J Comput Virol
DOI 10.1007/s11416-008-0084-2

ORIGINAL PAPER

Code obfuscation techniques for metamorphic viruses

Jean-Marie Borello · Ludovic Mé

Received: 15 June 2007 / Revised: 27 October 2007 / Accepted: 29 January 2008

Abstract This paper deals with metamorphic viruses. More precisely, it examines the use of advanced code obfuscation techniques with respect to metamorphic viruses. Our objective is to evaluate the difficulty of a reliable static detection of viruses that use such obfuscation techniques. Here we extend Spinellis' result (IEEE Trans. Inform. Theory, **49**(1), 280–284, 2003) on the detection complexity of bounded-length polymorphic viruses to metamorphic viruses. In particular, we prove that reliable static detection of a particular category of metamorphic viruses is an $\mathcal{NP}$-complete problem. Then we empirically illustrate our result by constructing a practical obfuscator which could be used by metamorphic viruses in the future to evade detection.

1 Introduction

Cohen's seminal work [10] defines computer viruses as self-reproducing programs. Since then, viral techniques have not stopped evolving and progressing, mainly to bypass detection algorithms and software. This evolution has given birth to more and more complex viruses, on which one of the most sophisticated form is metamorphism.

By metamorphic virus, we consider only viruses able to modify their own codes in order to produce viral copies that are as syntactically different as possible from their parents. Consequently, each binary offspring of a metamorphic virus aims to differ from its original form. We consider metamorphism as the self-reproduction technique used by such viruses. Metamorphism operates in two different steps: firstly, the viral program extracts the semantics of its own code in order to have its model at one's disposal. Secondly, the viral program applies obfuscation transformations to this model in order to produce a code as different as possible from its parent code, while keeping the same behavior.

As metamorphic variants of a virus aim to look different, pattern matching does not appear to be a reliable detection technique. Based on the assessment that a metamorphic virus is able to model itself before self-reproducing, a number of works on antiviral detection put forward the hypothesis that another program (e.g. an antivirus software) should also be able to model such a virus for detection purposes [4,5,18,27]. Our aim is to study the validity of this hypothesis with respect to the two existing detection techniques: static-based and dynamic (or behavior-based) detection.

In this paper, the first part of our research work will only address static detection. Here, static analysis is conceived as a process whose goal is to extract the semantics of a given program without any help of code execution. The global analysis of the reproduction cycle of a metamorphic virus would clearly exceed the scope of a single paper. That is the reason why the present paper will only consider obfuscation transformations. In other words, we will focus exclusively on the obfuscation techniques that could be envisaged by metamorphic viruses in the future. In particular, we will try to determine the possible, practical consequences of the use of such techniques in terms of antiviral detection capabilities.

The paper is organized as follows. Section 2 deals with obfuscation techniques and metamorphic virus detection. In Sect. 3, which is more formal, we expose the theoretical limitations that a detection algorithm is bound to be confronted

J.-M. Borello (✉)
CELAR, BP 7419, 35174 Bruz Cedex, France
e-mail: jean-marie.borello@dga.defense.gouv.fr

J.-M. Borello
ESAT, Laboratoire de Virologie et de cryptologie,
BP 18, 35998 Rennes, France

L. Mé
SUPELEC, SSIR (EA 4039), Rennes, France

Table 1 Obfuscation techniques used in known metamorphic codes

	EVOL (2000)	ZMIST (2001)	ZPERM (2000)	REGSWAP (2000)	METAPHOR (2001)
Instruction substitution					✓
Instruction permutation	✓	✓			✓
Variable substitution	✓	✓		✓	✓
Dead code insertion	✓	✓			✓
Changing the control flow		✓	✓		✓

with. Finally, in Sect. 4, we present a practical obfuscation approach which is likely to be used by metamorphic viruses in the future. This approach which aims to bypass static detection takes into account most of the limitations presented in the previous section.

2 Code obfuscation techniques and metamorphic virus detection

This first section presents code obfuscation techniques used by existing metamorphic viruses. It subsequently presents the main developments in obfuscation. Finally, this section addresses the detection of such viruses.

2.1 Code obfuscation techniques used by metamorphic viruses

From a practical point of view, code obfuscation consists in making a program as "unintelligible" as possible. This practice has essentially grown with the widespread use of high-level programming languages such as .NET and JAVA. Indeed, as far as high-level programming languages are concerned, the byte code format which is produced at the time of compilation, still contains the whole program information. Consequently, it is possible to go back to the program source and then to the underlying algorithms themselves. Code obfuscation tries to thwart decompilation in order to protect source codes.

It is worth mentioning that code obfuscation operates on both the program data and control flow. A detailed taxonomy of obfuscation techniques can be found in [11]. Below are the obfuscation techniques that are particularly used by metamorphic viruses:

– data flow obfuscation (instruction substitution, instruction permutation, *dead code* or *garbage code* insertion, variable substitution…);
– control flow obfuscation (changing the control flow).

Table 1 summarizes the main viruses which make use of these obfuscation techniques.

We briefly present the first three basic techniques of Table 1 which are well-documented [14,22,25].

– **Instruction substitution** consists in replacing a given instruction block with another instruction block while keeping the same code semantics.
– **Instruction permutation** consists in modifying the instruction execution order in a program while keeping the same semantics.
– **Variable substitution** consists, at the binary code level, in either exchanging the used registers or in modifying the scope of the variables by using local or global variables instead of registers.

Dead code insertion and control flow obfuscation are more interesting techniques. We illustrate these two techniques on which we base our obfuscation approach in Sect. 4.

– **Dead code insertion** consists in introducing useless codes into the program. In a similar way, it can be composed of more complex instructions that are never executed. Table 2 gives some examples of "dead codes". The left column lists useless instructions which are semantically equivalent to the `NOP` (No Operation) instruction. The exact meaning of each instruction is given in the right column. The first line corresponds to adding the value `0` to the `Reg` register. In the second line, the `Reg` register self-affects with its own value. The third line performs a logical "OR" on a `Reg` register with the null integer. While the last line applies a logical "AND" on the `Reg` register with the -1 value.
– **Changing the control flow** consists in modifying the execution flow of a program by introducing conditional

Table 2 Examples of "dead code"

Rules	Meaning
`ADD Reg,0`	`Reg ← Reg + 0`
`MOV Reg,Reg`	`Reg ← Reg`
`OR Reg,0`	`Reg ← Reg	0`
`AND Reg,-1`	`Reg ← Reg & -1`

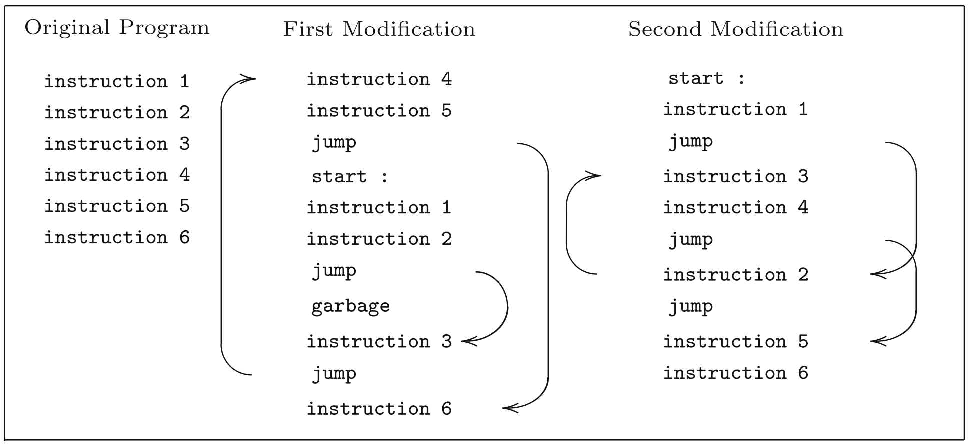


Fig. 1 Examples of control flow modification

or unconditional branching instructions, while preserving the program result. Figure 1 gives an example of such control flow modification by inserting unconditional branching instructions. An original code made of sequential instructions can be replaced by the two other codes whose execution flow has been modified so as to no longer be sequential (first and second alterations). In both cases, the same sequence of instructions is executed.

2.2 Main developments in obfuscation

Instead of presenting all the main contributions in the field of obfuscation schemes, we use Collberg's work [11] as reference. Our goal is to present only the notions useful in obfuscating a metamorphic virus. First, the authors define a metric commonly used to evaluate the efficiency of obfuscators. We sum them up as follows:

– the **potential** which evaluates the understanding complexity of an obfuscated code during human-driven analysis;
– the **resilience** which measures the complexity of the inverse operation (deobfuscation) by means of automatic tools (software);
– the **cost**, which defines the price we have to pay in terms of computing time and memory space required for the analysis.

It is easy to see that in the case of a virus the main interesting criteria is the resilience. In fact, resilience can informally be viewed as the difficulty to detect a metamorphic virus. According to this criterion, we are interested in control flow transformation as developed in [12]. This study presents the use of opaque predicates to increase the resilience of an obfuscated program. Broadly speaking, a predicate P is said to be opaque if it has a property q known to the obfuscator but which is hard for a deobfuscator to deduce. Our goal is to transpose such control-altering transformations from general purpose to metamorphic viruses in order to try to evaluate the detection difficulty (resilience). This idea was inspired by the contrast between obfuscations used in well-known viruses as presented in Sect. 2.1 and by advanced control flow transformations based on Collberg's work.

2.3 Detection techniques of metamorphic viruses

We will now address the issue of metamorphic virus detection and the techniques used by the existing antivirus software. Very few technical details are published on the existing tools. However, some works dealing with the reliability of detecting known viruses have pinpointed the severe limitations of existing antivirus software [8,9]. In fact, the tested tools have proved to still be inefficient at detecting even the most trivial obfuscation techniques, such as dead code insertion or instruction substitutions. These results show that simple detection pattern matching—looking for a fixed byte pattern which represents the virus signature—remains the main techniques used nowadays. This has been recently confirmed by more recent works [15,16].

However, it is a very well-known fact that perfect obfuscation is impossible to achieve. That is, the semantics of a program cannot be perfectly hidden [2,3]. In other words, given a fixed obfuscated program, its semantics can always be recovered. Nevertheless, the semantics extraction cannot be automated (Rice theorem [21]). For that purpose, some detection prototypes aim at building an optimized model of the program by using code compiling optimizations [1]. As shown in [7,13], the same techniques can be directly used to model binary codes. Such a modeling process is performed in two steps:

1. In the first step, a control flow graph of the program is built. This graph in fact is a model of the possible program execution flows.

2. In the subsequent step, the data flow is analyzed and simplified. It is thus possible to use this feedback on the control flow graph in order to further simplify it.

The optimized control flow graph represents a model for the program under analysis. Detecting a metamorphic virus can be viewed as checking whether or not the model we obtain corresponds to this virus. The construction of such a model is detailed in C. Cifuentes' thesis [7]. Many approaches have been proposed [5,8,24] to detect the state of the art viruses. The following section demonstrate the difficulty of reliable detection and especially in the case of a particular category of metamorphic viruses.

3 Limitations of reliable static detection of metamorphic codes

3.1 Impossibility of perfect static detection

In this section, we address the issue of evaluating the difficulty of perfect detection of metamorphic viruses, at the theoretical level. For that purpose, we will give some useful notations and definitions.

Notations: We consider two programs (algorithms) A and B which are respectively defined on the domains $\mathcal{D}_A$ and $\mathcal{D}_B$ respectively. A and B are said to be functionally equivalent, and we note $A \equiv B$, if and only if they output the same result on the same inputs, in other words:

$$\begin{cases} \mathcal{D}_A = \mathcal{D}_B, \\ \forall x \in \mathcal{D}_A,\ A(x) = B(x). \end{cases}$$

We give a first definition of perfect detection in terms of functional equivalence as follows:

Definition 1 The program D_V perfectly detects a metamorphic virus V if and only if for any program P,

$$\begin{cases} D_V(P) \text{ returns ``}true\text{''} \quad \text{if } P \equiv V, \\ D_V(P) \text{ returns ``}false\text{'' otherwise.} \end{cases}$$

Proposition 1 *There exists no algorithm which is able to determine for two given non null programs P and P' whether $P' \equiv P$.*

This Proposition 1 is just a particular instance of the Rice's theorem [21]. The direct consequence of Proposition 1 is that the perfect detection of a metamorphic virus as defined in Definition 1 is an undecidable problem.

3.2 Limitations of reliable static detection of a particular category of metamorphic codes

Let us suppose that all the (graph) paths with respect to a program P are potentially executable. Despite the fact that it is not systematically valid, this hypothesis is frequently assumed during static analysis [1]. Even under this hypothesis, the problem of functional equivalence remains an undecidable one. That is the reason why, we will consider a more restrictive category of metamorphic virus obtained when modifying the program control flow.

This paragraph starts with the presentation of a code transformation. Then, we prove that the problem of determining if a program is obtained by another program transformation is an $\mathcal{NP}$-complete problem. Finally, we deduce from this result the difficulty of detecting a particular category of metamorphic viruses.

Build of a program transformation $\mathcal{T}$

Definition 2 Let P be a program composed of n consecutive instructions $I_1 I_2 \ldots I_n$. We define a transformation $\mathcal{T}$ of the program P as follows:

- P is split into k blocks denoted P_i. Each block contains a random non null number of consecutive instructions and ends up with a sequential instruction (different from a conditional branching instruction).
- We consider a permutation σ on the set $[1, k]$. For each block P_i, we define a new block $P'_{\sigma(i)}$ as follows: each block $P'_{\sigma(i)}$ contains exactly the same instructions as the corresponding block P_i and ends up with two conditional branching instructions toward two other blocks P'_j and P'_l. This transformation produces a program P' from an original program P. We denote $P' = \mathcal{T}(P)$.

We then define a set of obfuscators (k-obfuscators) denoted $\mathcal{O}_k$ as follows: a transformation $\mathcal{T}$ is a k-obfuscator, denoted $\mathcal{T} \in \mathcal{O}_k$ if and only if $\forall i \in [1, k]$ during its execution, the last output of $P'_{\sigma(i)}$ is $P'_{\sigma(i+1)}$. By last output, we mean the destination of the conditional branching in block $P'_{\sigma(i)}$.

It is worth noting that if $\mathcal{T} \in \mathcal{O}_k$ then $\mathcal{T}(P) \equiv P$ and thus $\mathcal{T}$ is indeed an obfuscator.

Determining if $P' = \mathcal{T}(P)$ is $\mathcal{NP}$-complet

Proposition 2 *Under the assumption that each program path is potentially executable, for any pair of programs P and P', determining whether there exists $\mathcal{T} \in \mathcal{O}_k$ such that $P' = \mathcal{T}(P)$ is an $\mathcal{NP}$-complete problem.*

The proof given hereafter relies on a polynomial time reduction inspired from Landi's work [17]. More precisely, it relates to proof given in [19,26].

Proof 1. The problem is in $\mathcal{NP}$. Indeed, for every given pair of programs P and P', a non deterministic algorithm can check in polynomial time whether there is a

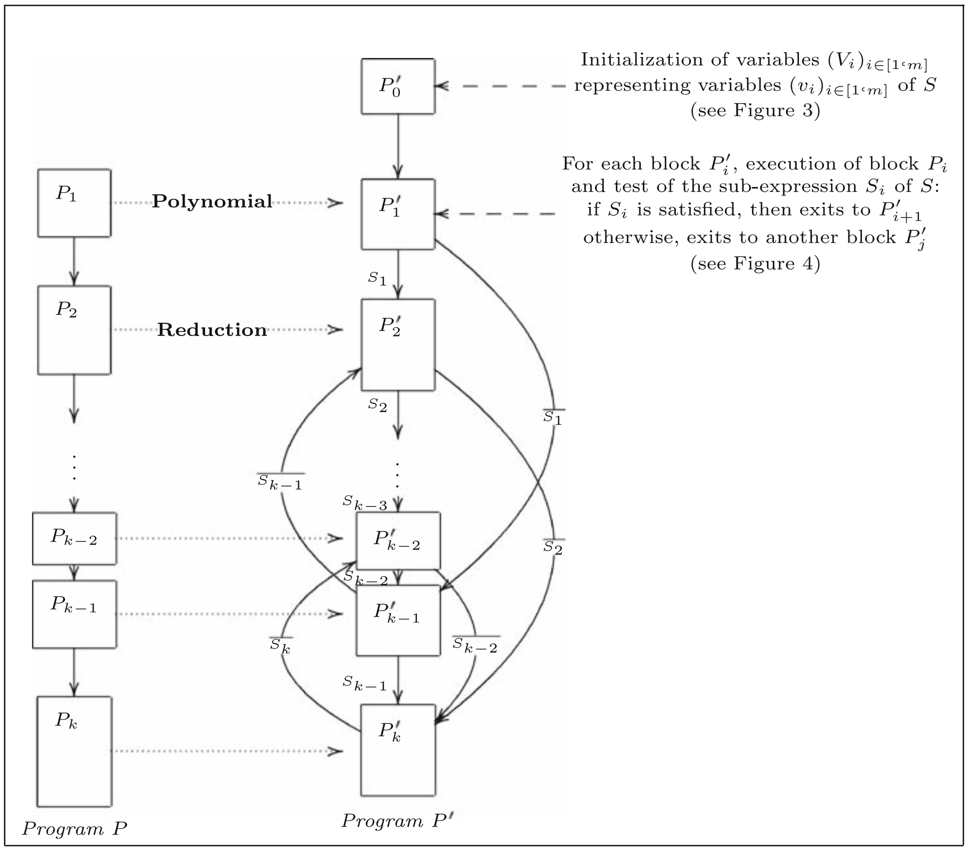

Initialization of variables $(V_i)_{i\in[1;m]}$ representing variables $(v_i)_{i\in[1;m]}$ of S (see Figure 3)
For each block P'_i, execution of block P_i and test of the sub-expression S_i of S: if S_i is satisfied, then exits to P'_{i+1} otherwise, exits to another block P'_j (see Figure 4)
Polynomial
Reduction
Program P
Program P'

Fig. 2 Construction scheme of the obfuscated program P' (*right*) from the program P (*left*)

transformation $\mathcal{T}$ which complies with Definition 2 such that $P' = \mathcal{T}(P)$. Then this algorithm can also verify in polynomial time, whether there exists an output from every block $P'(i)$ such that $\mathcal{T} \in \mathcal{O}_k$.

2. Let S be an instance of the $\mathcal{NP}$-complete problem 3-SAT, and P a program. We are going to prove that it is possible to build in polynomial time with respect to the size of S, a program P' ($P' = \mathcal{T}(P)$) such that S is satisfiable if and only if $\mathcal{T} \in \mathcal{O}_k$. An instance S of the 3-SAT problem has the following form:
$S = \bigwedge_{i=1}^{n}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$,
$$\begin{cases} \{v_1, v_2, \ldots, v_m\} \text{ a set of Boolean variables} \\ \forall(i,j) \in [1,n] \times [1,3], \exists k \in [1,m], \left| \begin{array}{l} l_{i,j} = v_k \text{ or} \\ l_{i,j} = \overline{v_k} \end{array} \right. \end{cases}$$
We build the family of u_i, denoted $(u_i)_{i\in[1,k]}$ such that it is a partition of consecutive elements of the set $[1, n]$.
$(\underbrace{1, 2, 3}_{u_1}, \underbrace{4, 5, \ldots, 8}_{u_2}, \ldots, \underbrace{n-1, n}_{u_k})$
We have: $S = \bigwedge_{i=1}^{k} S_i$ with $S_i = \bigwedge_{j=\min(u_i)}^{\max(u_i)}(l_{j,1} \vee l_{j,2} \vee l_{j,3})$
Let $\mathcal{T}$ be a transformation as defined in Definition 2 in order to build the program P' in a polynomial time. The program P' is made up of the family of $(P'_i)_{i\in[0,k]}$ obtained from the program P. Figure 2 illustrates such a reduction.

Since as each path is assumed to be executable, we do not care about the logical conditions with respect to the branching instructions: `if (-) {code1} else {code2}`. Figure 3 presents the pseudo-code contained in block P'_0. This code is used to initialize the program P'. Every variable V_i and its negation $\overline{V_i}$ are first declared as pointers of function pointers. Each of these variables may point to `true` or `false` (declared in line 2 as function pointers). Every V_i and $\overline{V_i}$ describe a Boolean variable in S and its negated form respectively. Thus, setting v_i to "*true*" for S, corresponds to set `*`V_i`=true` for P'. An execution path in block P'_0 (lines 4 to 6) thus initializes every variable V_i and $\overline{V_i}$, which is equivalent to fix the values of Boolean variables v_i in the equation S. There are two cases: either the setting of the v_i satisfies S, or it does not.
Figure 4 defines for every i ranging from 1 to $k-1$, block P'_i from block P_i. The pointer `false` is initialized with the address of block P'_{i+1}. The second line represents the insertion of the code contained within block P_i in such a way that executing block P'_i results in executing block P_i as well. Notation $l_{i,j}$ in expressions $*l_{i,j}$=&P'_g (lines 3 and 4) is an abstract one. This symbol stands for the variable V_k or its negation $\overline{V_k}$ (as in the formulation of S). Lines 3 and 4 enable us to modify the destination of the pointer `false` according to the values of the

```
1 void (**$V_1$)(),(**$\overline{V_1}$)(),...,(**$V_m$)(),(**$\overline{V_m}$)();
2 void (*true)(),(*false)();
3 if (-) {$V_1$=&true;$\overline{V_1}$=&false;} else {$V_1$=&false;$\overline{V_1}$=&true;}
4 if (-) {$V_2$=&true;$\overline{V_2}$=&false;} else {$V_2$=&false;$\overline{V_2}$=&true;}
  ...
5 if (-) {$V_m$=&true;$\overline{V_m}$=&false;} else {$V_m$=&false;$\overline{V_m}$=&true;}
6 goto $P'_1$;
```

Fig. 3 Pseudo-code of block P'_0

variables V_i and $\overline{V_i}$ which have been initialized in P'_0 (see hereafter). Finally, the function false is called at line 5. Block P'_k slightly differs from the other blocks P'_i, $i \in [1, k-1]$), for program termination purposes. Throughout the rest of the paper, unless differently mentioned, we will always refer to Fig. 4. Moreover, let us denote the *output* of block P'_i as one of the two possible destinations: P'_{i+1} or P'_g. We will now prove the equivalence between our problem as defined in Proposition 2 and the 3-SAT problem.

(a) First case: S **is satisfiable**. This means that $\forall i \in [1, k]$, S_i is satisfiable. S_i satisfiable corresponds at the pseudo-code level to $l_{j,1}$, $l_{j,2}$ or $l_{j,3}$ points toward the true variable $\forall j \in [\min(u_i), \max(u_i)]$. Since at least one literal of the form $l_{j,t}$, $t \in [1, 3]$ is pointing towards the variable true, then there exists at least one path in the graph for which the pointer destination true is assigned to the P'_g address. With this property being valid for every lines from 3 to 4, consequently there exists a path starting from block P'_i for which the true variable has only been re-allocated to block P'_g address, in line 5 and thus for which the false variable has never been redefined. In the present case, the *output* of block P'_i is block P'_{i+1}. This property being valid for every block P'_i, subsequently we have $\mathcal{T} \in \mathcal{O}_k$. Consequently, **if** S **is satisfiable then** $\mathcal{T} \in \mathcal{O}_k$.

(b) Second case: $\mathcal{T} \in \mathcal{O}_k$. This implies that for every i, the *output* of block P'_i is block P'_{i+1}. In other words for every block P'_i, the false variable points towards block P'_{i+1} in line 5. Now the false variable points towards block P'_{i+1} only if, for the path with respect to block P'_i in question, every literal of the form $l_{j,t}$ point towards the variable true. Indeed, if for this particular path, a single literal would point towards false, then the address of block P'_g would be allocated to false in lines 3 or 4, and thus the latter would point towards a block different from block P'_{i+1} in line 5. This result for block P'_i implies that S_i is satisfiable. Since this predicate must be valid for every i (ranging from 1 to k), S is satisfiable, hence the result we were looking for: **if** $\mathcal{T} \in \mathcal{O}_k$ **then** S **is satisfiable.**

Finally, we proved the equivalence between our problem and the 3-SAT problem. □

Consequences in terms of detecting a specific category of metamorphic viruses

Definition 3 Let us consider a metamorphic virus V whose new instance V' is defined by $V' = \mathcal{T}(V)$ where $\mathcal{T} \in \mathcal{O}_k$. Under the assumption that all paths are potentially executable, we define a reliable detection as follows: A program D_V detects the metamorphic virus V if and only if, for any program P,

$$\begin{cases} D_V(P) \textit{ returns "true" if there exists } \mathcal{T} \in \mathcal{O}_k \\ \quad \textit{such that } P = \mathcal{T}(V) \\ D_V(P) \textit{ returns "false" otherwise.} \end{cases}$$

We can then deduce that for such a category of metamorphic viruses (Definition 3), a reliable static detection is an $\mathcal{NP}$-complete problem. This result can be viewed as a complement of the one of Spinellis [23] about the complexity of bounded-length polymorphic code detection. Indeed, we extend his result over metamorphic code. Even if this formal result seems far from detection in reality where false positives are acceptable, we believe that it could give birth to complex viruses able to thwart recent detection strategies like [4,20]. We will illustrate our metamorphic code obfuscator approach in Sect. 4.

Fig. 4 Pseudo-code of P' program blocks

```
1 false=&$P_{i+1}$;
2 .../* insertion of $P_i$ code*/
3 if (-) {*$l_{min(u_i),1}$=&$P'_g$;} else if (-) {*$l_{min(u_i),2}$=&$P'_g$;} else {*$l_{min(u_i),3}$=&$P'_g$;}
  ...
4 if (-) {*$l_{max(u_i),1}$=&$P'_g$;} else if (-) {*$l_{max(u_i),2}$=&$P'_g$;} else {*$l_{max(u_i),3}$=&$P'_g$;}
5 false();
6 return;
```

4 Metamorphic code obfuscator making static analysis more complex

This section presents a possible design approach for obfuscators that could be used by sophisticated metamorphic codes. This practical obfuscator rests on the proof of Proposition 2.

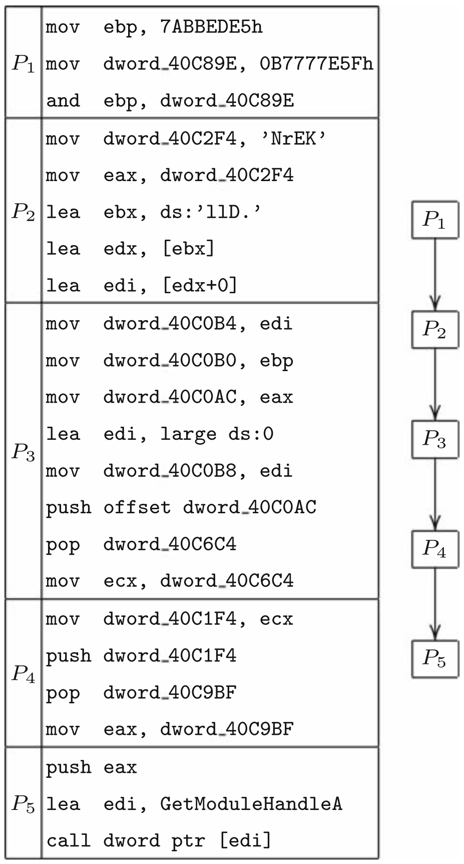
4.1 Obfuscation approach for the control flow

Our approach consists in modifying the control flow of a program P in order to produce an obfuscated program P'. It is close to the approach presented in [6] by means of finite state automata and that is presented in [19] which uses function pointers. However our own technique has two specific features:

1. Our proposed obfuscator works directly at the assembly source. This enables to apply all the obfuscation techniques presented in Sect. 2, independently of the control flow modifications. The transformations we use are kept during the code assembling process. It is worth mentioning that it is not always the case when considering higher level language source code. Indeed the compiling step generally performs some optimizations that can invert obfuscation transformations and thus lessen the obfuscation efficiency.
2. The obfuscation proposed hereafter is designed with severe constraints since it should enable the virus to model itself while making sure that the static detection of its own code is as complex as possible. This particular point will be addressed in the next subsection but is not exhaustively presented in this paper.

The global approach for building the obfuscated program P' can be summarized as follows:

1. As we did in the proof of Proposition 2, we split the program P—which contains a suite of n instructions $(I_j)_{j\in[1,n]}$—into k sets of random non null size, with each of them containing consecutive instructions. We thus get blocks $(P_i)_{i\in[1,k]}$. This program scissoring is performed in such a way that each block P_i does not end up with an unconditional jump (GOTO) or a routine exit (RET). This restriction is required otherwise the connection condition from block P_i to block P_{i+1} would be useless. Figure 5 presents an example of assembly code scissoring which corresponds to the first basic block in the WIN32.METAPHOR virus control flow graph. The detailed explanation of this code is not required for the understanding of the next part. The reader just needs to know that this basic block corresponds to the following call: `GetModuleHandle("Kernel32.dll")`. The block chaining here is linear $(P_1, P_2, \ldots, P_5)$.



```
P1  mov  ebp, 7ABBEDE5h
    mov  dword_40C89E, 0B7777E5Fh
    and  ebp, dword_40C89E
P2  mov  dword_40C2F4, 'NrEK'
    mov  eax, dword_40C2F4
    lea  ebx, ds:'llD.'
    lea  edx, [ebx]
    lea  edi, [edx+0]
P3  mov  dword_40C0B4, edi
    mov  dword_40C0B0, ebp
    mov  dword_40C0AC, eax
    lea  edi, large ds:0
    mov  dword_40C0B8, edi
    push offset dword_40C0AC
    pop  dword_40C6C4
    mov  ecx, dword_40C6C4
P4  mov  dword_40C1F4, ecx
    push dword_40C1F4
    pop  dword_40C9BF
    mov  eax, dword_40C9BF
P5  push eax
    lea  edi, GetModuleHandleA
    call dword ptr [edi]
```

$$\begin{tikzcd} P_1 \arrow[d] \\ P_2 \arrow[d] \\ P_3 \arrow[d] \\ P_4 \arrow[d] \\ P_5 \end{tikzcd}$$

Fig. 5 Example of program splitting into blocks (P_i) for a bootstrap input program of the WIN32.METAPHOR virus

2. From a practical point of view, we build a family of meaningful instruction sequences $(G_i)_{i\in[1,p]}$ in order to complicate the analysis. Each sequence has a non null random size. By meaningful, we mean that the instruction sequence comes from a real program and not from random instructions. This family describes dead code blocks. The purpose of these blocks is to increase the difficulty of the static detection of program P' only. In the following, we will use dead code blocks as illustrated in Fig. 6.
3. We now build the obfuscated program P' from blocks $(P'_i)_{i\in[0,k+p]}$ as follows:
   (a) $P'(0)$ is the starting block in the obfuscated program. It initializes a variable K which describes the unique switchpoint at the exit of each block P'_i, in such a way that $P' \equiv P$. This variable K represent a key element for the obfuscator. We call it the *obfuscation parameter*.
   (b) $\forall i \in [1, k+p]$, two choices have to be considered for building block P'_i:
   – either P'_i is a legitimate block, that is to say that $\exists! s \in [1, k]$ such that P'_i contains the code of P_s. In this case, we define a legitimate exit

G_1	`xor ebx, ebx`	G_2	`mov ebp, esp`
	`mov ebp, 24h`		`xor edi, edi`
	`mov eax, 26h`		`mov eax, 1F03FFh`

Fig. 6 Example of dead code blocks (G_i)

towards block P'_{i+1} and another random exit to the other blocks.

– or P'_i is an illegitimate block (dead code), that is to say that $\exists! t \in [1, p]$ such that P'_i contains the code of G_t. In this case, we define two random exits.

In any case, block exits are conditioned by K in such a way that the execution of program P is corresponding to that of P.

Initializing the obfuscation parameter K

Since K is a critical component for statically reconstructing the program P from P', we have to explain how to make the retrieval of this information more complex. We consider two different approaches:

1. The first one rests on mathematical complexity as presented in [11].
   – With respect to this approach, using some mathematical conjectures can dramatically increase the complexity of static analysis. For example, let us consider the Syracuse suite $(u_n)_{n\in\mathbb{N}}$ defined by:
   $$\begin{cases} u_0 \in \mathbb{N} \\ \forall n \in \mathbb{N}^*, u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{if } u_n \text{ is even,} \\ 3u_n + 1 & \text{otherwise.} \end{cases} \end{cases}$$
   The Syracuse conjecture claims that for every starting element u_0, the suite $(u_n)_{n\in\mathbb{N}}$ always ends up with the cycle 4, 2, 1. The obfuscation parameter K could then depend upon the smallest integer i which verify $u_i = 1$ ($K = f(i)$).
   – It is also possible to consider algebraic expressions which are complex to verify [19]. For example, the bit initialization k_i of K could be obtained by the following piece of code: `if (a*(a+1)%2==0) {`k_i`=1;} else {`k_i`=0;}` where `a` describes any integer. In this case, whatever the value of `a`, the previous predicate is always true. From a general point of view, resolving such a condition requires the use of complex algorithms as formal computing does.
2. Some difficulties induced by static analysis which cannot consider the program in an execution context. From a static point of view, the result of any external call (other programs, API…) remains unknown and cannot be solved unless running the program. The same is true for static analysis of results produced by iterative or recursive computing. For example, the use of hash functions or of encryption primitives can increase the complexity of the analysis. Let us suppose that $K = (x_m, x_{m-1}, \ldots, x_2, x_1)$ is defined by $\forall i \in [1, m], \exists j \in [1, m], x_i = \mathcal{H}(P_j)$ where $\mathcal{H}$ is any hash function outputting a single bit. Guessing a x_i needs to execute the function $\mathcal{H}$ which is a complex process in static analysis.

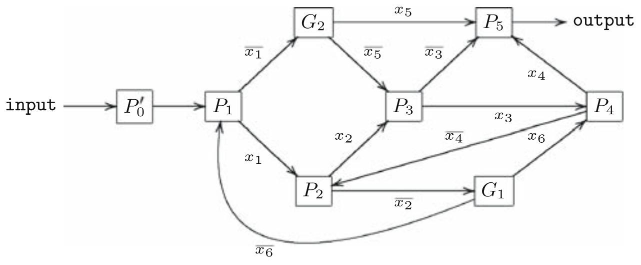


Fig. 7 Example of possible obfuscation for the program P

Table 3 Results of program P' execution for different values of K

K	Assignment of (x_i)	Execution paths	Program P' results
0	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, \overline{x_2}, \overline{x_1}$	P1,G2,P3,P5	Fatal error
1	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, \overline{x_2}, x_1$	(P1,P2,G1)*	Endless loop
2	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, x_2, \overline{x_1}$	P1,G2,P3,P5	Fatal error
3	$\overline{x_6}, \overline{x_5}, \overline{x_4}, \overline{x_3}, x_2, x_1$	P1,P2,P3,P5	Fatal error
4	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, \overline{x_2}, \overline{x_1}$	(P1,G2,P3,P4,P2,G1)*	Endless loop
5	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, \overline{x_2}, x_1$	(P1,P2,G1)*	Endless loop
6	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, x_2, \overline{x_1}$	P1,G2,(P3,P4,P2)*	Endless loop
7	$\overline{x_6}, \overline{x_5}, \overline{x_4}, x_3, x_2, x_1$	P1,(P2,P3,P4)*	Endless loop
8	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, \overline{x_2}, \overline{x_1}$	P1,G2,P3,P5	Fatal error
9	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, \overline{x_2}, x_1$	(P1,P2,G1)*	Endless loop
10	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, x_2, \overline{x_1}$	P1,G2,P3,P5	Fatal error
11	$\overline{x_6}, \overline{x_5}, x_4, \overline{x_3}, x_2, x_1$	P1,P2,P3,P5	Fatal error
12	$\overline{x_6}, \overline{x_5}, x_4, x_3, \overline{x_2}, \overline{x_1}$	P1,G2,P3,P4,P5	Incorrect
13	$\overline{x_6}, \overline{x_5}, x_4, x_3, \overline{x_2}, x_1$	(P1,P2,G1)*	Endless loop
14	$\overline{x_6}, \overline{x_5}, x_4, x_3, x_2, \overline{x_1}$	P1,G2,P3,P4,P5	Incorrect
15	$\overline{x_6}, \overline{x_5}, x_4, x_3, x_2, x_1$	**P1,P2,P3,P4,P5**	**Correct**

Example of static analysis complexity with respect to an obfuscated program

We will now try to illustrate the difficulty of deobfuscation through the example given in the following Fig. 7 which presents an obfuscated program P' obtained from an original program P.

Block P'_0 initializes the Boolean variables $(x_i)_{i\in[1,6]}$ which conditioned the exit of every block. For example, for block P_1, if $x_1 = 1$ then the next block to be executed is P_2 otherwise G_2. In order to make the execution of program P'

correspond to program P, we must keep the sequence of blocks as shown in Fig. 5. In this case, it corresponds to $\forall i \in [1, 4]$, $x_i = 1$. **Without knowledge of the value of K, the number of possible execution paths is equal to 2^{k+p-1}.**

Variables x_5 and x_6 conditioned the output of dead code blocks G_1 and G_2 only. These blocks are never executed whenever $P' \equiv P$. Table 3 thus presents the first 16 values of K only, by assuming that x_5 and x_6 are null. Instead of developing static analysis optimizations for the 16 possible $(x_i)_{i\in[1,4]}$ settings, we consider the execution result of the obfuscated program P' as shown in Table 3. This is relevant in this particular case since detecting this basic block (see Fig. 5) is equivalent to checking whether it comes to executing the `GetModuleHandle("Kernel32.dll")` call as previously noted. Table 3 presents four types of results which are summarized as follows:

- seven cases ($K = 1, 4, 5, 6, 7, 9, 13$) which correspond to an endless loop for which the `call` instruction is never reached;
- six cases ($K = 0, 2, 3, 8, 10, 11$) in which an error occurs during the WIN32 `GetModuleHandle` call. This error comes from an incorrect parameter (invalid pointer);
- two cases ($K = 12, 14$) exit 0 after the call. Here, the pointer given as parameter is valid indeed but it does not correspond to the awaited string (`"Kernel32.dll"`);
- a single case ($K = 15$) which corresponds to the awaited call according to the value of K which has been used for building P'.

This example clearly shows the concrete impact of the theoretical result on a static analysis whenever the obfuscation parameter K is unknown.

5 Conclusion

In this paper, we have proved that a reliable static analysis of a particular category of metamorphic viruses is an $\mathcal{NP}$-complete problem. As a practical application, we have presented an obfuscating approach whose resilience should be high enough to defeat static analysis tools. This approach operates by modifying the program control flow and has been designed to be applied to metamorphic viruses in order to prevent an efficient static analysis. Indeed, it rests on the fact that a metamorphic virus can, during its execution, solve problems which are considered as complex in static analysis. We emphasize that our result is close to perfect detection. However, we believe that this approach could present a problem for the state of the art detection tools. Fortunately, the obfuscation technique presented here cannot be applied to existing viruses: the metamorphic replication process operates by obfuscating the code after a modeling step. To use our approach directly, the virus should first be able to invert this transformation during its own execution in order to model itself. This modeling step, as well as the modifications it implies are not addressed in this paper. Our future work will consider this aspect in order to study the whole replication process: virus modeling then code obfuscation step.

Acknowledgments The authors would like to thank Guillaume Bonfante (LORIA) for his valuable comments on the first version of this paper.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and, Tools. Addison-Wesley, Reading, MA (1986)
2. Beaucamps, P., Filiol, E.: On the possibility of practically obfuscating programs—towards a unifed perspective of code protection. J. Comput. Virol. (WTCV'06 Special Issue, Bonfante, G., Marion, J.-Y. eds), **2**(4) (2006)
3. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: Crypto '01, LNCS No.2139, pp. 1–18 (2001)
4. Bruschi, D., Martignoni, L., Monga, M. : Detecting self-mutating malware using control-flow graph matching. In: Büschkes, R., Laskov, P. (eds.) Detection of Intrusions and Malware & Vulnerability Assessment, volume 4064 of LNCS, pp. 129–143. Springer, Berlin (2006)
5. Bruschi, D., Martignoni, L., Monga, M.: Using code normalization for fighting self-mutating malware. In: Proceedings of the International Symposium of Secure Software Engineering. IEEE Computer Society, Arlington (2006)
6. Chow, S., Gu, Y., Johnson, H., Zakharov, V.A.: An approach to the obfuscation of control-flow of sequential computer programs. In: ISC '01: Proceedings of the 4th International Conference on Information Security, pp. 144–155. Springer, London (2001)
7. Cifuentes, C.: Reverse Compilation Techniques. Ph.D. thesis, Queensland University of Technology, School of Computing Science (1994)
8. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: In Proceedings of the 12th USENIX Security Symposium, pp. 169–186 (2003)
9. Christodorescu, M., Jha, S.: Testing malware detectors. In ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, pp. 34–44. ACM Press, New York (2004)
10. Cohen, F.: Computational aspects of computer viruses. Rogue programs: viruses, worms and Trojan horses, pp. 324–355 (1990)
11. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Univerity of Auckland, New Zealand (1997)
12. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Principles of Programming Languages 1998, POPL'98, pp. 184–196 (1998)
13. Debray, S.K., Evans, W., Muth, R., De Sutter, B.: Compiler techniques for code compaction. ACM Trans. Program. Languages Syst. **22**(2), 378–415 (2000)
14. Filiol, E.: Advanced Viral Techniques: Mathematical and Algorithmic Aspects. Springer, Berlin (2006)
15. Filiol, E.: Malware pattern scanning schemes secure against blackbox analysis. J. Comput. Virol. (EICAR 2006 Special Issue, Broucek, V., Tuner, P. eds), **2**(1) (2006)

16. Jacob, G., Filiol, E., Le Liard, M.: Evaluation methodology and theoretical model for antiviral behavioural detection strategies. J. Comput. Virol. (TCV 2006 Special Issue, Bonfante, G., Marion, J.-Y. eds), **3**(1) (2007)
17. Landi, W.A.: Interprocedural Aliasing in the Presence of Pointers. Ph.D. thesis, New Brunswick, New Jersey, USA (1992)
18. Lakhotia, A., Kapoor, A., Kumar, E.U.: Are metamorphic viruses really invincible? Virus Bull. (2004)
19. Ogiso, T., Sakabe, Y., Soshi, M., Miyaji, A.: Softwate tamper resistance based on the difficulty of interprocdeural analysis. In: The Third International Workshop on Information Security Applications, pp. 437–452 (2002)
20. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07), pp. 377–388, January 17–19, 2007. ACM Press, New York (2007)
21. Rice, H.G.: Classes of recurively enumerable sets and their decision problems. Transactions of the American Mathematical Society, pp. 358–366 (1953)
22. Szor, P., Ferrie, P.: Hunting for metamorphic. In: Virus Bulletin Conférence, September 2001
23. Spinellis, D.: Reliable identification of bounded-length viruses is np-complete. IEEE Trans. Inform. Theory **49**(1), 280–284 (2003)
24. Sung, A.H., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer of vicious executables(save). In: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04). IEEE (2004)
25. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley, Reading, MA (2005)
26. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia (2000)
27. Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Normalizing metamorphic malware using term rewriting. SCAM 2006: The 6th IEEE Workshop Source Code Analysis and Manipulation, pp. 75–84 (2006)