World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:1, No:2, 2007

Metamorphism, Formal Grammars and Undecidable Code Mutation

Éric Filiol

Abstract**—** This paper presents a formalisation of the different existing code mutation techniques (polymorphism and metamorphism) by means of formal grammars. While very few theoretical results are known about the detection complexity of viral mutation techniques, we exhaustively address this critical issue by considering the Chomsky classification of formal grammars. This enables us to determine which family of code mutation techniques are likely to be detected or on the contrary are bound to remain undetected. As an illustration we then present, on a formal basis, a proof-of-concept metamorphic mutation engine denoted PB_MOT, whose detection has been proven to be undecidable.

Keywords**—** Polymorphism, Metamorphism, Formal Grammars, Formal Languages, Language Decision, Code Mutation, Word Problem.

I. Introduction

Polymorphism and metamorphism are the two techniques dedicated to hinder sequence-based antiviral detection. The principle is to cancel as much as possible any potential fixed element in a malware code that would represent a potential detection pattern. Polymorphism has formerly been introduced by Fred Cohen [5] with the concept of *Largest Viral Set* while metamorphism has appeared in the early 2000s as a response to the polymorphism's inherent limitations.

From a theoretical point of view [20], [21], the core of a polymorphic malware is its kernel which is made up of an infection trigger condition[1] $I(d,p)$, a payload routine $D(d,p)$, the corresponding payload trigger condition $T(d,p)$ and a selection function $S(p)$ (of target programs to infect). It is precisely the latter function which is in charge of the code mutation.

Metamorphic viruses differ from polymorphic viruses since their respective selection function are different. While all polymorphic forms of a virus share the same kernel, the metamorphic forms of a virus have a completely different kernel. Consequently, if detection remains tractable for some classes of polymorphism – the kernel does not change during the mutation process and thus can be used as a detection pattern –, it becomes far different as far as metamorphism is concerned.

There are very few theoretical results concerning the detection complexity of code mutation techniques. This problem has been addressed very recently only. D. Spinellis has proved [17] that detection of bounded-length polymorphic viruses is an NP-complete problem. Zuo and Zhou [20] have then proved that the set D_i of polymorphic viruses with an infinite number of forms is a Σ_3-complete set. Unfortunately, no results is known for other classes of polymorphic viruses and for the general case of metamorphism. Many open problems still remain.

Up to now, only very few examples of metamorphic codes are known to exist. The most sophisticated one is the *MetaPHOR* engine whose essential feature is a certain amount of non-determinism. Experiments in our laboratory showed that existing antivirus software can be very easily defeated by *MetaPHOR*-like technology. However, the analysis of this engine [9, Chap. 4] has proved that its metamorphic techniques still belong to trivial classes.

Our research thus focused on the formalisation of metamorphism by means of formal grammar and languages. We aimed at identifying the different possible classes of possible code mutation techniques. The first results, which are presented in this paper, enable to assert that detection complexity of code mutation techniques can be far higher that NP-complete and that for some well-chosen classes, detection is an undecidable problem.

The links between polymorphism and formal grammars has been introduced in [16] for the first time. Unfortunately, the author did touch on this issue only. Metamorphism is not addressed at all. Some aspects are dealt with in a very naive way and we will prove in this paper that some of its conclusions are totally wrong.

This paper is organised as follows. Section II presents the main theoretical tools of computability theory we use throughout this paper. Section III then explains how code mutation techniques can be modelled by formal grammars and how their detection can be reduced to the problem of deciding a language. Section IV will then presents our proof-of-concept metamorphic engine, denoted POC_PBMOT we have designed in order to validate our theoretical model. In particular, we will show that detecting this engine is an undecidable problem. Section V finally concludes and present some future work.

II. Formal Grammars and Rewriting Systems

Let us first recall basic notation and concepts we will use throughout this paper. We consider a finite set $\Sigma = \{a_1, a_2, \ldots, a_n\}$ as an alphabet whose elements are called *symbols*. A sequence of symbols of Σ is called a chain $b_1b_2b_3\ldots b_m$ with $b_i \in \Sigma$ and $m \geq 0$. The concatenation of two chains x and y is the chain $xy = b_1b_2\ldots b_mc_1c_2\ldots c_n$. Let A and B two sets of chains defined over Σ. Then we can

Also Associate Senior Professor at ESIEA - Laval *filiol@esiea.fr*
E. Filiol is with the Lab. of Virology and Cryptology, ESAT, Rennes (France)
Email: eric.filiol@esat.terre.defense.gouv.fr

[1]The running environment (d,p) is made up of *data* (d) and *programs* (p).

define the following sets:

$$\begin{aligned} AB &= \{xy | x \in A, y \in B\}, \\ A^* &= \{x_1 x_2 \dots x_n | n \geq 0, x_1, x_2, \dots, x_n \in A\}, \\ A^+ &= \{x_1 x_2 \dots x_n | n \geq 1, x_1, x_2, \dots, x_n \in A\}. \end{aligned}$$

This notation enables us to introduce the concept of *formal grammar*.

Definition 1: [14] A *formal grammar* G is the 4-tuple $G = (N, T, S, R)$ where:

- N is a set of non-terminal symbols;
- T is an alphabet of terminal symbols with $N \cap T = \emptyset$;
- $S \in N$ is the start symbol;
- R is a rewriting system, that is to say a finite set of rules $R \subseteq (T \cup N)^* \times (T \cup N)^*$, such that $(u, v) \in R \Rightarrow u \notin T^*$ (we cannot rewrite chains which contain only terminal symbols).

The rewriting system (still known as *semi-Thue system*) over Σ is in fact a finite subset of $\Sigma^* \times \Sigma^*$. In other words, it is the set $R = \{(u_1, v_1), \dots, (u_n, v_n)\}$. A pair $(u, v) \in R$ is a rewriting rule or *production*. It is denoted $u ::= v$ for short (instead of $(u, v) \in R$).

A rewriting system R enables to define a rewriting relation, denoted $\Rightarrow_R$ which is defined as:

$rus \Rightarrow rvs$ if and only if $(u, v) \in R$ and $(r, s) \in \Sigma^* \times \Sigma^*$.

This means that we can build the chain $rvs \in \Sigma^*$ directly (e.g. in one step) from the chain $rus \in \Sigma^*$.

Example 1: [12] Let us consider $\Sigma = \{A, a, b, c\}$ and

$$R = \{(A, aAa), (A, bAb), (A, c), (A, aca)\}.$$

$$\begin{aligned} A &\Rightarrow_R aAa \\ aAa &\Rightarrow_R aaAaa \\ aaAaa &\Rightarrow_R aacaa \end{aligned}$$

This relation allows to define a reflexive and transitive closure for the relation $\Rightarrow$. We will denote it $\Rightarrow_R^*$. This relation is defined, for every r, g, h in Σ^* by:

1) if $g \Rightarrow_R h$ then $g \Rightarrow_R^* h$
2) $g \Rightarrow_R^* g$
3) if $g \Rightarrow_R^* r$ and $r \Rightarrow_R^* h$ then $g \Rightarrow_R^* h$.

Equivalently, two words are related with respect to this relation, if and only if one can be produced from the another. As an example, in the previous example, we can replace the symbol $\Rightarrow_R$ by $\Rightarrow_R^*$.

A Thue system is a semi-Thue system in which the relation $\Rightarrow_R^*$ is symmetric. It is consequently denoted $\Leftrightarrow_R^*$. Let us consider the following example.

Example 2: Let us consider the grammar $G = (N, T, S, R)$ with $T = \Sigma = \{0, 1\} \cup \{\epsilon\}$, $N = \{S, X, Y\}$ and R defined by:

$$\begin{aligned} S &::= 1S | 0S | X \\ X &::= 0Y \\ Y &::= 1Y | 0Y | Z \\ Z &::= \epsilon \end{aligned}$$

This grammar builds every chain containing at least one zero.

A *formal language* is finally defined by the set $L(G) = \{x \in \Sigma^* | S \Rightarrow^* x\}$ with respect to the grammar $G = (N, T, S, R)$. It is nothing more than the "words" (or chains) generated with respect to this grammar. From this point of view, natural languages and programming languages are just instances of a wider concept.

A huge classification work of formal grammars has been done by Noam Chomsky [3], [4]. This author has identified four different classes.

- Class 0 grammars (or *free grammars*). They are all grammars whose productions may be freely written as $x ::= y$ where y is an arbitrary chain of symbols taken in $N \cup T$. These grammars all generate languages that can be decided by Turing machines. Equivalently, the languages they generate are recursively enumerable (see [6, Chap. 2]);
- Class 1 grammars (or *context-sensitive grammars*). The unique constraint on the production lies in the fact that the size of words cannot decrease. Consequently, the only possible productions are in the form of $x ::= y$ as long as $|y| \geq |x|$. This class contains all natural languages.
- Class 2 grammars *context-free grammars*. They are all grammars whose productions are in the form of $X ::= y$ where X is a unique nonterminal symbol and where y is an element of $(N \cup T)^*$. The term X can be rewritten independently from its context controry to class 1 grammars. Subsets of class 2 grammars are commonly used to implement programming languages.
- Class 3 grammars (or *regular grammars*). They are all grammars whose productions are in the form of $X ::= x$ or $X ::= xY$ with $(X, Y) \in N^2$ and $x \in T^*$.

III. Polymorphism, Metamorphism and Formal Grammars

In this section, we will first formalise code mutation in term of rewriting techniques. We will then be able to exhaustively address the complexity of code mutation detection according to the class of the grammar in use.

A. Code mutation and Formal Languages

Let us consider the set of x86 instructions as our working alphabet. These instructions may be combined according to (rewriting) rules that completely define every compiler. This set of rules can be defined as a class 2 formal grammar indeed. The assembly language is then the language which is generated by this grammar.

Implementing a polymorphic engine – in particular the garbage generator which is its most important part – consist in generating a formal language, denoted *polymorphic language*, with its own grammar.

Let us consider a polymorphic engine generated by the grammar (the example is derived from [16]).

$$G = \{\{A, B\}, \{a, b, c, d, x, y\}, S, R\}.$$

Instructions a, b, c and d represent garbage code while instructions x and y are the decryptor's instructions (e.g. $x =$ `XOR`

`[EDI]`, `AL` and $y =$ `INC EDI`). The rewriting system R can be defined as:

$$\begin{aligned} S &::= aS|bS|cS|xA \\ A &::= aA|bA|cA|dA|yB \\ B &::= aB|bB|cB|dB|\epsilon \end{aligned}$$

Consequently, our polymorphic language is made up of every word in the form of

$$\{a,b,c,d\}^*x\{a,b,c,d\}^*y\{a,b,c,d\}^*.$$

Every of these words corresponds to a mutated variant of the initial decryptor. It is thus "easy" (e.g for an antivirus software) to determine that the word `abcddxd` is not in this language with respect to G, contrary to the word `adcbxaddbydab`. Other examples of less trivial polymorphic grammars are presented in [9, Chap. 6].

All this being defined, the essential issue for any antivirus is to have an algorithm which is able to determine whether a "word" (a mutated form) belongs to a polymorphic language or not. But as soon as we consider sophisticated polymorphic techniques, this ability is very difficult to evaluate. That is precisely the interest of modelling code mutation with formal grammars. According to the grammar class, we can get a more accurate insight of the detection complexity.

B. Antiviral Detection and Language Decision

In order to set up our model, let us consider the following definition.

Definition 2: [12] Let $G = (N, T, S, R)$ be a grammar and $x \in T^*$ a chain with respect to G. The *language decision problem* with respect to G consists in determining whether $x \in L(G)$ or not. The *language completeness problem* is that which aims at deciding whether $L(G) = T^*$ or not.

The first problem models the detection problem of polymorphism (once the relevant grammar is known). The second one models the concepts of non detection and false positive.

In order to address the detection problem, let us just recall the existing algorithmics tools we have at our disposal's. They will thus enable us to give complexity results with the different instances of this problem. A detailed description of these tools can be found in any computability handbook (e.g [12]). The generic tool is a finite automaton. Two different kind of automata are to be considered.

- *Deterministic Finite Automata* (DFA). It is the most simple form of automaton. It may be represented by a directed graph, whose nodes are the possible states of the automaton and the arcs joining nodes are labelled by symbols of the alphabet. The symbols in fact deterministically define the transition condition from one state to another state.
- *Non deterministic Finite Automaton* (NFA). It is a generalised automaton. The essential difference with a DFA comes from the fact that more than one arc with the same label can start from a node. This means that from one given state, a condition can result in different effects. As a result we got a far higher number of evolving capabilities but also a far higher computing complexity. Moreover, NFAs allow transitions on the empty strings.

We can now formalise the action of any antivirus with respect to code mutation detection. Without loss of generality, we will consider NFAs only (in fact it is possible to reduce a NFA to a DFA, up to an exponential increase of the number of states [14]).

Definition 3: [12] We say that a chain $x = x_1x_2 \dots x_n$ with $x_i \in \Sigma$ is dedided by an automaton[2] $A = (Q, \Sigma, \tau, q_0, F)$ if there exist a state sequence $q_1, q_2, \dots, q_{n+1}$ of Q and a symbol sequence $x_1, x_2, \dots, x_n$ of $\Sigma \cup \{\epsilon\}$ such that $q_{i+1} \in \tau(q_i, x_i)$ for every $i \in \{1, 2, \dots, n\}$ with $q_0 = q_1$. Then we note $L(A)$ the set of any chain detected by A. It is the language decided by A. In other words, A decides whether $L(A) = L(G)$ or not. Consequently, the automaton A is a solution of the language decision problem with respect to the grammar G.

This definition describes the fact that as soon as an antivirus software embeds an automaton A which is able to solve the (polymorphic) decision problem with respect to a given polymorphic grammar, then it is able to detect any of its "polymorphic words" (e.g mutated forms). Two critical issues are then to be considered:

- the relevant complexity of the automaton,
- every time the polymorphic grammar is changing (e.g. the polymorphic engine is a new one), the antivirus software mut be upgraded with a new automaton which decides the new polymorphic language.

The last points underlines the essential interest of metamorphic techniques compared to polymorphic ones. That is the reason why antivirus software are bound to be defeated by metamorphism. Indeed every new metamorphic mutation aims at producing a new grammar and a new word generated by the latter at the same time. Consequently, we define metamorphism as a grammar whose words are themselves a set of productions with respect to a grammar. This may be sound as a naive assumption about antivirus software. Unfortunately, it is not. It has been proved in [7], [8] that these software still heavily relies on first- or second generation scanning techniques contrary to what is claimed by the antivirus vendors.

Let us now consider our formal definition of metamorphism.

Definition 4: (Metamorphic Virus) Let $G_1 = (N, T, S, R)$ and $G_2 = (N', T', S', R')$ be grammars where T' is a set of formal grammars, S' is the (starting) grammar G_1 and R' a rewriting system with respect $(N' \cup T')^*$. A metamorphic virus is thus described by G_2 and every of its mutated form is a word in $L(L(G_2))$.

The notation $L(L(G_2))$ which is more practical, stands for $L(x)$ for some $x \in L(G_2)$ where x is a grammar. This definition describes the fact that from one metamorphic form to another, the virus kernel is changing: the virus is mutating and change the mutation rules at the same time. Section IV will present a proof-of-concept of this formalisation. Definition 4 in fact somehow relates to *two-level grammars* (or Van

[2]The sets Q, Σ and F are the set of the automaton's possible states, a finite alphabet and a subset of states that can accepted by the automatom respectively. The state q_0 denotes the initial state whereas τ is the transition function $\tau : Q \times \Sigma \to Q$ which maps a state and a symbol to a state.

Wijngaarden grammars; 2VW grammars for short) [19], [10] in which grammar G_2 can be compared to some extent to the 2VW metagrammar. It is an open problem, at the moment, to determine whether our construction of Definition 4 is a particular case of 2VW grammars or not.

As a first consequence, managing metamorphism implies to have suitable automata at our disposal's in order to solve the language decision problem with respect to G_2-like grammars.

We now can give complexity results for this problem, according to the existing grammar classes:

Proposition 1: The language decision problem:

- is undecidable for class 0 grammars;
- has NP-complexity for class 1 and class2 grammars;
- has polynomial complexity for class 3 grammars.

Proof: This proof has been established by summarizing results given in [11], [14]. As far as class 0 grammars are concerned, we show that they generate recursively enumerable languages (their productions simulate Turing machines). Consequently, deciding whether $x \in L(G)$ or not for given G and $x \in \Sigma$ reduces the Halting problem.

For class 2, the language decision problem can be solved with non deterministic pushdown automata while class 1 grammars, it is solved with linear-bounded non deterministic Turing machines. Lastly, class 3 grammars are solved by deterministic ones. Hence the results. ∎

Proposition 1 stresses on the fact that the choice of underlying grammar is essential when designing a polymorphic engine. It has a direct impact on its resistance against its potential detection. Quite all known polymorphic engines refer to class 3 grammars. That is the reason why they can be successfully detected. But contrary to claims in [16] about systematic detection capabilities, untractability (classes 2 and 1) or even worse impossibility (class 0) rule out the practical detection when considering the cases of code mutation engines with respect to higher classes of grammars. From a practical point of view, no antivirus can monopolize computer ressources for minutes or even hours to solve some rather computable instances of a NP problem. Nowadays, our saving grace is that malware writers still seem to neglect or ignore what are the "good" grammars (or the "worst" ones from the defense point of view). This statement of course only holds for already detected or detectable malware.

There exist a huge number of theoretical results in the field of formal languages [2] that can be used to build code mutation techniques that are untractable or even impossible to detect. From a general point of view, the approach consists in considering the undecidability status for some known problem. In this respect, we may consider the Rice's Theorem. Let us consider a trivial property P about a language; in other words, there exists at least one recursively enumerable language (class 0) L for which P holds and at least one recursively enumerable language L' for which P does not.

Theorem 1: (Rice's Theorem [12]) For any non trivial property P with respect of languages, the problem of determining whether P holds for a language $L(M)$ of a Turing machine M, in undecidable.

This theorem is essential since it clearly indicates in which context to look at in order to systematically defeat antivirus. We will consider this in Section IV.

To end with formal grammars, it is worth giving the following results, whose proof will be found in [12, §10.4].

Theorem 2: Let $G_i = (N_i, T_i, S_i, R_i)$ with $i = 1, 2$ two context-free grammars. Deciding whether $L(G_1) \cap L(G_2) = \emptyset$ or not is an undecidable problem. Determining whether $L(G_i) = T_i^*$ or not is undecidable as well.

These two results, in the special context of context-free grammars illustrates the concepts of false positive (grammar G_1 is not a viral one while grammar indeed is a viral one) and of non detection, as far as antiviral detection is concerned [9, Chap. 2 and 4].

IV. Undecidable Mutation Techniques

A. The Word Problem

The *word problem* has been introduced and formalised by E. Post in 1950 [15]. Aside the Turing's Halting problem, it is one of the most famous problem which is known to be undecidable. This problem consists in deciding whether two finite words r and s over an alphabet Σ are equivalent or not, up to a rewriting system R. In other words, it consists in deciding whether $r \Rightarrow_R^* s$ or not.

Theorem 3: [15] The word problem with respect to a semi-Thue system is undecidable.

The proof consists in reducing the word problem to the Halting problem which is itself undecidable.

Example 3: (Tzeitzin semi-Thue Systems) Let the rewriting system R defined over the alphabet $\Sigma = \{a, b, c, d, e\}$.

$$\begin{aligned} (ac, ca), &\quad \text{commutation} \\ (ad, da), & \\ (bc, cb), & \\ (bd, db), & \\ (eca, ce), & \\ (edb, de), & \\ (cca, ccae) &\quad \text{deletion/insertion} \end{aligned}$$

This semi-Thue system is called *Tzeitsin system* [18]. It is the smallest semi-Thue system which is known to be undecidable. We will denote it T_0. As a consequence, any semi-Thue system which contains T_0 is itself undecidable. There exist many other undecidable Thue systems. We will use in the rest of the paper the Tzeitsin system T_1 defined by:

$$\begin{aligned} (ac, ca), \\ (ad, da), \\ (bc, cb), \\ (bd, db), \\ (eca, ce), \\ (edb, de), \\ (cdca, cdcae), \\ (caaa, aaa), \\ (daaa, aaa) \end{aligned}$$

B. Code Mutation Based on The Word Problem: the PBMOT *engine*

The central principle is to use formal grammars whose rewriting system is a Thue system which itself contains a Tzeitsin system or any other system which is known to be undecidable. From a general point of view, this implies that the code mutation engine based on such grammars will be undecidable as well. The core approach to design such an engine is to practically implement the concept of Definition 4. In other words, the engine's rewriting rules (namely the mutation rules) will change from mutation to mutation. For that purpose, two main constraints are to be satisfied:

- the rewriting system of G_2 contains an undecidable Thue system;
- every word (hence a grammar) in $L_i(G_2)$, during the i-th mutation step, contains an undecidable Thue system as well.

From an implementation point of view, the central approach consists in coding the rewriting system of $L_i(G_2)$ grammars as words on the alphabet $(N \cup T)^*$ where the sets T and N are those of grammars G_1^i. In other words the set R of rules (with respect to grammar G_1^i) :

$$R = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}$$

is coded as the following word:

$$u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n, \tag{1}$$

made of terminal and non terminal symbols.

The other essential point is to design a grammar G_2 which is able to manipulate such "grammar-words". The set T' contains words build on $(N \cup T)^*$ (Equation 1). As for the set N', it contains symbols which are specific to the grammar G_2 but it can also contains symbols in N. The starting element is a word in $(N' \cup T)^*$. We just have to deal with the rewriting system R' on words of $(N \cup T)^*$ with the constraint that $R' \supset T_0$ ou $R' \supset T_1$.

If the general principle ruling the design of grammar G_2 is simple to grasp, on the other hand its practical construction is technically far more complex. We will not present here due to lack of space – it would require tens of pages – but also not to give ready-to-use techniques that could be misused. We will give the two core principles of this practical construction only:

- the final code must be envisaged in functional terms and not in terms of code (assembly instructions). This point is critical since it is not the form of the different instructions but their interactions which is the most important aspect. If the rewriting rules are not trivial ones, the code mutation, in terms of opcodes, will work in a straightforward and "natural" way. From a technical point of view, this means that the rewriting rules have to deeply modify in words $u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n$, both the order of the $u_i v_i$ and the pairs (u_i, v_i) themselves at the same level;
- the whole code must be organised in terms of procedures (or blocks of codes) even if the coding itself is not structured in this way.

The deep nature of the chosen rules as well as their more or less sophisticated level will have a direct impact on the detectability of the engine which embeds them.

As far as our POC_PBMOT is concerned, the *MetaPHOR* engine code has been considered as a starting point. The design steps are hereafter presented.

1) the different modules of the *MetaPHOR* engine have been analysed in depth[9, Chap. 4]. The set T has then been built accordingly: it corresponds, up to some minor differences, to the different possible instructions;
2) in a second step, our analysis aimed at indentifying the main functionalities involved in the code mutation. A proto-rewriting (embryonic) system R'_0 has been first designed, in such way that it includes the T_0 system. The system R'_0 is a framework whose essential role is to perform code mutation itself. In other words, the rewriting takes the pairs (u_i, v_i) in words $u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n$ into consideration. At this early stage, the set N is not defined yet;
3) the analysis was then concerned with modifying the functions involved in the code transformation at a meta-level (the central point in metamorphism). This part has enabled to choose the set N of non terminal symbols in a first step. These symbols are essential since they are directly implied in the structure of the words in the form of $u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n$ at a macro-level. In a second step, the proto-system R'_0 has been modified and tuned up in order to achieve the final rewriting system R' which includes the T_1 system.

The critical point lies in the mutation of a word in the form of

$$u_1 v_1 u_2 v_2 \ldots u_{n-1} v_{n-1} u_n v_n$$

into a system

$$R = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}.$$

Indeed, the successive rewriting steps may induce variations of size in subwords u_i and v_i. It is thus necessary to record and update all these variations. In the same or equivalent way as the *MetaPHOR* engine does, the rewriting management must take the conditional or not jumps into account.

We now can state the following result.

Proposition 2: The detection of POC_PBMOT-based metamorphic codes is an undecidable problem.

Sketch of Proof.
Every mutated form ν_i is a word in the form of

$$L(L_i(G_2)) = L(u_1^i v_1^i u_2^i v_2^i \ldots u_{n-1}^i v_{n-1}^i u_n^i v_n^i).$$

Detecting such a code consists in deciding whether two words

$$\nu_i = u_1^i v_1^i u_2^i v_2^i \ldots u_{n-1}^i v_{n-1}^i u_n^i v_n^i$$

and

$$\nu_j = u_1^j v_1^j u_2^j v_2^j \ldots u_{n-1}^j v_{n-1}^j u_n^j v_n^j,$$

with $j > i$, are such that $\nu_i \Leftrightarrow^*_{G_2} \nu_j$. Grammar G_2 contains T_0 and T_1 systems, which are undecidable systems. Hence the result. □

Remark. Proposition 2 refers to sequence based detection, in other words not in an execution context. In particular, it implies that potential successful detection is bound to consider another approach like behaviour monitoring. In this latter case, it not sure that antivirus software would be more efficient at detecting a PB_MOT-like metamorphic engine [8].

C. Discussion

In [13], it has been suggested that metamorphic viruses are ultimately constrained in complexity. To quote the authors "*...a metamorphic must be able to disassemble and reverse itself. Thus a metamorphic virus cannot utilize [...] techniques that make it harder or impossible for its code to be disassembled or reverse engineered by itself.*" So what about the detection of PB_MOT metamorphic codes? Two different aspects are to be considered:

- any sequence-based detection will fail since mutation is based on indecidable problems as stated by Proposition 2;
- in an execution context, indeed the code has to disassembly itself into a code that the processor is able to run in order the virus operates. That means that once the code is unprotected, it may be analysed. From a theoretical point of view, this is true but antivirus and virus are not bound to play the same game. As shown in [1], [9] with τ-obfuscation, a metamorphic malware can delay its own disassembly more than the antivirus can accept. While indeed ultimately constrained in complexity, any metamorphic malware can reverse engineer itself in an arbitrary time τ contrary to any antivirus, which is a commercial product before anything else.

V. Future Work and Conclusion

Mutation code techniques are very efficient at practically hindering or even forbidding antiviral detection. But those techniques must be efficient enough. The theoretical approach with formal grammars is a new, promising way to systematically distinguish efficient techniques from non trivial or unefficient ones. Until now, known (theoretically detected) metamorphic codes refer to rather naive or trivial instances for which detection remains "easy".

Existing mutation code techniques, by definition, aim at preventing sequence-based detection, in a broader meaning of the term. On the other hand, some behaviours may represent useful invariant that could be considered by antivirus in the future. Consequently, the next step is likely to be behavioural polymorphism/metamorphism: code behaviours both at the micro- and the macro level would change from replication to replication. Current work in our laboratory already shows that this approach is not only powerful but also very worrying in terms of antiviral detection and protection.

It is nothing but very likely that if technical solutions to detect metamorphic codes exist, they would be non suitable for commercially-viable antivirus software. This is essentially due to the intrinsic algorithmic complexity of the detection algorithms. On the other hand, grammar-based formalisation should help antivirus programmers to identify and choose more powerful detectors to better manage existing code mutation techniques. Second generation scanners do not explore all the might of deterministic and non deterministic automata. As a consequence, existing antivirus can still be defeated by classical code mutation techniques.

References

[1] Beaucamps P., Filiol E. (2006), On the possibility of practically obfuscating programs - Towards a unified perspective of code protection, *Journal in Computer Virology*, (2)-4, WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds.

[2] Carton O. (2006), *Langages formels, calculabilité et complexité*, Cours de l'École Normale Supérieure. Available on `http://www.jussieu.fr/~carton/Enseignement/Complexite/ENS/Support/`

[3] Chomsky N. (1956), Three models for the description of languages, *IRE Transactions on Information Theory*, 2, pp. 113–124.

[4] Chomsky N. (1969), On certain formal properties of grammars, *Information and Control*, 2, pp. 137–167.

[5] Cohen F. (1986), *Computer viruses*, Ph. D thesis, University of Southern California, January 1986.

[6] Filiol E. (2005), *Computer viruses : from theory to applications*, IRIS International Series, Springer Verlag France, ISBN 2-287-23939-1.

[7] Filiol E. (2006), Malware Pattern Scanning Schemes Secure Against Black-box Analysis. In: *Proceedings of the 15th EICAR Conference*. The extended version has been published in *Journal in Computer Virology*, EICAR 2006 Special Issue, Vol. 2, Nr. 1, pp. 35–50.

[8] Filiol E., Jacob G. et Le Liard M. (2006), Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies, *Journal in Computer Virology*, (3)-1, WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds.

[9] Filiol E. (2007), *Techniques virales avancées*, Collection IRIS, Springer Verlag France. An English translation is pending and will be in print in mid 2007.

[10] Grune D. (1984), How to Produce All Sentences From a Two-level Grammar, *Information Processing Letters*, 19, pp. 181-185.

[11] Hopcroft J. E., Motwani R. and Ullman J. D. (2006), *Introduction to Automata Theory, Languages and Computation*, 3rd ed., Addison Wesley.

[12] Jones N. D. (1997), *Computability and Complexity*, MIT Press.

[13] Lakhotia A., Kapoor A and Kumar E. U. (2004), Are Metamorphic Viruses Really Invicible? Part1, *Virus Bulletin*, 12, pp. 5–7.

[14] Papadimitriou C. H. (1995), *Computational Complexity*, Addison Wesley, ISBN 0-201-53082-1.

[15] Post E. (1947), Recursive unsolvability of a problem of Thue, *Journal of Symbolic Logic*, 12, pp. 1–11.

[16] Qozah (1999), Polymorphism and grammars, *29A E-zine*, 4, `http://www.29a.net/`.

[17] Spinellis D. (2003), Reliable Identification of Bounded-length Viruses is NP-complete, *IEEE Transactions in Information Theory*, Vol. 49, No. 1, pp. 280–284.

[18] Tzeitzin G.C (1958), Associative calculus with an unsolvable calculus problem, *Tr. Math. Inst. Steklov Akad. Nauk SSSR*, 52, pp. 172–189.

[19] van Wijngaarden A., Mailloux B.J., Peck J.E.L., Koster C.H.A., Sintzoff M., Lindsey C.H., Meertens L.G. and Fisker R.G. (1975), Revised Report on the Algoithmic language ALGOL 68, *Acta Informatica*, 5, pp. 1–236.

[20] Zuo Z. et Zhou M. (2004), Some further theoretical results about computer viruses, *The Computer Journal*, Vol. 47, No. 6.

[21] Zuo Z, Zhou M. (2005), On the Time Complexity of Computer Viruses, *IEEE Transactions in Information Theory*, (51), 8.