A computability perspective on self-modifying programs

Guillaume Bonfante, Jean-Yves Marion, Daniel Reynaud

▶ To cite this version:

Guillaume Bonfante, Jean-Yves Marion, Daniel Reynaud. A computability perspective on self-modifying programs. 7th IEEE International Conference on Software Engineering and Formal Methods - SEFM 2009, Nov 2009, Hanoi, Vietnam. inria-00433472

HAL Id: inria-00433472
https://inria.hal.science/inria-00433472

Submitted on 19 Nov 2009

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

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

A computability perspective on self-modifying programs

Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud-Plantey
Nancy University – LORIA,
615, rue du Jardin Botanique, BP-101, 54602 Villers-lès-Nancy, France
Guillaume.Bonfante—Jean-Yves.Marion—Daniel.Reynaud-Plantey@loria.fr

***Abstract*—In order to increase their stealth, malware commonly use the self-modification property of programs. By doing so, programs can hide their real code so that it is difficult to define a signature for it. But then, what is the meaning of those programs: the obfuscated form, or the hidden one? Furthermore, from a computability perspective, it becomes hard to speak about the program since, its own code varies over time. To cope with these issues, we provide an operational semantics for self-modifying programs and we show that they can be constructively rewritten to a non-modifying program.**

***Keywords*-Self-modifying code, semantics, computability, virus, obfuscation**

I. Introduction

Self-modifying programs are programs which are able to modify their own code at runtime. Nowadays, self-modifying programs are commonly used. For example, a packer transforms any program into a program with equivalent behavior, but which decompresses and/or decrypts some instructions. Thus, packers transform programs into self-modifying programs. Another example of self-modifying programs are just-in-time compilers.

Self-modifying techniques allow obfuscation of codes, thus protecting the intellectual property of the program authors. Besides of these positive applications, malware heavily use self-modification to armour themselves and to avoid detection, and so throw the self-modification paradigm in the dark side of programming.

There are lots of reasons to study self-modifying programs from both a theoretical and a practical point of view. One reason is to be able to have a good understanding of what can be done with self-modifying programs. Another reason is to provide tools to analyse them in the context of malware. We may foresee difficulties of such an analysis by reading for example the introduction of [1]. As opposed to traditional programs, we do not have a static access to the instructions of a self-modifying program. That is why, we shall introduce pseudo-programs, that is programs for which we just have a fragment of the listing (corresponding to the current step of a computation). Indeed, a self-modifying program may write and run some new code and it cannot be predicted a priori without execution. So we just have a partial view of the code. In short, runtime analysis is very hard even for trained professional reverse engineer but currently remains the only practical approach. On the other hand, we are not aware of any effective static analysis for self-modifying programs. This situation certainly comes from the lack of studies on self-modifying constructions. To our knowledge, there are only a few scientific papers on this topic, and without being exhaustive, we may mention: [2] which proposes an axiomatic semantics, and [3] which tries to provide a semantics.

More recently in [4], we developed a dynamic type system and a tool, TraceSurfer, in order to analyse self-modifying binary programs, to recognize packer signatures and to establish some non-interference like properties on binary code. TraceSurfer outputs a view of the relations between layers of dynamic code (monitoring, generation, secure erasing). We have observed that the strategies of the virus writers are sophisticated. This is one of our motivation for a deep analysis of self-modifying programs.

This study is an attempt to contribute to the understanding of self-modifying programs. For this, we provide some semantics. Next to the traditional approaches, operational, axiomatic and denotational semantics, we claim that deobfuscation also plays the role of a semantics. Obfuscation usually hides the real code of a program by transforming it according to some rules. In some way, the real code still exists, but in a hidden form. Deobfuscation then consists in rediscovering the initial code within the fog.

Our contribution is to show that classical computability results may give a better understanding of self-modifying programs and deobfuscation. This study follows the spirit of the works of Jones [5], [6]. Our main result is a constructive interpretation of Rogers's isomorphism theorem. The original result says that given two (acceptable) programming languages, there is an effective isomorphism between both languages. In our context, we use Rogers's construction to define a computability semantics of a self-modifying program.

II. An abstract assembly language

One point here is about the design of the assembly language. In a "real" machine language, addresses and values are encoded say by 32-bit words and so they are finite. Let us cite Jones at that point "We here have a paradoxical situation: that the most natural model of daily computing on computers, which we know to be finite, is by an infinite (i.e., potentially unbounded) computation

model.". For this reason, we use an infinite model, however, we have tried to keep things as finite as possible. We use finitely many registers, finitely many instructions (of fixed size) and memory cells contain only one letter. But, in order to deal with unbounded addresses, since they are stored in registers, we allow the content of these registers to be unbounded.

The rationale of our model of a machine is to put the focus on one (unilateral) infinite storage tape, where instructions are loaded, executed and potentially transformed. Registers serve only for the computation of intermediate values and for the storage of the current instruction address.

In that sense, our model differs from the usual *random access machine* (RAM) model which puts efforts on registers (in particular, register machines employ a denumerable set of registers). As a matter of fact, our model is closer to a *counter machine* (CM). Since we have only a finite number of registers, our model is less powerful than a RAM. On the other hand, it is closer to the functioning of current computers.

Anyway, what makes the present model different from these two standard models is that we store the program within the configuration, not in an idealized stable world. Consequently, usual simulations of (say) Turing Machines by RAM, and all classical results and notions (such as specializers, self-interpretation, padding, Kleene's fixpoint Theorem and so on) must be reconsidered in the present context.

A. The syntax

Let B be a finite set of *letters* modelling bytes. B^* denotes the set of finite words over B. We call elements in B^* *addresses* or *pointers*. From now on, we suppose that there is a blank character $\Box \in B$. On B^*, we use the following operations.

- $|w|$ denotes the size of words.
- The concatenation operation is written with a dot.
- Given a word w, we denote by w_i its i-th letter, begining with index 0, that is $w = w_0.w_1 \cdots w_{|w|-1}$.

Furthermore, we suppose given an arithmetic on pointers by means of an isomorphism between $(B \backslash \{\Box\})^*$ and $\mathbb{N}$, let us say via $\iota : (B\backslash\{\Box\})^* \to \mathbb{N}$[1]. Then, $\iota^{-1}(0)$ is the initial address, $\iota^{-1}(\iota(w)+1)$ returns the "next" address, etc. To avoid tedious notations, we will no longer make a clear distinction between addresses and natural numbers, and we will write $w + k$ where w is an address and k an integer. The context shows what is going on. We extend ι to words $w \in B^*$, saying that $\iota(\Box^n u) = \iota(u\Box^n) = \iota(u)$ with $u \in (B\backslash\{\Box\})^*$ and $n \in \mathbb{N}$. In other words, a $\Box$ used as prefix or suffix is transparent for ι. As a matter of fact, one will have observed that such an arithmetic is largely used in low-level programming languages.

Finally, let R be a finite (non empty) set of *registers*. Without loss of generality, one of these registers is `ip` , the *instruction pointer*. The choice of the other registers belongs to the design of the framework (the machinery).

A function $\rho : R \to B^*$ is named a *register valuation* and a function $\sigma : B^* \to B$ is called a *store*. $\mathbb{S}$ denotes the set of stores. In the present settings, we do not introduce the notion of stack. This could be done without harm.

The function $0 : R \to B^*$ is the constant function $r \mapsto 0$. For stores, $\Box : B^* \to B$ is the function $w \mapsto \Box$. We introduce an update function on stores. Given $\sigma : B^* \to B$, $k \in \mathbb{N}$ and a word $w \in B^*$, we write $\sigma[k \leftarrow w]$ for the store:

$$\begin{array}{rcl} \sigma[k \leftarrow w] : B^* & \to & B \\ v & \mapsto & \begin{cases} \sigma(v) & \text{if } v < k \text{ or } v \geq k + |w| \\ w_i & \text{if } v = k + i \end{cases} \end{array}$$

In other words, looking at the store as a tape, it means that one writes the word w from the index k. Finally, we use the notation $\sigma(m..n)$ where $m \leq n \in \mathbb{N}$ for the word $\sigma(m).\sigma(m+1)\cdots\sigma(n)$. If $n < m$, then $\sigma(m..n)$ is the empty word.

The abstract assembly language (ASL) is:

```
LOAD r r r   CPY r r      MOV r r
TEST r r     JUMP r       STOP
L_SHIFT r r  R_SHIFT r r
L_CCAT l r   R_CCAT l r
OP r r r     NOT r
```

with `r` and `l` respectively register names and letters and $\mathtt{OP} \in \{\mathtt{ADD}, \mathtt{SUB}, \mathtt{MUL}, \mathtt{DIV}, \mathtt{MOD}, \mathtt{CCAT}, \mathtt{EQ}, \mathtt{LEQ}, \mathtt{AND}, \mathtt{OR}\}$.

The concrete syntax of the assembly language (CAL) is an encoding of the ASL with words in B^*. To avoid being too abstract, we provide now such an encoding. However, one should keep in mind that we essentially use only one feature of this encoding: the language of instructions must be prefix, that is there are no words $w_1, w_2 \in CAL$ such that $w_1 = w_2.u$ with $u \in B^*$. The reason is that instructions are encoded in memory. Therefore, at one address in the store, there should be no ambiguity on the current instruction to be executed.

Let us consider words $\mathtt{mov}, \mathtt{l_shift}, \mathtt{add}, ... \in (B\backslash\{\Box\})^*$ to encode the ASL lexemes[2]. Registers are encoded in the same way by words $\mathtt{ip}, \mathtt{ap}, ... \in (B\backslash\{\Box\})^*$ which are taken to be different from the latter ones.

By a clever choice of the encoding words, we can suppose that the encoding of ASL instructions is a prefix language. Moreover, we can even suppose that encoded instructions all have the same size, say $\mathcal{K}$.

[1] Actually, since definitions are totally relative to ι, the isomorphism could not be computable. However, in order to provide a concrete implementation of the machine, we require it to be so.

[2] One could use B^*, but this condition ensures that non-self modifying programs can be written in $(B\backslash\{\Box\})^*$, a property used for Theorem 11.

Due to the fact that CAL is prefix, the ternary relation instr defined below is actually a (partial) function $\mathbb{S} \times B^* \to CAL$:

$$(\sigma, k, w) \in \texttt{instr} \Leftrightarrow \sigma(k..k + |w| - 1) = w.$$

Thus, we will write $\texttt{instr}(\sigma, k)$ to mention the unique instruction w such that $\texttt{instr}(\sigma, k, w)$ if such an instruction exists. Otherwise, we write $\texttt{instr}(\sigma, k) = \bot$.

Traditionally, a program has a fixed text. Its code is a list of instructions invariant wrt any run, on which analyses can be performed. In the context of self-modifying programs, the situation is different because we don't have access to the whole code. The code evolves during a computation and may depend on the input. So we introduce the notion of *pseudo-program*.

Definition 1. *A* pseudo-program *is a piece of text* $\texttt{p} \in B^*$ *which potentially contains the code which will be executed. To distinguish pseudo-programs from arbitrary strings, we use a type writer font and we use* P *as an alias for* B^* *for the set of pseudo-programs.*

Contrary to what happens in the usual case, one cannot make a clear distinction between instructions and data since some data may become instructions after being rewritten and vice versa. So, we cannot define a pseudo-program to be a string in CAL^* which would be the natural presentation for non self-modifying programs.

III. Operational semantics

A *configuration* is given by a couple (ρ, σ) where ρ is a register valuation and σ a store. Configurations caracterize the states of the machine.

Definition 2 (Operational Semantics)**.** *The* successor *relation on configurations is defined in Figure 1. As usual, we write* $(\rho, \sigma) \to^n (\rho', \sigma')$ *the fact that* $(\rho, \sigma) = (\rho_0, \sigma_0) \to (\rho_1, \sigma_1) \to \cdots \to (\rho_n, \sigma_n) = (\rho', \sigma')$ *and* $\to^*$ *is the transitive closure of* $\to$*.*

One may have observed that indirect addressing is done via the store. Since we have only finitely many registers, we can name them directly. However, to denote some particular window in the memory, we use two registers, one for the begining and one for the length of the window.

From the remaining of the section, we suppose given a the domain of computations Σ^* where $\Sigma \subseteq B$. In particular, one will have observed that pseudo-programs (by making $\Sigma = B$) can be used as data of some other programs.

Given a pseudo-program $\texttt{p} \in P$ and k words $w_1, \ldots w_k \in \Sigma^*$, the initial configuration (for these words) is defined as $c_0(\texttt{p}, w_1, \ldots, w_k) = (0, \Box[0 \leftarrow \texttt{p}, |\texttt{p}| + 1 \leftarrow w_1.\Box.w_2.\Box \cdots \Box w_k])$.

A function $\phi : (\Sigma^*)^k \to \Sigma^*$ is computed by a pseudo-program $\texttt{p}$ if for all $w_1, \ldots, w_k \in \Sigma^*$, we have $c_0(\texttt{p}, w_1, \ldots, w_k) \to (\rho_1, \sigma_1) \to \cdots \to (\rho_n, \sigma_n)$ where a) $\texttt{instr}(\rho_n(\texttt{ip})) = \texttt{stop}$, and b) $\rho(\texttt{out}) = \phi(w_1, \ldots, w_k)$ with out a given and fixed register. Conversely, a program $\texttt{p}$ computes the unique function $\phi : B^* \to B^*$ such that:

- $\phi(x) = \rho(\texttt{out})$ if one has $c_0(\texttt{p}, w_1, \ldots, w_k) \to (\rho_1, \sigma_1) \to \cdots \to (\rho_n, \sigma_n)$ and $\texttt{instr}(\rho_n(\texttt{ip})) = \texttt{stop}$.
- $\phi(x)$ is otherwise undefined.

This function is written $[\![\texttt{p}]\!]$.

A. Some examples

For the notation of programs, we use the semi-column instead of the dot to denote the concatenation of words.

Example 1. *Let us introduce some syntactic sugar. Given a word* w*, we define:*

$$\begin{aligned} \texttt{l_ccat}\ w\ \texttt{r} \quad \triangleq \quad & \texttt{l_ccat}\ w_{|w|-1}\ \texttt{r}; \\ & \texttt{l_ccat}\ w_{|w|-2}\ \texttt{r}; \\ & \vdots \\ & \texttt{l_ccat}\ w_0\ \texttt{r} \end{aligned}$$

To test if a register $\texttt{r}$ *equals some word* $w \in B^*$ *and jump otherwise to the content of the register* $\texttt{p}$*, we use:*

$$\begin{aligned} \texttt{test}_w\ \texttt{r}\ \texttt{p} \quad \triangleq \quad & \texttt{l_ccat}\ w\ \texttt{tp1}\ ; \\ & \texttt{eq}\ \texttt{r}\ \texttt{tp1}\ ; \\ & \texttt{test}\ \texttt{tp1}\ \texttt{p} \end{aligned}$$

where tp1 *is a temporary register.*

The following program computes the length of its first argument (written in $(B \backslash \{\Box\})^*$).

Example 2. *Registers are* $\texttt{r}, \texttt{s}, \texttt{ap}, \texttt{out}, \texttt{b}, \texttt{p}$*.*

$$\begin{aligned} \texttt{length} \quad \triangleq \quad & \texttt{l_ccat}\ k_1\ \texttt{r}; \texttt{l_ccat}\ k_2\ \texttt{s}; \\ & \texttt{l_ccat}\ k_3\ \texttt{b}; \texttt{l_ccat}\ 1\ \texttt{ap}\ ; \\ & \texttt{load}\ \texttt{r}\ \texttt{ap}\ \texttt{p}; \texttt{test}_\Box\ \texttt{p}\ \texttt{s}; \\ & \texttt{add}\ \texttt{ap}\ \texttt{out}\ \texttt{out}\ ; \texttt{add}\ \texttt{ap}\ \texttt{r}\ \texttt{r} \\ & \texttt{jump}\ \texttt{b}; \texttt{stop} \end{aligned}$$

where k_1 *is the address of the argument (that is the size of the program plus one),* k_2 *is the address of the instruction* stop *and* k_3 *is the address of the instruction* load r ap p*.*

We present in Appendix A a technique to compute the k_i*s. This technique will be used later on and, in particular, in Proposition 8.*

B. The robustness of the model

One first point deals with the computational cost of each step of computation. Reading the instructions can be done in constant time, indeed, we took the precaution to encode instructions with words of size equal to a constant $\mathcal{K}$. As this happens for RAM, the unit cost of operations on registers depends on the size of the content of these registers. We refer

$$\frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{load}\ r_1\ r_2\ r_3 \quad \rho(r_1) = n \quad \rho(r_2) = \delta \quad \sigma(m..(m+\delta)) = w \quad k + |\texttt{instr}(\sigma,k)| = m}{(\rho,\sigma) \to (\rho[\texttt{ip} \leftarrow m, r_3 \leftarrow w], \sigma)}$$

$$\frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{mov}\ r_1\ r_2 \quad \rho(r_1) = w \quad \rho(r_2) = n \quad k + |\texttt{instr}(\sigma,k)| = m}{(\rho,\sigma) \to (\rho[\texttt{ip} \leftarrow m], \sigma[n \leftarrow w])}$$

$$\frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{op}\ r_1\ r_2\ r_3 \quad \text{op}(\rho(r_1),\rho(r_2)) = w \quad k + |\texttt{instr}(\sigma,k)| = m}{(\rho,\sigma) \to (\rho[\texttt{ip} \leftarrow m, r_3 \leftarrow w], \sigma)}$$

where op $\in$ {add, sub, mul, div, mod, ccat, eq, leq, and, or (*)}

$$\frac{\texttt{instr}(\sigma,\rho(\texttt{ip})) = \texttt{l_shift}\ r_1\ r_2 \quad \rho(r_1) = l.w \quad \rho(\texttt{ip}) + |\texttt{instr}(\sigma,\rho(\texttt{ip}))| = m}{(\rho,\sigma) \to (\rho[\texttt{ip} \leftarrow m, r_1 \leftarrow w, r_2 \leftarrow l], \sigma)}(**)$$

$$\frac{\texttt{instr}(\sigma,\rho(\texttt{ip})) = \texttt{l_ccat l}\ r \quad \rho(r) = w \quad \rho(\texttt{ip}) + |\texttt{instr}(\sigma,\rho(\texttt{ip}))| = m}{(\rho,\sigma) \to (\rho[\texttt{ip} \leftarrow m, r \leftarrow \texttt{l}.w], \sigma)}(**)$$

$$\frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{test}\ r_1\ r_2 \quad \rho(r_1) = \top \quad k + |\texttt{instr}(\sigma,k)| = m}{(\rho,\sigma) \to (rho[\texttt{ip} \leftarrow m], \sigma)}$$

$$\frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{test}\ r_1\ r_2 \quad \rho(r_1) = \bot \quad \rho(r_2) = m}{(\rho,\sigma) \to (rho[\texttt{ip} \leftarrow m], \sigma)}$$

$$\frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{jump}\ r \quad \rho(r) = m}{(\rho,\sigma) \to (\rho[\texttt{ip} \leftarrow m], \sigma)}$$

$$\frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{not}\ r \quad \texttt{not}(\rho(r)) = b}{(\rho,\sigma) \to (\rho[r \leftarrow b, \sigma)} \qquad \frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{stop}}{(\rho,\sigma) \to (\rho,\sigma)}$$

$$\frac{\rho(\texttt{ip}) = k \quad \texttt{instr}(\sigma,k) = \texttt{cpy}\ r_1\ r_2 \quad \rho(r_1) = w}{(\rho,\sigma) \to (\rho[r_2 \leftarrow w, \sigma)}$$

$(*)$ Substration is defined on natural numbers as sub $(n,m) = \max(0, n-m)$. Concatenation is ccat $(u,v) = u.v$. For binary operations, there are two (arbitrary) values $\bot, \top$, respectively for "true" and "false". Usually, 0 serves as false, and otherwise, the value is considered as true. $(**)$ where $l \in B$. The l_shift operation on the empty word returns the empty word. The rule for r_shift is analogous, and has been ommitted, so is the rule for `r_ccat`. In case of an instruction op r_1 r_2 ip, we put the priority on the normal flow, that is ip $= k + |\texttt{instr}(\sigma,k)| = m$ after the instruction. The same remark holds for `l_shift` , `l_ccat` , ...

Figure 1. The rules of the operational semantics

to Jones [6] for a full discussion about these issues. Anyway, complexity theory is outside the scope of this paper, so that we take the simplest notion of time complexity: the computation length.

Definition 3 (Time complexity)**.** *Let* $\texttt{p} \in P$*, we define* $\texttt{time}(\texttt{p}(x_1,\ldots,x_n))$ *to be its computation length. That is:* $\texttt{time}(\texttt{p},x_1,\ldots,x_n) = \min\{k \in \mathbb{N} \mid c_0(\texttt{p},x_1,\ldots,x_n) \to^k (\rho,\sigma)\}$*, where* $\texttt{instr}(\rho(\texttt{ip})) = \texttt{stop}$*, and is undefined otherwise.*

We say that a program p is *static* if for all computations $c_0(\texttt{p},w_1,\ldots,w_k) \to^n (\rho_n,\sigma_n)$,

- $\sigma_n(0..|\texttt{p}|-1) = \texttt{p}$,
- the current instruction is an instruction of p, that is $\texttt{instr}(\texttt{ip}, \rho_n) < |p|$.

In other words, the text of the program remains unchanged, and the instruction pointer never goes outside the program.

Proposition 4. *There is a constant* R *such that any static program for a machine with* $k+1$ *registers working in time* $T(n)$ *(*n *is the size of the input) can be simulated by a static program written for a machine with* R *registers and a computation time* $O(T(n))$*.*

Proof: The principle of our simulation is to use 7 registers to encode the $k+1$ registers $r_1,\ldots,r_k$ of the simulated pseudo-program p. Let us call them $\texttt{ip}, \texttt{m}, \texttt{M}, \texttt{np}, \texttt{r}, \texttt{tp0}, \texttt{tp1}$.

- m contains the maximal length of the registers r_i,

- $\mathtt{M}$ contains $(|B|-1)^{\mathtt{m}}$, that is an address on the tape which is free,
- np contains a word made of $\mathtt{m} \times k$ "$\square$" symbols is used to "clean" the memory,
- r encodes the content of the k registers (but not ip).
- and $\mathtt{tp0}, \mathtt{tp1}$ are temporary registers.

r is organized as follows $\rho(\mathtt{r}) = \rho(r_1)\square^{k_1} \cdots \rho(r_k)\square^{k_k}$ where $k_i + |\rho(r_i)| = \rho(m)$. To get access to the value of register r_i, we perform the following operations:

`mov r M;`	move r in (free) memory
`mul` i `m tp0 ; add M tp0 tp0 ;`	computes the address of r_i
`load tp0 m tp0 ;`	load the content of r_i
`mov np M`	clean the memory

To push the value stored in tp0 corresponding to register r_i into r, we perform:

`mov r M;`	as above
`mul` i `m tp1 ; add M tp1 tp1 ;`	
`mov tp0 tp1 ;`	push tp0 in memory
`mul m` k `tp1 ; load M tp1 r;`	back in the register
`mov np M`	clean the memory

One may observe that these operations can be done in constant time. The management of m, M and np is facilitated by the following observation: the size of the result of each operation $\mathtt{op}(m, n)$ can be easily bounded by $O(|m| + |n|)$. Augmenting the values of the three registers m, M and np accordingly can be done in constant time. Using the program length of example 2, we can give an initial value to M and np. It is then routine to write the entire simulation. ■

IV. Self-modifying programs

Definition 5. *A pseudo-program* p *is said to be* self-modifying *whenever it is not stable. S denotes the set of self-modifying pseudo-programs and $N = B^* \backslash S$ the set of non-self-modifying pseudo-programs, that is of stable programs.*

In other words, for a self-modifying program, either the code of the pseudo-program has been modified during the execution, or the instruction pointer goes outside the code. Actually, there are some room for the definition of self-modification. One may argue, solution (1), that modifying the memory within $\sigma_0(0 \cdots |\mathtt{p}| - 1)$ corresponds to self-modification. However, there is no reason to restrict the program to its initial segment: indeed, a program can write a new instruction in another part of the memory, and then jump to this instruction. Solution (1) is too restrictive since it does not deal with some programs which dynamically transform their code. So, one may imagine to extend the scope of the definition to the entire memory. That is solution (2): it corresponds to any program which writes a new instruction in memory. But, again, since there is no clear distinction between data and instructions, it may happen that a bunch of data can be wrongly interpreted as an instructions. And then, solution (2) considers as self-modifying some programs which execute only instructions present at the begining.

Example 3. *A short (if not the shortest) self-modifying program is:*

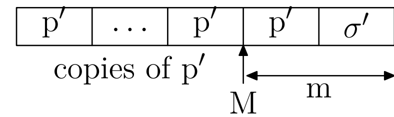
`cpy ip ap ;`	*gets the address of the current instruction*
`l_ccat "stop" r;`	*stores the word* stop *in* r
`mov r ap ;`	*rewrites the first instruction*
`jump ap`	

where the second "instruction" use the shorthand notation of Example 1. The jump instruction transfers the control to the first instruction which has been rewritten to stop.

Definition 6 (Running programs). *A program* p *is said to be* running *whenever for all computations $c_0(\mathtt{p}, w_1, \ldots, w_k) = (\rho_0, \sigma_0) \to (\rho_1, \sigma_1) \to^* (\rho_n, \sigma_n)$, the sequence $\rho_0(\mathtt{ip}), \ldots, \rho_n(\mathtt{ip})$ is increasing. A running program never goes back.*

Running stable programs are executed in a constant number of steps. Clearly, that subset of programs is not Turing-complete. But the set of self-modifying running programs is Turing-complete. This shows one of the fundamental difference between stable programs and self-modifying programs. To prove our proposition, we compile any stable program into a self-modifying program. Since stable programs are Turing-complete (Proposition 8), the conclusion follows.

Let us consider a stable program p. To avoid technicalities, we suppose that it is written with n instructions $\mathtt{I}_1 \cdots \mathtt{I}_n$, using registers ip $, r_1, \ldots, r_k$. We suppose furthermore that ip does not appear as the target of some instruction $\mathtt{op}\ \mathtt{r}_1\ \mathtt{r}_2\ \mathtt{r}_3$. We compile it using the same registers with 4 extra-registers: $\mathtt{M}, \mathtt{m}, \mathtt{tp0}, \mathtt{tp1}$. The principle is to write a program $\mathtt{p}'$ which simulates the instructions of p. Along the computation, the content of the r_i is equal to the original content, and the memory looks like



where M gives the address of $\mathtt{p}.\sigma'$ which correspond to the content of the simulated memory and where m gives the length of the occupied memory. Initialy, $\rho(\mathtt{M}) = |p'|$ and m is computed by the length proram.

Each instruction I of p is translated to c instructions (see the translation rules below). Consequently, we have $|\mathtt{p}'| = c|\mathtt{p}|$. The number of copies of $\mathtt{p}'$ in memory is given by ip div $|\mathtt{p}'|$, and the instruction pointer ip of $\mathtt{p}'$ corresponds to the execution of the instruction $(\mathtt{ip} \bmod |\mathtt{p}'|)/c$ of p. Now,

the rule of the translation are given by:

$$\begin{array}{rcl}
\texttt{op}\ r_1\ r_2\ r_3 & \mapsto & \texttt{op}\ r_1\ r_2\ r_3\\
\texttt{op}'\ r_1\ r_2 & \mapsto & \texttt{op}'\ r_1\ r_2\\
\texttt{op}''\ r_1 & \mapsto & \texttt{op}''\ r_1\\
\texttt{mov}\ r_1\ r_2 & \mapsto & \texttt{add}\ r_1\ \texttt{M}\ \texttt{tp0}\ ;\texttt{mov}\ \texttt{tp0}\ r_2\\
\texttt{load}\ r_1\ r_2\ r_3 & \mapsto & \texttt{add}\ r_1\texttt{M}\ \texttt{tp0}\ ;\texttt{load}\ \texttt{tp0}\ r_2\ r_3\\
\texttt{stop} & \mapsto & \texttt{stop}\\
\texttt{jump}\ r & \mapsto & \texttt{r_shift_mem}\ \texttt{M}\ \texttt{m};\texttt{mul}\ c\ r\ \texttt{tp0}\ ;\\
 & & \texttt{add}\ \texttt{M}\ \texttt{tp0}\ \texttt{tp0}\ ;\texttt{add}\ |\texttt{p}'|\ \texttt{M};\\
 & & \texttt{jump}\ \texttt{tp0}\\
\texttt{test}\ r\ \texttt{q} & \mapsto & \texttt{add}\ \texttt{ip}\ |\texttt{p}'|\ \texttt{tp1}\ ;\texttt{add}\ c\ \texttt{tp1}\ \texttt{tp1}\ ;\\
 & & \texttt{mul}\ c\ \texttt{q}\ \texttt{tp0}\ ;\texttt{add}\ \texttt{M}\ \texttt{tp0}\ \texttt{tp0}\ ;\\
 & & \texttt{r_shift_mem}\ \texttt{M}\ \texttt{m};\texttt{add}\ |\texttt{p}'|\ \texttt{M};\\
 & & \texttt{test}\ r\ \texttt{tp0}\ ;\\
 & & \texttt{jump}\ \texttt{tp1}
\end{array}$$

- op corresponds to ternary operators, op′ to binary operators and op′′ are unary operators (not, l_ccat , r_ccat),
- op m r_1 r_2 where m is an integer is a shorthand defined as in example 1,
- r_shift_mem M m shifts the memory content from $|\texttt{p}'|$ letters using M and m and make a new copy of p′ at address M,
- when ip is used as an operand of some instruction, we get its content through the instructions: mod ip $|\texttt{p}'|$ tp0 ; div tp0 c tp0 and replace ip by tp0,
- the management of m is not shown in the translation, but it is simple: at each step, multiply it by 2,
- to make all translations have exactly c instructions, we pad the shorter ones with dummy instructions cpy r r.

V. Computing non-self-modifying programs from pseudo-programs

Now, thinking of self-modifying programs as obfuscated forms of normal programs, one may argue that the meaning of a self-modifying program is its (one of) non-self-modifying form.

Definition 7 (Deobfuscating semantics)**.** *A* deobfuscating semantics *is a function $\psi : P \to N$ such that for all $\texttt{p} \in P$, we have $[\![\psi(\texttt{p})]\!] = [\![\texttt{p}]\!]$.*

To define an effective deobfuscating semantics, we have to show that the set of functions computed by pseudo-programs is Turing complete. There is nothing surprising with that result. However, to keep a constructive approach, and since some part of the definitions are used later on, we provide a complete proof of it.

For that sake, we introduce a slight variant of GOTO-programs as employed by Jones in [6]. We suppose given a finite set of variables $\texttt{X}_1, \dots, \texttt{X}_n$ ranging on words. A GOTO-program is then given by a list of instructions $1 : \texttt{I}_1, 2 : \texttt{I}_2, \dots, n : \texttt{I}_n$ with instructions being given by:

$$\begin{array}{rl}
\texttt{I} ::= & \texttt{X}_i := \texttt{nil} \mid \texttt{X}_i := a \mid \texttt{X}_i := \texttt{X}_j \mid \texttt{X}_i := \texttt{l_shift}\ \texttt{X}_j\\
 & \mid \texttt{X}_i := \texttt{ccat}\ \ \texttt{X}_j\ \texttt{X}_k \mid \texttt{if}\ a\ \texttt{goto}\ \ell \mid \texttt{stop}
\end{array}$$

where $a \in B$ and $\ell \in \mathbb{N}$. A configuration is given by a valuation of the variables and the address $n \in \mathbb{N}$ of the instruction to be executed. To denote a configuration, we use the notation $(\texttt{x}_1, \dots, \texttt{x}_n, \ell)$. Here, $\texttt{x}_i$ is the content of $\texttt{X}_i$ and ℓ is the current label of the instruction. If $\ell > n$ or $\ell = 0$ or ℓ labels a stop instruction, the machine stops. Otherwise, the semantics of instructions is a binary relation on configurations. Suppose that $\texttt{I}_\ell = \texttt{X}_i := \texttt{nil} \mid \texttt{X}_i := a \mid \texttt{X}_i := \texttt{X}_j \mid \texttt{X}_i := \texttt{l_shift}\ \texttt{X}_j \mid \texttt{X}_i := \texttt{ccat}\ \texttt{X}_j\ \texttt{X}_k$, then, $(\texttt{x}_1, \dots, \texttt{x}_n, \ell) \to (\texttt{x}_1, \dots, \texttt{x}'_i, \dots, \texttt{x}_n, \ell + 1)$ where $\texttt{x}'_i$ is computed from the $\texttt{x}_i$'s according to the right value of the expression. For $\texttt{I}_\ell = \texttt{if}\ a\ \texttt{goto}\ \ell'$, then $(a, \texttt{x}_2 \dots, \texttt{x}_n, \ell) \to (a, \texttt{x}_2, \dots, \texttt{x}_n, \ell')$ and otherwise $(\texttt{x}_1, \texttt{x}_2 \dots, \texttt{x}_n, \ell) \to (\texttt{x}_1, \texttt{x}_2, \dots, \texttt{x}_n, \ell + 1)$.

Proposition 8. *Any function computed by a GOTO-program can be computed by a program in N, and consequently by a program in P. Conversely, programs in P (and N) can be simulated by GOTO-programs.*

Proof: It is done by a direct simulation of instructions. We use the registers $\texttt{r}_i$ for the variables $\texttt{X}_i$ plus two extra registers, np contains the empty word, and tp1 is a temporary register. There is no explicit register for ℓ: actually ip will follow the flow of GOTO-instructions. Consider a GOTO instruction I, we associate the following instructions $\alpha(\texttt{I})$:

$$\begin{array}{rcl}
\texttt{X}_i := \texttt{nil} & \mapsto & \texttt{cpy}\ \texttt{np}\ \texttt{r}_i\\
\texttt{X}_i := a & \mapsto & \texttt{cpy}\ \texttt{np}\ \texttt{r}_i;\texttt{l_ccat}\ a\ \texttt{r}_i\\
\texttt{X}_i := \texttt{X}_j & \mapsto & \texttt{cpy}\ \texttt{r}_j\ \texttt{r}_i\\
\texttt{X}_i := \texttt{l_shift}\ \texttt{X}_j & \mapsto & \texttt{l_shift}\ \texttt{r}_j\ \texttt{tp1}\ ;\texttt{cpy}\ \texttt{r}_j\ \texttt{r}_i;\\
 & & \texttt{ccat}\ \texttt{tp1}\ \texttt{r}_j\ \texttt{r}_j\\
\texttt{X}_i := \texttt{ccat}\ \ \texttt{X}_j\ \texttt{X}_k & \mapsto & \texttt{ccat}\ \texttt{r}_j\ \texttt{r}_k\ \texttt{tp1}\ ;\texttt{cpy}\ \texttt{tp1}\ \texttt{r}_i\\
\texttt{stop} & \mapsto & \texttt{stop}\\
\texttt{if}\ a\ \texttt{goto}\ \ell & \mapsto & \texttt{cpy}\ \texttt{np}\ \ \texttt{tp1}\ ;\texttt{l_ccat}\ @\texttt{I}_\ell\ \texttt{tp1}\\
 & & \texttt{test}_a\ \texttt{r}_1\ \texttt{tp1}
\end{array}$$

where $@\texttt{I}_\ell$ in the translation of $\alpha(\texttt{if}\ a\ \texttt{goto}\ \ell)$ is an integer which will be instantiated according to the rest of the instructions. Given a Turing Machine $M = 1 : \texttt{I}_1, \dots, n : \texttt{I}_n$, we translate it as the concatenation of instructions $\alpha(\texttt{I}_1).\alpha(\texttt{I}_2) \cdots \alpha(\texttt{I}_n)$ where the "$@I_\ell$"s appearing in the translation are computed as follows. Let us call $@\texttt{I}_i$ the address of instruction $\texttt{I}_i$ in memory, that is the length of the string $\alpha(\texttt{I}_1) \cdots \alpha(\texttt{I}_{n-1})$. Let us write $\lfloor \texttt{I} \rfloor$ the size of its encoding where we dropped the addresses. That is for

instance $\lfloor \texttt{left} \rfloor = |\alpha(\texttt{left})|$ and $\lfloor \texttt{if}\ a\ \texttt{goto}\ \ell \rfloor = |\alpha(\texttt{if}\ a\ \texttt{goto}\ \ell)| - |\texttt{l_ccat}\ @\texttt{I}_\ell\ \texttt{tp1}|$.

The addresses verify:

$$\begin{cases} @I_1 = 0 & \\ @I_{i+1} = @I_i + \lfloor I_i \rfloor + \mathcal{K} \times |\ell| & \text{if } \texttt{I} = \texttt{if}\ a\ \texttt{goto}\ \ell \\ @I_{i+1} = @I_i + \lfloor I_i \rfloor & \text{otherwise} \end{cases}$$

In other words, we have again a fixpoint equation. It can be solved as in example 2.

For the other direction, the rules of the operational semantics show clearly that the successor relation is computable, and then GOTO-computable. ■

A. N and P are acceptable languages

Proposition 9. *The sets N and P of programs are acceptable languages in the sense of Rogers and Uspenski [7], [8].*

Proof: We have seen that both langages N and P are Turing-complete. We need to provide two more constructions, a specializer and a self-interpreter for N and P. We recall that the specializer S_n is defined by the equation $[\![S_n(\texttt{p}, x_0)]\!](x_1, \ldots, x_n) = [\![\texttt{p}]\!](x_0, \ldots, x_n)$. Refering to the operational semantics, both for N and P, we state that $S_n(\texttt{p}, x_0) = \texttt{p}.\Box.x_0$ solves the problem.

For the universal function, from Proposition 8, we know that there is a GOTO-program M such that computing M on $(\texttt{p}, x_1, \ldots, x_n)$ we get $[\![\texttt{p}]\!](x_1, \ldots, x_n)$. Translating this machine back (with the same Proposition 8), we get a program I_M such that $[\![I_M]\!](\texttt{p}, x_1, \ldots, x_n) = [\![\texttt{p}]\!](x_1, \ldots, x_n)$. ■

Remark 10. *From its definition, it is straightforward (but it must be observed) that the specializer S_n is efficient: that is,* $\texttt{time}(S_n(\texttt{p}, x_0)(x_1, \ldots, x_n)) = \texttt{time}(\texttt{p}(x_0, \ldots, x_n))$.

We end this part with Kleene's recursion theorem. In [9], we have shown its central role in computer virology. The theorem can be used as a compiler for viruses. In particular, we provided a classification of viruses by means of a stratification of some variants of the Theorem [10].

Theorem 11. *[Kleene's fixpoint] Given a computable function $g : B^* \times B^* \to B^*$, there is a program $\texttt{e}$ in N such that $[\![\texttt{e}]\!](x) = g(\texttt{e}, x)$.*

Proof: Suppose that g is a program for the function g. The pseudo-program $\texttt{p}_{1,2}$, by scanning the memory, pushes the first argument in register `out` and lets the memory unchanged. $\texttt{p}_{2,2}$ is the second projection, it pushes the second argument in register `out`. Finally, we suppose that `clean p` cleans the memory from the address stored in p. The function $x, y \mapsto g(S_1(x, x), y)$ is then computed by the following program:

$$\begin{array}{lcl} \texttt{q} & \triangleq & \texttt{l_ccat}\ k\ \texttt{p}.\ \text{//the length of q} \\ & & \texttt{p}_{1,2}; \texttt{cpy out tp0}\ ; \\ & & \texttt{p}_{2,2}; \texttt{cpy out tp1}\ ; \\ & & \texttt{clean p}; \texttt{r_ccat}\ \Box\ \texttt{tp0}\ ; \\ & & \texttt{ccat tp0 tp0 tp0}\ ; \\ & & \texttt{ccat tp0 tp1 tp1}\ ; \\ & & \texttt{mov tp1 p}; \texttt{g} \end{array}$$

where k is the length of q[3]. Defining $\texttt{e} = S_1(\texttt{q}, \texttt{q}) = \texttt{q}\Box\texttt{q}$, we have the equalities:

$$\begin{array}{rcl} [\![\texttt{e}]\!](x) & = & [\![S_1(\texttt{q}, \texttt{q})]\!](x) \\ & = & [\![\texttt{q}]\!](\texttt{q}, x) \\ & = & g(S_1(\texttt{q}, \texttt{q}), x) = g(\texttt{e}, x) \end{array}$$

■

B. Semantics by deobfuscation

Proposition 12. *The set S is Σ_1-complete.*

Proof: The formulation of Definition 5 shows that S is Σ_1. We show that it is actually complete. Take a TM M and call $\alpha(M)$ its translation according to Proposition 8 where one transforms the translation of `stop` to

$$\begin{array}{lcl} \texttt{stop} & \mapsto & \texttt{cpy ip tp1 .cpy np tp0 .} \\ & & \texttt{l_ccat "stop" tp0 .} \\ & & \texttt{mov tp0 tp1 .jump tp1} \end{array}$$

If the machine halts, one of the instructions `stop` is executed. The instruction `mov tp0 tp1` writes a `stop` instrution, and then we jump to this instruction. Consequently, the program is self-modifying. For a non-halting machine, as we have seen, the simulation is performed by a non-self-modifying program. Then, the machine halts iff its translation is in S, we get the desired result. ■

This result is important in our quest of deobfuscation as a semantics. Let us call $\psi : P \to N$, the deobfuscation semantics we are looking for. It is natural to say that the semantics of a non-self-modifying program is the program itself (since it is not obfuscated!). To sum up, we are looking for a function ψ such that:

(i) $[\![\psi(\texttt{p})]\!](x) = [\![\texttt{p}]\!](x)$,
(ii) for $\texttt{p} \in N$, $\psi(\texttt{p}) = \texttt{p}$.

Unfortunately, there is no such computable function. This is a corollary of Proposition 12. We prove it ad absurdum. Suppose that ψ is constructive. Then, we have $\texttt{p} \in S \Leftrightarrow \psi(\texttt{p}) \neq \texttt{p}$. Indeed, if $\texttt{p} \in S$, then $\psi(p) \in N$ implies that $\psi(\texttt{p}) \neq \texttt{p}$. Otherwise, $\texttt{p} = \psi(\texttt{p})$ by the requirement on ψ.

Consequently, the price of the effectiveness of the deobfuscation function is to have a less precise deobfuscation

[3] Again, we use a fixpoint argument to compute it.

notion: an effective deobfuscation semantics must modify some non-self-modifying programs.

Theorem 13 (Rogers [8]). *There is a computable isomorphism between any two acceptable languages, that is between two Turing complete programming languages equipped with a specializer and a universal function.*

As a corollary, since both N and P are acceptable languages, there is a computable procedure which sends any program $\mathtt{p} \in P$ to some program in N. This isomorphim actually defines a deobfuscating semantics as mentioned in the beginning of the Section.

This is, up to our knowledge, an original use of this theoretical result as a tool to deobfuscate programs. However, Rogers's construction has some drawbacks. First, whenever the procedure is effective, we have no ideas of its complexity. Second, and toughest point, this (deobfuscating) semantics does not provide a link between the complexity of the obfuscated form of a program and its deobfuscated one. In particular, it could happen that the computations of the deobfuscated form of a program takes much more time than its obfuscated form. This goes clearly against the intuition of (de-)obfuscation. One of the requirements of Rogers construction is that the morphism is actually bijective. This feature is meaningless in the present setting, where we only need a compilation procedure (neither necessarily injective, nor surjective).

Theorem 14. *There is a compilation procedure $\pi : P \to N$ such that:*

- *π is computable in polynomial time,*
- *for each program $\mathtt{p} \in P$, we have* $\mathtt{time}(\pi(\mathtt{p})) = O(\mathtt{time}(\mathtt{p}))$.

Proof: We use the first Futamura projection [11]. This construction is quite analogous to the use of a virtual machine that we use daily to analyse a malware in a safe environment.

Consider that we have a (relatively) efficient interpreter I_N^P of P programs (written in N), that is the program I_N^P verifies $[\![I_N^P]\!](p, d) = [\![p]\!](d)$ and $\mathtt{time}(I_N(p, d)) = O(\mathtt{time}(p(d)))$. Then, the following procedure solves the problem: $\pi : p \mapsto S_1(I_N^P, p)$. First, it can be computed in polynomial time. Indeed, I_N^P is a constant parameter and the definition of S_1 shows it is computable in polynomial time. Second, the compilation is correct:

$$\begin{aligned} [\![S_1(I_N^P, \mathtt{p})]\!](d) &= [\![I_N^P]\!](\mathtt{p}, d) \\ &= [\![\mathtt{p}]\!](d) \end{aligned}$$

And third, for the time complexity of the program $\mathtt{p}$, we have the equations:

$$\begin{aligned} \mathtt{time}(S_2(I_N^P, \mathtt{p})(d)) &= \mathtt{time}(I_N^P(\mathtt{p}, d)) \text{ see Remark 10} \\ &\leq O(\mathtt{time}(\mathtt{p}(d))) \text{ by hypothesis} \end{aligned}$$

So, the last point of the proof is to show the existence of such an interpreter. Given a pseudo-program with registers $\mathtt{r}_1, \ldots, \mathtt{r}_n$, we simulate it, using registers $\mathtt{r}'_1, \ldots, \mathtt{r}'_n$ plus some extra registers ip , tp0 , tp1 , K. The interpreter I is designed as:

```
ccatl K K    //the length of instructions
ccatl k1 tp1  //tp1 points to cpy tp2 ip'
load ip' K tp0    //load next instruction
switch tp0 with
  case load r1 r2 r3 -> load r'1 r'2 r'3
  case mov r1 r2 -> mov r'1 r'2
  case op r1 r2 r3 -> op r'1 r'2 r'3
  case shiftl r1 r2 -> shiftl r'1 r'2
  case shiftr r1 r2 -> shiftr r'1 r'2
  case test r1 r2 -> cpy r'2 tp2.
                     test r'1 tp1
  case jump r -> cpy r' tp2. jump tp1
  case stop -> stop
end_switch
add K ip ip.
jump k2
cpy tp2 ip'. jump k2
```

where k_1 corresponds to the address of the instruction `cpy tp2 ip'` and k_2 to the instruction `load ip' K tp0`. The switch construct is defined as follows.

$$\begin{array}{l} \mathtt{switch\ tp1\ with} \\ \quad \mathtt{case\ w1} \ \rightarrow \ \mathtt{e1} \\ \quad \vdots \\ \quad \mathtt{case\ wn} \ \rightarrow \ \mathtt{en} \end{array} \triangleq \begin{array}{l} \mathtt{test}_{w1}\ \mathtt{tp1\ m2.e1.jump\ k} \\ \vdots \\ \mathtt{test}_{wn}\ \mathtt{tp1\ k.en.jump\ k} \end{array}$$

where mi with $2 \leq i \leq n$ points to the instruction corresponding to the i-th test and k points to the address at the end of the construction.

It is clear that I is a non-self-modifying program. From the construction, one may observe that each instruction is simulated by a finite number of instructions. Consequently, the time loss of our simulation is constant for each instruction, more precisely, we have $\mathtt{time}(\mathtt{I}(\mathtt{p}, d)) = O(\mathtt{time}(\mathtt{p}(d)))$. And lastly, the program above is actually stable. We have seen in Section III that for these programs, we could have an encoding of registers at a constant cost. It is then routine to encode the interpreter with the right number of registers. ∎

VI. Conclusion

We have proposed a computational model which is close to the functionning of real machines. The program is loaded in memory, and can be changed along the computations.

This paper rises some open questions. First, we think that the notion of self-modification should be refined: one way is to use [4], where the authors show that typing can be used to characterize pseudo-programs. Secondly, we have shown that there are no deobfuscating procedure keeping the stable

programs constant. Linked to this question, can we find some tools to approximate both sets N and S, from above, or from below? Such techniques find an immediate application in the verification of the security of a computer systems.Finally, writing self-modifying program is difficult. We think that Kleene's recursion theorem is a major tool to build them. Indeed, fixpoint programs have access to their own code and, consequently, can manipulate it. Finally, running programs can be seen as kind of generalized traces.

References

[1] A. Issa, "Raw assault on a poly/metamorphic engine," in *EICAR*, 2009, pp. 173–184, industry paper.

[2] H. Cai, Z. Shao, and A. Vaynberg, "Certified self-modifying code," vol. 2007, 2007, pp. 66–77.

[3] S. Debray, K. Coogan, and G. Townsend, "On the semantics of self-unpacking malware code," 2008.

[4] J. Marion and D. Reynaud, "Surfing on code waves," LORIA, Tech. Rep., 2009.

[5] N. Jones, "Computer implementation and applications of kleene's S-m-n and recursive theorems," in *Lecture Notes in Mathematics, Logic From Computer Science*, Y. N. Moschovakis, Ed. Springer Verlag, 1991, pp. 243–263.

[6] ——, *Computability and Complexity: From a Programming Perspective*. Cambridge, MA, USA: MIT Press, 1997.

[7] V. Uspenskii, "Enumeration operators and the concept of program," *UMN*, vol. 11, 1956.

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

[9] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "On abstract computer virology from a recursion-theoretic perspective," *Journal in Computer Virology*, vol. 1, no. 3-4, 2006.

[10] ——, "A classification of viruses through recursion theorems," in *CIE*, ser. Lecture Notes in Computer Science, vol. 4497. Springer, 2007, pp. 73–82.

[11] Y. Futamura, "Partial evaluation of computing process– an approach to a compiler-compiler." in *Systems, Computers, Controls*, vol. 5, 1971, pp. 45–50.

Appendix

Refering to example 2, there is one issue with the k_i's which we discuss now. One may observe that they are defined by means of themselves. Indeed, consider for instance k_1. The length of the macro instruction `l_ccat` k_1 `r` depends on $|k_1|$, that is on k_1. So, the length of the program depends on k_1. But k_1 is defined as the length of the program plus one!

To solve this, we use a fixpoint equation. Refering to the definition of `l_ccat` n `tp` , the size of these instructions is $\mathcal{K} \times |n|$ where $\mathcal{K}$ has been defined as the length of instructions (see Section II). Let us introduce $\alpha = |$`l_ccat 1 ap`$|$, $\beta = |\alpha| + |$`load r ap p.test`$_\Box$ `p s.add ap out .add ap r.jump b`$|$ and $\gamma = \beta + |$`stop`$| + 1$. Consider now the functions:

$$\begin{aligned} f_1(x_1,x_2,x_3) &= (|x_1|+|x_2|+|x_3|) \times \mathcal{K} + \gamma \\ f_2(x_1,x_2,x_3) &= (|x_1|+|x_2|+|x_3|) \times \mathcal{K} + \beta \\ f_3(x_1,x_2,x_3) &= (|x_1|+|x_2|+|x_3|) \times \mathcal{K} + \alpha \\ f(x_1,x_2,x_3) &= (f_1(x_1,x_2,x_3), f_2(x_1,x_2,x_3), \\ & \quad f_3(x_1,x_2,x_3)) \end{aligned}$$

One may observe that (k_1, k_2, k_3) is a fixpoint for the function f. To compute it, we use the algorithm:

```
Addr = [1,1,1]
Addr' = Addr
while(Addr' != Addr) do
      Addr = Addr'
      Addr' = f(Addr) //with f defined above
  od
return Addr
```

If the algorithm ends, then, the result is a fixpoint. Let us prove that the algorithm terminates. One may observe that f is contracting for sufficiently large values. Indeed, let us write $\delta(m,n) = \max(m-n, n-m)$. Whatever $c > 0$ is, by intermediate value theorem, for all $x, y > \dfrac{1}{c \times \ln(|B|-1)}$, one has $\delta(|x|,|y|) = \delta(\log_{|B|-1}(x), \log_{|B|-1}(y)) \leq \dfrac{1}{c} \times \delta(x,y)$. We introduce the distance $\delta((x_1,x_2,x_3),(y_1,y_2,y_3)) = \delta(x_1,y_1) + \delta(x_2,y_2) + \delta(x_3,y_3)$. Take $c = \dfrac{1}{6 \times \mathcal{K}}$, we compute $\delta(f(x_1,x_2,x_3), f(y_1,y_2,y_3))$:

$$\begin{aligned} &= 3 \times \mathcal{K} \times \delta(|x_1|+|x_2|+|x_3|, |y_1|+|y_2|+|y_3|) \\ &\leq 3 \times \mathcal{K}(\delta(|x_1|,|y_1|) + \delta(|x_2|,|y_2|) + \delta(|x_3|,|y_3|) \\ &\leq 3 \times \mathcal{K} \times \frac{1}{c} \times \delta((x_1,x_2,x_3),(y_1,y_2,y_3)) \\ &\leq \frac{1}{2} \times \delta(x_1,x_2,x_3),(y_1,y_2,y_3)) \end{aligned}$$

As a consequence, the algorithm converges. Moreover, we can state that the convergence is geometric wrt the input. Consequently the compilation procedure can be performed in polynomial time.