mémoire puis exécuté. . 44, 46, 92

**Pagination** Le mécanisme de segmentation permet de traduire toute adresse logique manipulée par un programme logiciel en une adresse virtuelle. La pagination permet de traduire une adresse virtuelle en adresse machine (adresse physique). Le principe du mécanisme de pagination est de proposer aux différents composants du système une vision virtuelle de la mémoire.
Par ce mécanisme, on associe à chaque adresse de l'espace mémoire une adresse de la mémoire physique. Le noyau du système d'exploitation a la possibilité de stocker dans un registre processeur l'adresse d'un répertoire de table de page potentiellement différent pour chaque tâche sous son contrôle. Ceci permet à chaque application d'avoir un espace qui lui est propre et qui ne lui permet pas d'accéder aux pages mémoires des autres entités. Le mécanisme de pagination permet d'autre part le swap. Contrairement au mécanisme de segmentation, le mécanisme de pagination est un mécanisme optionnel. Il est cependant employé par la plupart des systèmes d'exploitation. . 274

**PIO** Le mode PIO (*Programmed I/O*) est l'un des deux modes principaux de communication entre le bloc processeur/mémoire et les périphériques. Dans ce mode, les registres de configuration des périphériques sont projetés sur un bus distinct au moins logiquement du bus mémoire principal. On accède à l'un des registre de ce bus à l'aide des instructions assembleur `in` (lecture) et `out` (écriture d'un registre). Les registres projetés en PIO sont appelés ports d'entrées-sorties.
Les adresses des registres sur ce bus sont principalement déterminées par le chipset lors de la séquence de démarrage du système. En mode protégé, toute tâche s'exécutant en ring 0 peut accéder librement à tout registre PIO. En revanche, une tâche du ring 3 ne peut le faire qu'en obtenant d'une tâche du ring 0 la délégation de privilèges d'entrées-sorties pour les ports correspondants. Il est courant de nommer les registres projetés en PIO ports d'entrées-sorties. . 106

**Portes, gates** L'architecture du processeur définit un ensemble de descripteurs spéciaux appelés portes (*call gates*, *interrupt gates*, *trap gates* et *task gates*). Ils fournissent des passerelles vers des procédures et gestionnaires système qui peuvent opérer à un niveau de privilège plus élevé.
Un `call` vers un call gate peut fournir un accès à une procédure ou à un segment de code qui est à un niveau de privilège plus élevé que celui du segment de code courant. Pour accéder à une procédure via un call gate, la procédure appelante soumet le descripteur du call gate en opérande du `call`. Le processeur effectue un contrôle des droits d'accès au call gate, en comparant le CPL avec le niveau de privilège du call gate et du segment de code pointé par le call gate. Si l'accès au segment de code de destination est autorisé, le processeur récupère le sélecteur du segment de code et un décalage dans ce segment de code depuis le call

gate. Si l'appel nécessite un changement de niveau de privilège, le processeur commute aussi vers la pile du niveau de privilège cible. Le sélecteur du segment de la nouvelle pile est obtenue depuis le segment TSS de la tâche en cours d'exécution.
Une Call Gate est donc une structure pouvant être insérée dans la GDT ou la LDT d'un système et permettant l'exécution de code situé à une adresse mémoire définie dans la structure avec un niveau de privilège processeur différent du niveau de privilège courant. Une façon de réaliser un appel système est donc de définir une Call Gate pouvait être utilisée par du code s'exécutant avec des privilèges du ring 3 et permettant l'exécution du code de l'appel système avec les privilèges du ring 0. Il est bien entendu crucial qu'il soit impossible pour un attaquant d'ajouter à volonté des Call Gates dans la GDT ou la LDT du système dans la mesure où cela lui donnerait la possibilité d'exécuter depuis un ring qu'il choisit (le ring 3 vraisemblablement) du code à une adresse qu'il choisit, avec le niveau de privilège processeur qu'il choisit (ring 0 vraisemblablement). . 198

**Privilèges d'E/S** En mode protégé, toute tâche s'exécutant en ring 0 peut accéder librement à tout registre . En revanche, une tâche moins privilégiée ne peut le faire qu'en obtenant d'une tâche du ring 0 (en général le noyau) la délégation de privilèges d'accès aux ports correspondants que l'on nomme privilèges d'entrées-sorties. Le noyau dispose de deux mécanismes matériels. Le premier de ces mécanismes est de mettre à jour les deux bits IOPL du registre EFLAGS du processeur de tel sorte que la valeur de l'IOPL soit supérieure (au sens large) à la valeur du CPL. Dans ce cas, l'accès est autorisé en lecture ou écriture sur l'ensemble des ports PIO définis pour le système. Le second mécanisme est le bitmap d'entrées sorties qui permet de définir des permissions différentes pour chaque port 8 bits. Le bitmap d'entrées-sorties est une structure stockée dans le TSS (Structure d'état) de la tâche courante. Il est donc potentiellement différent pour chaque tâche processeur. . 106

**Profil de Protection** Un Profil de Protection est un document définissant un ensemble d'objectifs et d'exigences de sécurité, ce indépendamment de l'implémentation, pour une catégorie de produit. Il énonce rigoureusement la problématique de sécurité associée à cette catégorie de produit et spécifie le besoin de sécurité. Un profil de protection peut servir de modèle pour la cible de sécurité d'une évaluation. Il peut donc être vu comme une cible de sécurité générique. À la différence du profile de protection, la cible de sécurité d'un produit contient des informations spécifiques à sa mise en oeuvre qui ont pour objet de démontrer comment le produit répond au besoin de sécurité. Le processus d'évaluation peut se faire en conformité avec un (ou plusieurs) profils de protection. . 168

**Qualification** Dans le schéma français d'évaluation, le processus de qualification intervient en aval et en amont du processus d'évaluation. En amont, il permet une validation itérative de la cible de sécurité. En aval, il permet la prise en compte des résultats de l'analyse cryptographique. Il se concrétise par la délivrance de la qualification standard ou renforcée.
Ce terme est également souvent utilisé pour désigner le fait qu'une fonction, un mécanisme ou un outil "fait bien ce qu'il est censé faire", soit parce qu'il satisfait à des critères précis permettant d'assurer la confiance, soit parce qu'il a déjà été expertisé. . 7

**Registres de segment** Les registres de segments (segment de code CS, segment de données DS, segment de pile SS, segments extra ES, FS, GS) sont employés pour délimiter les différents segments utilisés par un programme.
- En mode réel, ils contiennent une valeur de 2 octets que l'on combine avec une valeur de deux octets contenue dans l'un des registres de déplacement DI, SI, afin de former une adresse de 20 bits.
- En mode protégé, ils contiennent des sélecteurs de segments. Un sélecteur de segment contient trois champs: le niveau de privilège demandé RPL= $0, 1, 2$ ou 3, la table contenant le descripteur (0:GDT ou 1:LDT) et un décalage qui sert d'index dans la table des descripteurs de segment.

. 274

**Rigidité** La rigidité d'un schéma de détection est définie [Fil07c, page 16] comme la difficulté plus ou moins grande qu'aura un plagiaire à le contourner. . 7

**RPL** Le RPL (*Requested Privilege Level*) est un niveau de privilège supplémentaire assigné aux sélecteurs de segments. Le processeur contrôle conjointement le CPL et le RPL pour décider si l'accès à un segment est autorisé. Même si le CPL du programme est suffisant pour accéder à un segment, l'accès lui est refusé si le RPL est insuffisant. . 274

**Schéma de détection** Un schéma de détection $\{\mathcal{S}_{\mathcal{M}}, f_{\mathcal{M}}\}$ d'un *Malware* $\mathcal{M}$ est défini dans [Fil07c, page 13] comme la donnée d'une information $\mathcal{S}_{\mathcal{M}}$ (ensemble d'octets dans le cas de l'analyse de forme, ensemble de fonctions d'un programme dans le cas de l'analyse comportementale) et d'une fonction de détection $f_{\mathcal{M}}$ (relativement à $\mathcal{M}$). J'utilise ce terme pour désigner de manière générale l'algorithme d'un moteur de détection et les données qu'il utilise pour détecter un *Malware*. . 1, 3

**Segmentation** La segmentation est un mécanisme de gestion de la mémoire obligatoire (i.e. qu'il est activé par défaut et ne peut être désactivé) sur les processeurs x86 en mode protégé. Lorsque l'on exécute du code sur un tel processeur, on manipule des adresses logiques. La MMU (unité de gestion de la mémoire du processeur) traduit via le mécanisme de segmentation cette adresse logique en adresse virtuelle qui sera ensuite traduite en adresse machine (ou physique) via le mécanisme de pagination. Le mécanisme de segmentation permet de partitionner la mémoire virtuelle en zones mémoires (segments) et de définir un contrôle d'accès basique sur chacune de ces zones mémoires.
Quatre segments couvrant tout l'espace mémoire virtuel sont utilisés par le système d'exploitation Windows: une paire code/données pour le noyau et une autre pour le mode utilisateur. . 274

**SHF** Les fichiers possédant l'attribut SHF (*Super Hidden File*) sont créés par exemple par utilisation de la fonction `CreateFile` avec en argument le masque `FILE_ATTRIBUTE_SYSTEM|FILE_ATTRIBUTE_HIDDEN`. Ces fichiers restent cachés à l'utilisateur dans Windows Explorer même lorsque l'utilisateur le paramètre pour montrer les fichiers cachés. . 110

**Sémantique dénotationnelle** La sémantique dénotationnelle $S_d$ préserve le comportement observable d'un programme, i.e. son comportement en termes d'entrées/sorties. Elle abstrait l'historique des calculs en n'observant que les relations d'entrées/sorties des traces finies:

$$\{\sigma \in \Sigma^+ \mid \exists \sigma' \in T[\![P]\!] \cap \Sigma^+ \, \sigma_0 = \sigma'_0 \text{ et } \sigma_F = \sigma'_F\}$$

et les entrées sur lesquelles les traces infinies divergent:

$$\{\sigma \in \Sigma^w \mid \exists \sigma' \in T[\![P]\!] \cap \Sigma^w \, \sigma_0 = \sigma'_0\}.$$

On a donc:

$$S_d(X) = \{\sigma \in \Sigma^+ \mid \exists \sigma' \in X \cap \Sigma^+ \, \sigma_0 = \sigma'_0 \text{ et } \sigma_F = \sigma'_F\} \cup \{\sigma \in \Sigma^w \mid \exists \sigma' \in X \cap \Sigma^w \, \sigma_0 = \sigma'_0\}.$$

La sémantique dénotationnelle $S_d$ peut être vue comme une abstraction de la sémantique des traces maximales (voir la hierarchie des sémantiques d'un système de transition [Cou02]). . 84, 243

**TLB** Le processeur possède un cache appelé TLB (*translation look-aside buffer* qui conserve les derniers résultats de traduction d'adresses (associations page virtuelle - page physique ainsi que les protections appliquées à ces pages). Lors de la traduction d'une adresse virtuelle, celle-ci est recherchée dans ce cache. Si elle est trouvée, l'adresse physique associée est utilisée pour accéder à la mémoire, évitant ainsi une recherche plus lente dans l'annuaire de pages et la table de pages . 274

**TR, Task Register** Le registre TR (Task Register) contient le sélecteur du segment TSS de la tâche courante. . 274

**Trap, capture des événements** Mécanisme du processeur pour capturer un thread en cours d'exécution, lorsqu'une interruption ou une exception survient, et transférer le contrôle à une routine spécialisée du système d'exploitation. Sous Windows, le processeur transfère le contrôle à un gestionnaire d'événement (*trap handler*), une fonction spécifique à une interruption ou une exception particulière. . 274

**TSS** Le segment TSS (*Task-State Segment*) est un segment système qui définit l'état de l'environnement d'exécution d'une tâche. Il inclus l'état des registres généraux, des registres de segment, des registres `EIP`, `EFLAGS`, ainsi que le sélecteur de segment pour la LDT associée à la tâche, l'adresse de base de la page table, et les sélecteurs de segments de 3 piles, une pile pour chaque niveau de privilège (autres que le niveau de privilège courant). . 274

**Unpacker** J'appelle unpacker un programme qui prend en entrée un exécutable packé, supprime le *loader* de protection, et fournit en sortie l'exécutable embarqué. . 1, 92

**VMM** La virtualisation de ressources consiste à développer une couche d'abstraction (généralement logicielle, parfois partiellement matérielle). Cette couche d'abstraction, appelée VMM (*Virtual Machine Monitor*) ou hyperviseur de VM, permet de présenter aux composants logiciels destinés à s'exécuter sur la machine une vision virtuelle du matériel. Les vues présentées à chacune des applications sont potentiellement différentes de l'architecture matérielle réelle, et potentiellement différentes entre elles. Chacun des composants logiciels a alors l'impression de s'exécuter seul sur une machine réelle. Elle s'exécute en réalité sur une machine virtuelle. Les appels aux différentes fonctionnalités de la machine virtuelle sont redirigés par le moniteur de machine virtuelle au matériel réel de la machine qui gère notamment le contrôle d'accès aux ressources et les éventuels accès concurrents à ces dernières. L'application fondamentale de la virtualisation est l'isolation de composants. Il est possible en utilisant des techniques de virtualisation de faire tourner plusieurs systèmes d'exploitation en parallèle sur une même machine. Chacun des systèmes d'exploitation a alors l'impression de s'exécuter seul sur une machine réelle.

Lorsqu'un système est virtualisé, le rapport entre le nombre d'instructions CPU exécutées en logiciel et directement par le matériel peut être utilisé pour déterminer s'il s'agit d'une machine virtuelle de type CSIM (*Complete Software Interpreter Machine*), HVM (*Hybrid VM*), d'une VMM (*Virtual Machine Monitor*) de type I ou II, ou une machine réelle:

- une CSIM ou émulateur utilise uniquement l'interprétation logicielle (toutes les instructions CPU sont émulées par un programme logiciel) ;
- une HVM interprète chaque instruction privilégiée par le biais du logiciel ;
- une VMM de type II s'exécute comme une application sur l'OS hôte (elle interprète uniquement les instructions sensibles en logiciel. Les instruction privilégiées non-sensibles sont exécutées directement par le CPU) ;
- une VMM de type I s'exécute comme un OS ou le noyau d'un OS. Comme c'est la cas d'une VMM de type II, elle interprète uniquement les instructions CPU sensibles en logiciel. Les instructions privilégiées non-sensibles sont exécutées directement par le CPU. VMware ESX et Xen sont des VMM de type I.

Les processeurs x86 les plus récents comportent des extensions dites d'aide à la virtualisation ou encore de virtualisation matérielle. Intel et AMD ont ainsi développé en parallèle deux technologies d'aide à la virtualisation. La technologie Intel se nomme VT (anciennement Vanderpool) [UNR+05] et la technologie AMD Pacifica [Zei05]. Ces technologies reposent sur la même philosophie mais sont par ailleurs incompatibles. . 106

**VxStripper** VxStripper est un outil d'analyse virale. Je l'ai conçu pour la rétro-analyse de programmes protégés et potentiellement hostiles. Il implémente un émulateur et plusieurs fonctions d'analyse spécialisées, permettant d'observer le programme cible et son environnement d'exécution en toute sécurité pour l'utilisateur. Il sert de base tant à l'extraction d'information d'un code viral qu'au maintient d'une base virale exploitable dans le cadre des tests en boîte noire. Son nom provient du terme anglais *strip* qui signifie déshabiller. L'objectif de l'analyste est en effet d'ôter les protections et informations parasites du programme viral afin de pouvoir l'admirer dans sa plus stricte nudité. . 7, 115–117, 188–190, 195

**Résumé :**

Cette thèse est consacrée à la problématique de l'évaluation des produits antivirus. L'utilisateur final d'un produit antivirus ne sait pas toujours quelle confiance il peut placer dans son produit antivirus pour parer convenablement la menace virale. Pour répondre à cette question, il est nécessaire de formuler la problématique de sécurité à laquelle doit répondre un tel produit et de disposer d'outils et de critères méthodologiques, techniques et théoriques permettant d'évaluer la robustesse des fonctions de sécurité et la pertinence des choix de conception au regard d'une menace virale identifiée. Je concentre mon analyse de la menace virale sur quelques mécanismes (mécanismes cryptographiques, transformations de programme) adaptés au contexte boîte blanche, i.e. permettant à un virus de protéger ses données critiques en confidentialité et de masquer son fonctionnement dans un environnement complètement maîtrisé par l'attaquant. Une première étape incontournable avant toute organisation d'une ligne de défense consiste à analyser les virus, afin d'en comprendre le fonctionnement et les effets sur le système. Les méthodes et techniques de la rétro-ingénierie logicielle occupent ici - au côté des techniques de cryptanalyse - une place centrale. J'ai pris le parti de focaliser sur les méthodes dynamiques d'extraction d'information fondées sur une émulation du matériel supportant le système d'exploitation. L'évaluation d'un moteur de détection selon des critères objectifs requiert un effort de modélisation. J'étudie quelques modèles utilisés actuellement (grammaires formelles, interprétation abstraite, statistique). Chacun de ces modèles permet de formaliser certains aspects du problème de la détection virale. Je m'attache à l'étude des critères qu'il est possible de définir dans chacun de ces cadres formels et à leur utilisation pour partie des travaux d'évaluation d'un moteur de détection. Mes travaux m'ont conduit à la mise en oeuvre d'une approche méthodologique et d'une plate-forme de tests pour l'analyse de robustesse des fonctions et mécanismes d'un produit anti-virus. J'ai développé un outil d'analyse de code viral, appelé VxStripper, dont les fonctionnalités sont utiles à la réalisation de plusieurs des tests. Les outils formels sont utilisés pour qualifier en boîte blanche - ou sur la base des documents de conception détaillée - la conformité et l'efficacité d'un moteur de détection.

**Mots clés :** Cryptographie, Crypto-virologie, Rétro-ingénierie, Analyse virale, Détection virale, Interprétation abstraite, Modèles statistiques.

**Abstract :**

*This thesis is devoted to the problem of anti-virus software evaluation. The end user of an anti-virus software package does not always know what confidence he can place in his product to counter effectively the viral threat. In order to answer this question, it is necessary to formulate the security issue which such a product must respond to and to have tools and methodological, technical and theoretical criteria for assessing the robustness of security functions and the relevance of design choices against an identified viral threat. I concentrate my analysis of the viral threat on a few mechanisms suited to the white box context, i.e. allowing a virus to protect the confidentiality of its critical data and to mask its operations in an environment completely controlled by the attacker. A first mandatory step before any organization of a line of defense is to analyze viruses, in order to understand their operation and impact on the system. Software reverse-engineering methods and techniques occupy a central place here (alongside cryptanalysis methods). I basically focus on the dynamic methods of information extraction based on an emulation of the hardware supporting the operating system. The evaluation of a detection engine based on objective criteria requires an effort of modeling. I study a few models (formal grammars, abstract interpretation, statistics). Each of these models makes it possible to formalize some aspects of the viral detection problem. I make every effort to study the criteria it is possible to define in each of these formal frameworks and their use for part of the evaluation tasks of a detection engine. My work led me to implement a methodological approach and a testing platform for the robustness analysis of the functions and mechanisms of an anti-virus. I developed a tool to analyze viral code, called VxStripper, whose features are useful to several tests. The formal tools are used to qualify (on the basis of the detailed design information) compliance and effectiveness of a detection engine.*

**Keywords :** *Cryptography, Cryptovirology, Reverse-engineering, Virus Analysis, Virus Detection, Abstract Interpretation, Statistical Models.*