Abstract Self Modifying Machines

Hubert Godfroy * and Jean-Yves Marion

Université de Lorraine, CNRS and Inria
LORIA
firstname.lastname@loria.fr

Abstract. We describe a new framework for self-modifying programs, that is programs which can execute what they have themselves written. On the first hand, we use an abstract machine which makes explicit some typical behavior, such as turning data into executable code and vice versa. Moreover memory is also separated between data (what we can read and write) and code (what we can execute). On the other hand, we add another level of granularity in memory location, in order to deal with the problem of interpreting intensional behavior of the program more easily.

1 Introduction

Self-modification shifts the current programming language theory paradigm. In textbooks, see for example Winskel's [11], programs are defined by fixed text. The (text of the) program is available and invariant with regard to any execution. The theory leans on a clear distinction between programs and data. This separation between data and programs is also reflected in models of computation. In the case of Turing machines, or of Random Access memory, the program is "wired" inside the control unit. The case of abstract machines for functional languages is a bit different and may be seen as an exception. Indeed, functions are seen as values and data as terms (e.g. Church numerals) but this apparent confusion between data and programs is an artefact rather than a fundamental property of abstract machines. Indeed, there is (always) a clear distinction between data and terms at higher semantics levels.

There are models of computation in which data are first order citizens like Random access stored programs (RASP) [4, 5]. However, RASP are not satisfactory because self-modifying behaviors are hidden.

Our objective is to devise a model of computation, which has self-modifying mechanisms. There are at least two dual motivations. The first is to compile a program into a self-modified code. Nowadays, a lot of programs protects themselves against analysis by hiding their behaviors thanks to packers. A packer compresses (and encrypts) a program preserving its semantics. The result is wrapped with a short decompression routine. At runtime, this routine unpacks the code and then it transfers the control to the initial code. Unpacker builds

* This work is supported by French Direction Générale de l'Armement.

a quite simple, but efficient, self-modifying procedure. We could think of much more involved self-modifying mechanisms. For this, we should make a clear distinction between two notions: *programs* and *codes*. Programs will be considered invariant unlike codes which are executable data. Given a programming language $\mathscr{L}$, a *program* is compiled into a *code*, which is designed to run on a given model of computation. This model of computation may have features which are not in $\mathscr{L}$ semantics like the ability of modifying a code at runtime. Our abstract machine ASM_2 would provide an intermediate stage for compilation where self-modifying mechanisms could be used.

The other motivation is the analysis of self-modifying codes, because nowadays all malware use this kind of obfuscation [2, 1]. The goal is then to recover from codes the original programs in order to understand its meaning.

This work is also motivated by the work of Jones et al. [6] on biological systems. In this model, computations are made by a net of blobs where data are programs, with a strong notion of locality. There are models of self-modifying codes like Della Preda et al. [9] where a fixed-point semantics of self-modifying programs is defined, or like [3, 7] where a framework developed in the COQ proof assistant is constructed. Finally, self-modifying codes are related to Kleene's second fixed point theorem as it is explained in [8].

2 Abstract Self Modifying Machines

The salient feature of Abstract Self Modifying Machines (ASM_2) is the memory architecture. The memory consists of two zones of registers: (i) the *executable zone* containing *code registers* and (ii) the *data zone* containing *data registers*. The access to both zones are restricted and are different. We can run every register in the executable zone, but we cannot write on it. We can write on every register in the data zone, but we can not run it. This apparent separation between data and codes is broken by transfer instructions. We have an instruction which transfers a data register into the executable zone, together with an instruction which runs any code register. As a result, we can construct a code inside the data zone. When it is ready, we can transfer it to the executable zone and run it. This is quite similar to a decompress/decryption loop currently used by unpackers. As a result, a register contains a word, which may be decoded as a sequence of instructions when it is put in the executable zone. This is a typical behavior of a self-modifying code.

Formally, a memory is a pair of zones $(\mathbf{X}, \mathbf{D})$ where $\mathbf{X}$ and $\mathbf{D}$ are finite sequences of registers referring to the *executable zone* and to the *data zone*. Each register is identified by a natural number. Each register contains a word on some alphabet Σ. Concatenation over Σ^* is noted $\bullet$. Moreover, we have a set of instructions $\mathcal{I}$ and an encoding partial function $\mathcal{E} : \Sigma^* \to \mathcal{I}$

A state $\langle (\mathbf{X}, \mathbf{D}) \mid \mathtt{RP} \mid \mathtt{IP} \rangle$ consists of a memory $(\mathbf{X}, \mathbf{D})$, a register pointer `RP` and an instruction pointer `IP`. The machine counter is broken into a register pointer `RP` and an instruction pointer `IP`. Hence, the current instruction starts from address pointed by `IP` of the register pointed by `RP`. At each step, the current

instruction is run. The next instruction depends on the instruction kinds. The instruction pointer IP then moves accordingly. So, we write $\texttt{IP} = \texttt{IP} + 1$ to increment the register IP of the length of an instruction opcode, i.e. to move forward of one instruction, and $\texttt{IP} = n$ to point the n^{th} instruction.

The execution flow may move from one register to another by changing the values of the register pointer RP.

If the register n is in executable zone (resp. data zone), we use $\mathbf{X}[n]$ (resp. $\mathbf{D}[n]$) to denote it. The notation $\mathbf{D}[n \leftarrow u]$ denotes the data zone $\mathbf{D'}$ such that $\mathbf{D'}[m] = \mathbf{D}[m]$ for all m except m where $\mathbf{D'}[n] = u$.

Instr.	Meaning
`move` a, n	Write the letter a at the end of $\mathbf{D}[n]$
`pop` n	Pop the letter on the top of $\mathbf{D}[n]$
`jump` n	Go to the instruction n
`case` n	Conditional jump depending on $\mathbf{D}[n]$
`exec` n	Control transfer to register $\texttt{RP} = n$ and $\texttt{IP} = 0$
`activate` n	Activate $\mathbf{D}[n]$ and $\texttt{IP} = 0$
`inactivate` n	Inactivate $\mathbf{X}[n]$

Fig. 1. ASM_2 instruction set

2.1 Transition rules

We will now describe the transition relation. The rules are divided in two categories and are given in the shape : $\langle(\mathbf{X}, \mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle \xrightarrow{\texttt{inst}} \langle(\mathbf{X'}, \mathbf{D'}) \mid \texttt{RP'} \mid \texttt{IP'}\rangle$ where `inst` is the instruction beginning at the address pointed by IP is the code register $\mathbf{X}[\texttt{RP}]$.

Calculation rules Calculation rules are devised to compute in an environment where data and codes are separated. No executable registers are modified.

– We concatenate the letter a at the end of the data register $\mathbf{D}[n]$:

$$(\textsc{Move})\ \langle(\mathbf{X}, \mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle \xrightarrow{\texttt{move}\, a,n} \langle(\mathbf{X}, \mathbf{D}[n \leftarrow \mathbf{D}[n] \bullet a]) \mid \texttt{RP} \mid \texttt{IP} + 1\rangle$$

– We erase the letter a on the top of the data register $\mathbf{D}[n]$:

$$(\textsc{Pop})\ \langle(\mathbf{X}, \mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle \xrightarrow{\texttt{pop}\, n} \langle(\mathbf{X}, \mathbf{D}[n \leftarrow u]) \mid \texttt{RP} \mid \texttt{IP} + 1\rangle \text{ where } \mathbf{D}[n] = a \bullet u$$

– We jump to the instruction m^{th} instruction in the register $\mathbf{X}[\texttt{RP}]$. For this, $\texttt{IP}$ is set to m:

$$(\textsc{Jump})\ \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle \xrightarrow{\texttt{jump}\, m} \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid m\rangle$$

– If the value on the top of the register $\mathbf{D}[n]$ is a, then we run the next instruction, otherwise we jump to the second instruction:

$$\begin{aligned}(\textsc{Case})\ \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle &\xrightarrow{\texttt{case}\, an} \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}+1\rangle \text{ if } \mathbf{D}[n] = a \bullet u\\ \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle &\xrightarrow{\texttt{case}\, an} \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}+2\rangle \text{ otherwise}\end{aligned}$$

Code motions

– We run the code register n starting at the first instruction (i.e. $\texttt{IP} = 0$):

$$(\textsc{Exec})\ \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle \xrightarrow{\texttt{exec}\, n} \langle(\mathbf{X},\mathbf{D}) \mid n \mid 0\rangle$$

– The data register n is moved to the executable zone. As a result, it can be executed thanks to the above rule:

$$(\textsc{Activate})\ \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle \xrightarrow{\texttt{activate}\ n} \langle(\mathbf{X} \oplus \mathbf{D}[n], \mathbf{D} \setminus \mathbf{D}[n]) \mid \texttt{RP} \mid \texttt{IP}+1\rangle$$

– The code register n is put into the data zone. It cannot be run but it can be modified:

$$(\textsc{Inactivate})\ \langle(\mathbf{X},\mathbf{D}) \mid \texttt{RP} \mid \texttt{IP}\rangle \xrightarrow{\texttt{inactivate}\ n} \langle(\mathbf{X} \setminus \mathbf{X}[n], \mathbf{D} \oplus \mathbf{X}[n]) \mid \texttt{RP} \mid \texttt{IP}+1\rangle$$

Typically, program halts if $\texttt{IP}$ "goes below" the current code. There are other reasons to halt. The register $\texttt{IP}$ may point to a word, which is not the opcode of an instruction. Another way to halt is when a jump is performed with an address outside of the register addresses.

2.2 Example

We present here a very simple self-modifying program which decrypts the register C with key in register K (simply a XOR), puts result in register M and then executes it.

We suppose that our alphabet Σ is the set $\{0, 1, \#\}$. We need four registers I, C, K and M whose last tree ones have already been described, the first is the initialising register, which is executed first. In the beginning, C, K and M belong to $\mathbf{D}$ and I to $\mathbf{X}$. The register and instruction pointers are initialized to I and α_0. The first state of the machine is thus $\langle(\{I\}, \{C, K, M\}) \mid I \mid \alpha_0\rangle$. In the following, if the top of a register is $\#$, the register is said empty. We suppose that M is empty at the beginning.

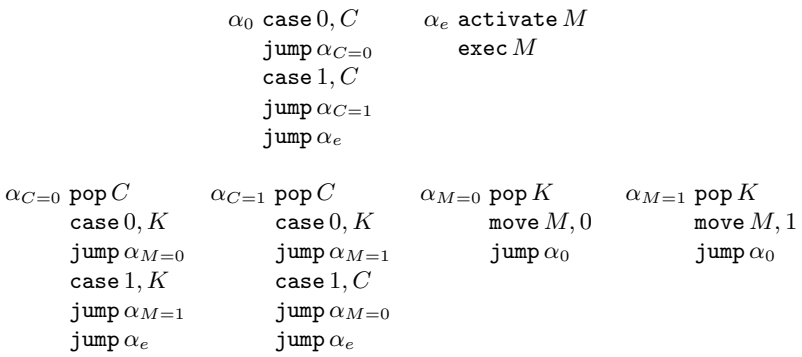

Fig. 2. Self-modifying program in ASM_2

The program begins by inspecting the first element of the register C (the encrypted message) and decrypts it using the first element in the register K (the key) by jumping to either address $\alpha_{C=0}$ or address $\alpha_{C=1}$. After having been decrypted, it writes the result in the register M (addresses $\alpha_{M=0}$ and $\alpha_{M=1}$) end goes back to the beginning. If the register C is empty, is goes to the evaluation procedure at address α_e which starts the execution of register M.

The self-modifying behavior is clearly exposed thanks to explicit instructions `activate` and `exec`.

3 Further works

Giving this abstract machine, we have first adapted a measure of self-modifying behavior given in [10], the waves, which relates executed codes with codes which wrote them. It appears that this framework is more appropriate to deal with the notion of waves thanks to the time separation of actions of writing and executing, with instruction `activate`. It makes it possible to more precisely define the moment when a set of addresses can be seen as a program. Annotations make it easier to sequentialize self-modification.

On the other hand, this machine adds a new granularity on the top of addresses, the registers. This allows to structurally see the organisation of the program and infer more easily the intentional behavior of the program. Gathering instructions into high level semantics functions is an important step in the comprehension of programs.

Another field of investigation is to reconstruct self-modifying programs without self-modifications by "unfolding" successive waves of code. This study is based on recovering the executed code of program from execution traces, for any trace. Working with our machine is pleasant because, thanks to separation

between data and code, we can more easily define what a program is at a given moment than with x86 semantics where code and data are not distinguishable.

By using this framework, we are also developing a static representation of self-modifying programs based on waves and which takes advantage of larger granularity (registers) of our machine. Indeed, this gives the ability to gather instructions not executed at the same time (for instance with different inputs) hoping that each register contains one and only one high level function. Giving that, we can decide to see statically each program as the tree of all registers which was written and then executed for all inputs, related according to writing actions. This static point of view can be seen as a specification of self-modifying programs, because it describes how each registers are created, that is to say, which register creates another. We can imagine using this specification to design a compilation function taking a non self-modifying program and returning a equivalent self-modifying program in ASM_2 with respect to the given specification.

References

1. Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier, and Aurélien Thierry. Code synchronization by morphological analysis. *Malware*, 2012.
2. Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier, and Aurélien Thierry. Analysis and diversion of duqu's driver. *Malware*, 2013.
3. Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. *PLDI*, 2007.
4. Calvin C. Elgot and Abraham Robinson. Random-access stored-program machines, an approach to programming languages. *Journal of the Association for Computing Machinery*, 11(4):365–399, October 1964.
5. Juris Hartmanis. Computational complexity of random access stored program machines. *Mathematical Systems Theory*, 1971.
6. L. Hartmann, N.D. Jones, J.G. Simonsen, and S.B. Vrist. Programming in biomolecular computation: Programs, self-interpretation and visualisation. *Scientific Annals of Computer Science*, 21(1):73–106, 2011.
7. Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ml and assembly. *Principles of programming languages*, 2011.
8. Jean-Yves Marion. From turing machines to computer viruses. *Phil. Trans. R. Soc. A*, 370:3319–3339, July 2012.
9. Mila Dalla Preda, Roberto Giacobazzi, and Saumya Debray. Modeling metamorphism by abstract interpretation. *Theoretical Computer Science*, 2012.
10. Daniel Reynaud. *Analyse de codes auto-modifiants pour la sécurité informatique*. PhD thesis, INPL, 2010.
11. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.