Automated Extraction of Polymorphic Virus Signatures using Abstract Interpretation

Serge Chaumette, Olivier Ly, Renaud Tabary
Laboratoire Bordelais de Recherche en Informatique
University of Bordeaux, France
{chaumette, ly, tabary}@labri.fr

***Abstract*—In this paper, we present a novel approach for the detection and signature extraction for a subclass of polymorphic computer viruses. Our detection scheme offers 0 false negative and a very low false positives detection rate. We use context-free grammars as viral signatures, and design a process able to extract this signature from a single sample of a virus. Signature extraction is achieved through a light manual information gathering process, followed by an automatic static analysis of the binary code of the virus mutation engine.**
Keywords: binary program analysis; virus detection; virus signatures extraction; abstract interpretation;

I. Introduction

Since the creation of the first computer virus, virus authors and antivirus vendors have constantly fought in an evasion/detection game. Computer malwares have become more and more sophisticated, using advanced *code obfuscation* techniques to resist antivirus detection. *Polymorphic* and *metamorphic* computer viruses are currently the hardest kinds of viruses to detect. Both types of viruses are able to mutate into an infinite number of functionally equivalent copies of themselves. The set of all possible copies of a given virus V is called the *language* of the virus, denoted by L_V.

In this paper, we will only consider polymorphic viruses. Such a malware is composed of three parts: the virus body, the mutation engine and the decryptor (figure 1). At infection time, a polymorphic virus V will replicate itself in a new virus $V' \in L_V$ by picking a random symmetric encryption key k, ciphering the virus body and the mutation engine with this k and finally generating a new decryptor, embedding k. Because the virus body and the mutation engine are ciphered with a changing random key, detecting a polymorphic virus often means detecting its clear-text decryptor. Thus, for this kind of virus, L_V is the language of the decryptor, instead of the language of the whole virus.

In spite of the enormous industrial stakes, methods for detecting polymorphic malwares have not significantly been improved over the last decades. Today's most wide-spread malware detectors use signature-based methods to recognize threats (see [1], [2]). Signatures are usually regular expressions, mostly byte patterns which are unique to the malicious programs to detect [3]. The goal is to match all instances of a given virus, i.e to recognize a superlanguage of L_V, but as close as possible to it. The ease of use of such detectors as well as their relatively low false positives rate have led to their widespread deployment. However antiviral solutions currently face two problems:

- the number of unique malware discovered per day is quickly raising (~ 8000 new signatures each day) [4];
- as viruses tend to be written by professional profit-driven virus writers, malwares are harder to detect, due to the wide use of sophisticated *mutation engines*.

The current detection methods are not effective against highly-mutating polymorphic viruses. The problem is twofold. On the one hand, the set of all instances of a polymorphic malware may form a complex formal language [5] and thus cannot be recognized by means of regular expressions. Classical regular expression based approaches have to recognize a superlanguage of the virus, leading to high false positive rates. Second, the signatures extraction process will eventually fail to handle such an increasing number of new unique malwares. In fact, the current signatures extraction process is either based on a manual analysis, or based upon *learning*, as we will see later. It has been proven that learning complex language classes, such as context-free or even regular languages, is not possible from only positive inputs [6]. Industrials have managed to extract malware signatures until now thanks to the huge manpower at their disposal and the low sophistication of polymorphic engines. But the evolutions in the malware field may soon make this approach untractable.

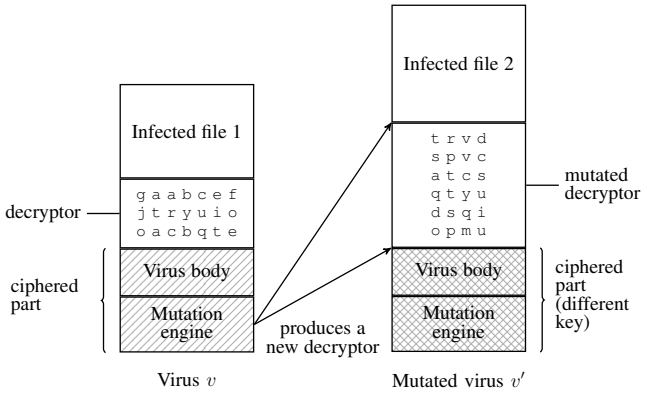

Figure 1: Polymorphic virus infection process

In this paper we propose a novel approach for signatures extraction and detection of highly polymorphic self-replicating malwares. Our work is based upon two ascertainments. First, most mutating engines used in current malwares produce code belonging to a language that is rather low in complexity. Experience showed that code produced by such engines often belongs to either a regular language or a **context-free language**. Some advanced polymorphic malwares are able to produce viral code that can be modeled by a context-sensitive language, but as we will see later they can be overapproximated by a context-free language without a significant loss of

precision. Second, self-replicating malwares have to **embed their mutation engine** inside their body. It is thus easy for an antivirus company to obtain the code of the mutation engine.

Our idea is that, under some conditions, it is feasible to **extract the grammar of the mutation engine** via a **static analysis** of the binary code of the engine. In other words, the binary code of the mutation engine expresses indirectly the grammar of the language that it produces. This grammar can be overapproximated by a context-free grammar, and then used as a signature able to match **all instances** of any virus that uses the same mutation engine. Our contribution is twofold:

- We use **context-free grammar** as signature, improving the overall detection precision. Moreover, we guarantee that our signature can match **all instances** of the virus. The matching being more precise, is also less likely to issue *false positives*;
- We design a process to extract the signature of a polymorphic virus from a **single sample** of this malware. Our process is composed of a short manual analysis (locating the mutation engine and where mutated code is written) followed by a static analysis to **automatically extract** the viral signature.

With our solution, the amount of manpower required to detect a polymorphic virus is limited to the malware mutation engine identification. The rest of the process, from malware grammar extraction to the production of a matcher program that can match all possible variants of a polymorphic virus, is fully automated. Note that our work can extend to the detection of **other kinds of malwares** using polymorphic-alike mutation engines, such as worms or trojan horses, provided one has access to their mutation engine.

In the rest of this paper, we will review the recent academic work in the field (section II), and then present the formal model used for the static analysis of malicious executable files and the overall idea of this new technique (section III). We will explain the signature grammar extraction process (section IV) and eventually show some results (section V).

II. Related work

The theoretical limits of malicious program detection has been the subject of many research projects. It is well known that the detection problem is undecidable in the general case [5], [7]. As we have previously seen, sophisticated malwares make in fact great use of software obfuscation techniques in order to avoid detection. For polymorphic malwares, the decryptor part of the virus is mutated at each infection, thanks to common obfuscation techniques: dead-code insertion, register reassignment, and instruction replacement. A more exhaustive list of obfuscation operations can be found in [8], [9]. In order to detect such high-mutating viruses, several solutions have been developed.

A. Byte-level detection solutions

Current antiviral solutions use different techniques in order to detect malicious files.The techniques are : pattern-matching, emulation, dynamic behavioral detection, and various heuristics [1]. Let us focus on pattern-matching techniques (heuristics are aimed at new malware detection and are subject to a high false positive rates, while emulation may not always succeed; dynamic malware detection, while achieving good results, is out of the scope of this paper). Because they have time and complexity constraints, the models and detection algorithms used in today's antiviral products are relatively simple. Programs are modeled as a sequence of bytes. Viral signatures are regular expressions, approximating the language L_V of the virus. The detection algorithm consists of determining whether a given binary program is recognized by one of the viral signature. Since regular expressions are used as signature descriptions, antivirus products may make use of finite state automatons to perform linear-time detection. The counter-part of such a simple model is a relatively high false positives rate: regular expressions are not expressive enough to precisely model the language of the virus and may lead to a too wide overapproximation of L_V.

Most of the actual signatures extraction techniques are based upon *learning*. A malicious file v is replicated inside a controlled environment, resulting in a (possibly small) subset of the virus instances, and the language of the virus is learnt. Different techniques are used to achieve this process. The most common techniques ([3], [10]) are based upon *binary diffing*: the longest common pattern that appears in all infected files is extracted. This technique, while achieving good results against simple viruses, is not able to extract a signature from sophisticated polymorphic viruses. For this kind of malwares, these techniques rely on **manual analysis**. Work by Sung et. al. [11] has explicitly targeted polymorphic malwares, but still relies on the presence of a common core shared by all instances of the virus. Other works [12] try to use multiple signatures in order to decrease both false positives and false negatives rates, but no implementation has been proposed.

Another emerging approach consists in using machine-learning techniques in order to detect malicious files [13]. Several models have been tested: data-mining [14], Markov chains on n-grams [15], [16], Naive Bayes as well as decision trees [17]. These methods provide an automatic way to extract signature from malicious executables. But while the experiments has shown good results, the false positive and negative rates are still not negligible.

Both binary-diffing and machine-learning based techniques also suffer from a common drawback. In order for these techniques to have a low false negatives rate, they must replicate the malware a large number of times. Many polymorphic malwares are aware of this fact and use **slow polymorphism**: the mutation of the virus does not happen at each infection, but rather when a particular, hard to simulate, condition is met.

B. Structural and semantic models

The pattern-matching solutions discussed above are based on a model vulnerable to code obfuscation techniques. If one specific mutation has not been observed during the learning phase, most byte-level detection schemes will be likely to miss any malicious instance of the virus featuring this unseen mutation [2], [18]. In order to circumvent this problem, recent work in the academic field has focused on the design of new, mutation-resistant, models. Malwares are not seen as a sequence of bytes, but are abstracted as a higher-level representation that is less likely to be defeated by code obfuscation.

In [19], [20], graphs are used as a model for malwares. The control flow graph (CFG) of a malware is computed (when

possible). Then, subsets of this graph are used as a signature. Detection of a malware is done by comparing a suspicious file against these sub-CFGs, and seeing if any part of the CFG of the file is equivalent (with a semantics-aware equivalence relation for [19]) to a sub-CFG in the viral database. The idea is that most of mutation engines' obfuscations do not alter the control flow graph of the malware. Unfortunately, this is not always the case (e.g. Zmist virus [21]), and the problem of the detection falls into the NP class.

In [22], Kinder et. al. used *CTPL* to detect malwares. CTPL is a variant of *CTL* [23], able to handle register-renaming obfuscation. Detection is done via model checking of API call sequence, while signatures extraction is done manually.

A promising approach was initiated by Preda et. al [24] and followed by [25]. It consists in using the *semantics* of a metamorphic malware as a viral signature. As we have seen previously, mutation engines produce *functionally equivalent* mutated instances of a virus. In order to compare the semantics of binary files, programs are *abstracted* using abstract interpretation techniques, and their abstract computation tree is then compared. Unfortunately, virus detection cost suffers from the complexity of the analysis, and no automatic signatures extraction algorithm is given in [24], [25].

In [5], [26] formal grammars were used as a viral signature. Unfortunately, the design of the grammar is based on the knowledge of the mutation transformations used by the mutation engine of the virus. None of them provides an automated process to extract this grammar for a given malware.

To the best of our knowledge, the only work similar to ours that uses abstract interpretation for extracting the signature of a mutating malware is by Preda et. al [27]. Their goals are more ambitious than ours, as they model a whole metamorphic virus. Using abstract interpretation, they overapproximate the malware execution traces as finite state automata. Our work differs from theirs as we only analyze the mutation engine of polymorphic viruses and we use context-free grammars as signature, which are potentially more precise. We also have studied and stressed the feasibility and complexity of our solution: we provide an implementation and test it against real viruses.

III. Idea and theory

A. Our idea

In order to circumvent the undecidability result in the malware detection problem [5], we restrain ourselves to the detection of a special class of polymorphic malwares. We will focus on polymorphic viruses using mutation engines that can be approximated as *CF-programs*. These engines produce machine code that can be recognized, or over-approximated by, a context-free language. Our detection solution gives relatively good results on mutation engines which have the following three properties:

1) the produced code is written sequentially, to a statically computable memory location;
2) no statically unresolvable dynamic call is performed, besides function returns which are handled by the function discovery.
3) context-insensitive behavior: the mutation engine makes limited use of global input variables. Input variables would be over-approximated to top by our numerical analysis and would lead to imprecision in detection. In particular, the virus mutation engine must **produce** obfuscated code, and not mutate existing code.

Most of existing polymorphic viruses fall in this subclass. In fact our solution could apply to other polymorphic malwares (worms, trojan horses) provided we have access to the mutation engine. Although the third point is the most limiting one. The metamorphic viruses would not give good results since their mutation engine is in fact a decompiler/compiler: the machine code of generation n is used as input by the mutation engine to produce the generation $n+1$ code.

We use a hybrid approach to detect the particular class of malwares we target. First, we use **syntactic** matchers to detect malwares. Using simple byte-level models for the detection allows **efficient** and practical detection algorithms, unlike most semantics-aware detection schemes. Thus our viral model and detection algorithm is very similar to the work presented in section II-A. Nevertheless, we use **context-free grammars as viral signatures** instead of regular expressions in order to improve detection precision.

Second, we use a **semantics-aware** approach to extract the viral signature. The signature extraction schemes used in previous work, based on learning, suffer from many drawbacks (cf. section II-A). We propose to perform a static analysis of the mutation engine of a malware in order to extract its signature. A short manual analysis is first achieved, gathering informations on the engine such as its location and the address where mutated code is produced. Then, a static analysis of the engine binary is performed, in order to **automatically extract the signature** of the malware. Our scheme requires **one sample** of the malware to infer a signature matching **all virus instances**. We will base our reasoning on an abstract model of mutation engines, called the CF-program model.

B. CF-programs semantics

CF-programs are high-level abstractions of programs behaving like push-down automata. A CF-program P is defined over an alphabet Σ: each run of a given CF-program will produce a word $w \in \Sigma^*$. The set of all CF programs is denoted $\mathbb{P}_{CF}$. The set of all words that can be produced is called the *language* of the program, denoted by $L(P)$.

Definition 1: A CF-program $P \in \mathbb{P}_{CF}$ is a tuple $P = (F, I, e, \gamma, \Sigma)$ where:

- $I \subseteq \mathbb{N}^*$ is the set of instruction addresses;
- $F \subseteq I$ is the set of function entry points;
- $e \in F$ is the program entry point;
- $\gamma : I \to L_{CF}$ associate to each instruction address the corresponding instruction in the L_{CF} language;
- Σ is the alphabet of the output language of P.

Instructions I of a CF-program are defined under the program language L_{CF} defined figure 2. Intuitively, a CF-program is a program where functions are clearly identified (the set F), supporting function calls (`call` and `ret` instructions). CF-programs are able to output tokens belonging to the Σ alphabet via the `write` W instruction, that outputs a token chosen arbitrarily in the set W of words. They also feature a `jND` instruction that perform a undetermined jump to either location j_1 or j_2 (see figure 2). The set $States_P$ of the states of a CF-

L_{CF} ::= `jmp` j_1 | `jND` j_1 j_2 | `call` f | `ret` | `write` W | `end`

$P = (F, I, e, \gamma, \Sigma)$ (program) $W \in 2^{\Sigma^*}$ (values)

$j_1, j_2 \in \mathbb{I}$ (address) $f \in F$ (function)

Figure 2: The L_{CF} language

program $P \in \mathbb{P}_{CF}$ is defined as a set of tuples $\langle i, S, B\rangle$ where $i \in I \cup \{0\}$ is the address of the next program instruction to be executed, $S \in I^*$ is the current call stack of the program, $B \in \Sigma^*$ is the sequence of output tokens generated till this point. The operational semantics which formalizes one-step of execution of a CF-program is given figure 3 as a relation $\rightsquigarrow$, such that $\rightsquigarrow \subseteq States_P \times States_P$. We let $\overset{*}{\rightsquigarrow}$ denote the transitive closure of $\rightsquigarrow$.

$$\frac{\texttt{i: ret}}{\langle i,t::s,b\rangle \rightsquigarrow \langle t,s,b\rangle} \qquad \frac{\texttt{i: jmp } j}{\langle i,s,b\rangle \rightsquigarrow \langle j,s,b\rangle} \qquad \frac{\texttt{i: end}}{\langle i,s,b\rangle \rightsquigarrow \langle 0,s,b\rangle}$$

$$\frac{\texttt{i: jND } j_1\ j_2}{\langle i,s,b\rangle \rightsquigarrow \langle j_1,s,b\rangle} \qquad \frac{\texttt{i: jND } j_1\ j_2}{\langle i,s,b\rangle \rightsquigarrow \langle j_2,s,b\rangle} \qquad \frac{\texttt{i: call } f}{\langle i,s,b\rangle \rightsquigarrow \langle f,i+1::s,b\rangle}$$

$$\frac{\texttt{i: write } W,\ W \in 2^{\Sigma^*}}{\langle i,s,b\rangle \rightsquigarrow \langle i+1,s,b::w_i\rangle, w_i \in W}$$

Figure 3: CF-program operational semantics

Let $P = (F, I, e, \gamma, \Sigma)$, we define $L(P) \subseteq \Sigma^*$ as the *language* of the program P: $L(P) = \{b \mid \langle e, \emptyset, \epsilon\rangle \overset{*}{\rightsquigarrow} \langle 0, s, b\rangle\}$ The language of a CF-program is the set of all words the executions of this program may produce. For a mutation engine of a polymorphic virus V, it will be the set of all decryptors, in other words the language L_V.

Theorem 1: Let $P = (F, I, e, \gamma, \Sigma) \in \mathbb{P}_{CF}$. A context-free grammar G_P recognizing $L(P)$ can be extracted from P using algorithm 1.

It is easy to see that CF-programs are equivalent to push-down automata. There is in fact a one-to-one mapping between a CF-program state $s = \langle i, S, B\rangle \in States_P$ and any word $w \in (V \cup \Sigma)^*$ such that w has been constructed from the grammar start symbol using only leftmost derivations.

$$\langle i, s_j...s_m, b_0...b_n\rangle \leftrightarrow b_0...b_n V_i V_{s_i}...V_{s_m}$$

Using this one to one mapping, one obtains immediately that for each leftmost derivation chain of G_P, there exists a transition sequence producing the same word, and vice versa.

The definition of the CF-program model and the grammar extraction algorithm allows us to design the basis of our antiviral solution. We designed a static analysis that overapproximates a polymorphic engine P as a CF-program P_{CF}. Then, by applying algorithm 1 to the result of this analysis, we are able to extract a signature (a context free grammar) recognizing $L(P_{CF})$ from a single sample of a virus. In this way we avoid the drawbacks of learning-based signature extraction solutions, and thus the problem issued by slow polymorphic engines. Moreover, provided the static analysis is sound, i.e $L(P_{CF})$ is a superlanguage of $L(P)$, the signature issued by our solution will lead to no false negative. The false positive rate depends on the precision of the analysis. To summarize, our virus signature extraction and detection scheme involves the following steps:

1) Extract the mutation engine $P \in \mathbb{P}$ of the virus V and locate where mutated code is written (*manual analysis*);
2) Approximate (sound) P to a CF-program $P_{CF} \in \mathbb{P}_{CF}$;
3) Extract the grammar G of P_{CF} using algorithm 1;
4) Use G as a signature to detect all variants of V.

The first step, while being manual, is just a matter of locating and extracting the mutation engine from the virus binary code. This kind of task is a lot quicker and easier than manual signature extraction, and should not be an issue for experimented malware analysts. The third step has been already covered in section III-B and algorithm 1 shows the basis of the process. The last step, generating a push-down automaton recognizing exactly the grammar G, is a well-known classical process (see e.g. [28]). The difficult point in our scheme resides in step 2, i.e. the overapproximation of the virus mutation engine by a CF-program producing a superlanguage of the virus language. This step is the subject of the next section.

Algorithm 1: Grammar extraction

input : $P = (F, I, e, \gamma, \Sigma) \in \mathbb{P}_{CF}$
output: $G_P = (V, \Sigma, R, S)$

$V \leftarrow \{V_i, i \in I\}$;
$R \leftarrow \emptyset \times \emptyset$;
foreach $i \in I$ **do**
  switch $\gamma(i)$ **do**
    case `write` $W = \{w_0, ..., w_n\} \in 2^{\Sigma^*}$
      foreach $w_i \in W$ **do**
        $R \leftarrow R \cup \{V_i \rightarrow w_i V_{i+1}\}$;
    case `call` f
      $R \leftarrow R \cup \{V_i \rightarrow V_f V_{i+1}\}$;
    case `jmp` j
      $R \leftarrow R \cup \{V_i \rightarrow V_j\}$;
    case `jND` j k
      $R \leftarrow R \cup \{V_i \rightarrow V_j | V_k\}$;
    case `return` **or** `end`
      $R \leftarrow R \cup \{V_i \rightarrow \epsilon\}$;
return (V, Σ, R, V_e)

IV. Binary mutation engines as CF-programs

When analyzing malwares, mutation engine comes in binary form and is sometimes obfuscated. Translating a mutation engine from its binary form to a more abstract CF-program is not a straightforward task. Binary machine code often comes with a complex semantics. We propose here a static analysis able to approximate such a complex semantics by a CF-program through the following steps:

1) Abstraction of the machine code semantics: translation to the Dynamic Binary Automaton [29] (DBA) model;
2) Pre-analysis (engine extraction, function discovery);
3) Numerical analysis of the binary program;
4) Translation to a CF program.

Each of these steps will be presented in this section.

A. Binary model

We use the DBA model in order to abstract the machine-dependant assembly language of the mutation engine. DBA programs feature a reduced instructions set whose side-effects are explicit. A simplification of the language is presented figure 4. Each instruction is given a unique address in $\mathbb{PP}$, the

$\mathbb{I}$::= **Assign** [$\mathbb{E}$], $\mathbb{E}$	standard assignment lhs := rhs
| **Guard** $\mathbb{E}$, n_1, n_2	guard on conditions $\mathcal{E}$
| **Skip**	no operation
| **Jump** $\mathbb{E}$	non-statically known jump to expression

$\mathbb{E}$::= n[x:y[| ($\mathbb{E}$ op $\mathbb{E}$)[x:y[| [$\mathbb{E}$][x:y[

Figure 4: The DBA language

set of program points. A concrete state in the DBA semantics is a pair $\langle ip, m \rangle$ where $ip \in \mathbb{PP}$ is the current instruction address and $m : \mathbb{N} \to \mathbb{B}$ is the memory environment of the program, a map associating to each memory cell its content, a bitvector of size 8. For the sake of concision we assume that the processor registers are given a unique special address in this map. The concrete semantics of the language is straightforward and will not be explained there. The BINCOA framework features a set of decoders able to translate a `x86` or `ARM` program to this DBA model [29].

B. Pre-analysis

In order for the signature extraction scheme to succeed, a few preliminary steps need to be done.

a) Mutation engine extraction: The first step of our scheme consists in locating the mutation engine inside a single instance of the analyzed virus. This step must be performed through manual reverse engineering of the malicious sample. Hopefully, mutation engines are kept in the ciphered part of the polymorphic virus that is not likely to be detected by an antivirus, and most of the time are not obfuscated. Thus, a simple memory dump from a debugger performed by a qualified malware analyst once the mutation engine appears in clear text should suffice. In the following sections, we assume that the mutation engine has been successfully extracted and translated to the DBA model. In order to illustrate our analysis, we will use an extract of the `ETMS` engine, a mutation engine used in `Aldebaran` and `Antares` virus families. A small code snippet is presented figure 5 in assembly syntax. This mutation engine sample produces x86 machine code that will perform a stack push. The malware author uses this subroutine of the mutation engine to produce *junk code*. Because of its length, the DBA model cannot be presented here.

b) Writes identification: The next step of our pre-analysis consists in identifying the memory location where mutated code is written by the mutation engine. Most of the time, mutation engines output binary code sequentially to either a statically known memory location, or through a predefined input register (for example the `edi` register in most of x86 mutation engines). In our detection scheme, this location (an address range or a register) must be retrieved manually through a short manual code analysis and provided to the analysis. We model this information as a function $IsWrite : \mathbb{E} \to \{true, false\}$ that tells us if a memory

```
1  ;edi--> produced
       code
2  cmp   ebx,0
3  je    22
4  call  26
5  shr   al,1
6  jnc   17
7  mov   ax,0c483h ;"
       sub esp, <cst>"
8  stosw
9  call  23
10 jz    9
11 cmp   al,ebx
12 ja    9
13 sub   ebx,al
14 shl   al,2
15 stosb
16 ret
17 call  23
18 or    cl,058h ;"push
       <reg>"
19 xchg  al,cl
20 dec   ebx
21 stosb
22 ret
23 call  26 ;[0-7] in
       eax
24 and   eax,7
25 ret
26 rdtsc ;random in eax
27 ret
```

Figure 5: Extract from ETMS mutation engine

location expressed as a DBA expression belongs to the output buffer of the mutation engine.

The $IsWrite$ function can either use the result of our numerical analysis to discover if an `Assign` instruction writes to the output buffer, or use simple pattern matching if, for example, writings are done through a predefined register. The latter case is preferred, when possible, since it does not suffer from the overapproximation that takes place during the numerical analysis. In figure 5, if we consider that the produced code is written at `[edi]`, three token outputs would be identified, at lines 8, 15 and 21.

c) Functions discovery and dynamic jumps resolution: The last pre-analysis step is automated and consists in identifying the set of functions in the DBA-model of the virus. This kind of information is not available in a binary program. We try to identify sequences of DBA instructions whose behavior is equivalent to a `call` or a `ret` in the CF-program semantics (cf. figure 3). Once `calls` and `rets` are located, we define functions as all the instructions located on the path from a `call` target to any `ret`-alike instruction in the DBA model. Function `calls` and `rets` may be identified in two ways:

- For most of the architectures (including x86), the instruction set contains `call` and `return` alike opcodes. These can be used for `call`/`ret` identification;
- When function calls are obfuscated, we must analyze precisely the DBA model to identify equivalent DBA-instructions. Such techniques are presented in [30]. Most of the times, they are imprecise (i.e. identify too many functions) but are able to deal with obfuscated code.

Concerning the example of figure 5, the function identification step highlights three functions at lines 2-22, 23-25 and 26-27. Once function are identified, we modify the DBA model of the mutation engine the following way:

- function calls are replaced with a single `Skip` DBA instruction, whose successor is the instruction following the `call`;
- function returns are replaced with a `Skip` instruction;
- dynamic jumps, if any, are resolved using state of the art binary CFG reconstruction techniques [22];
- three sets L_{funcs}, L_{calls} and L_{rets} are computed, containing respectively the function entry point locations, the identified calls locations and the returns locations.

The result of this last step is a set of DBA programs, one for each function of the program, featuring no dynamic jump.

C. Numerical analysis

The semantics of the DBA model is too rich to be directly translated as a CF-program. For example, the `Assign` instruction in the DBA model has no image in the CF-program semantics, as CF-programs only use the stack. To solve this issue, we use abstract interpretation techniques [31] to design a static analysis of the mutation engine. Our goal is to overapproximate the DBA concrete semantics, in particular the content of the concrete memory at each program point. This information will be used later to discover which values are written by the mutation engine at writing locations.

We perform a context-insensitive symbolic analysis that overapproximates concrete memory cells content with symbolic terms. The main structure of the analysis is a map S^k that we call *summary*. It associates to each concrete memory address a disjunctive formula of k expressions, where k is a fixed analysis parameter. By appending two special elements top ($\top$) and bottom ($\bot$) to the co-domain of the summary function, we are able to build a complete lattice.

$$S^k : \mathbb{N} \to (\mathbb{E} \cup \{\top, \bot\})^k$$

The union of two summaries is defined intuitively as the union of the disjunctive terms for each memory cell address, or $\top$ if the resulting disjunctive formula contains more than k elements. The abstract domain used during analysis is also a complete lattice $A^k \subseteq \wp(\mathbb{PP} \times L_S^k \times F_S^k)$ where $L_S^k = \mathbb{PP} \times S^k$ associates to each program point a summary overapproximating the concrete program state at this address and $F_S^k = L_{funcs} \times S^k$ associates to each function the summary approximating the concrete state at any return location of the function. Because of space constraints, the abstract predicate transformer $P_A^k : A^k \to A^k$ cannot be presented here. Instead, we give a few insights about the analysis:

- the analysis can be seen as alternating between intraprocedural phases, where function and location summaries are computed for each function using expression substitution, and interprocedural phases where computed function summaries are used to refine the location summaries at call sites;
- the intraprocedural analysis is path-insensitive: multiple paths are merged at join nodes in the CFG, during fixpoint computation;
- the interprocedural analysis is context-insensitive: one unique summary is associated to each function. Although, when context insensitiveness leads to too much imprecision, some functions may be inlined;
- convergence is ensured through the use of widening on summaries: disjunctive formulas of terms are set to top $\top$ whenever their size exceeds k, or when any expression of the formula exceeds a given depth;

All along the analysis, special attention is paid to the simplification of formulas in summaries. In particular, mutation engines often construct opcodes at a bit-level precision. Thus, it is very important to compute most of the bits of each term. The symbolic analysis is well suited for this purpose, as expressions feature a bitfield concatenation operator. The analysis embeds several rewriting rules that focus on determining the maximum of bits in the term. For example, in figure 5, the symbolic evaluation process would give the following term for the token output at line 21:

$$58[0{:}8[\texttt{ OR } (\top[0{:}8[\texttt{ AND } 7[0{:}8[)$$

Instead of over-approximating the term to $\top$, we use term-rewriting rules to compute most of its content (figure 6).

Rewriting rules:
$x[0{:}a[\texttt{ AND } (2^k - 1) \to x[0{:}k[:: 0[0{:}a - k[$
$x[0{:}a[\texttt{ OR } (y[0{:}b[:: 0[0{:}a - b[) \to (x[0{:}b[\texttt{ OR } y[0{:}b[) :: x[b{:}a[$
Result:
$58[0{:}8[\texttt{ OR } (\top[0{:}8[\texttt{ AND } 7[0{:}8[) \to \top[0{:}3[:: 58[3{:}8[$

Figure 6: Bit-level term rewriting example

The result of the numerical analysis is a function $Value : \mathbb{PP} \to S^k$ giving for each program point the summary overapproximating the concrete state when the instruction at this address has been executed.

D. Extracting the grammar and generating the decoder

In the last part of our signature extraction scheme, we translate the mutation engine P from its DBA model representation to a CF-program P_{CF}. After the pre-analysis and the numerical analysis, we have access to the $IsWrite$ function, the L_{calls} and L_{rets} sets and the result of our numerical analysis $Value : \mathbb{PP} \to S^k$. We also define a concretization function $\gamma_S : S^k \times \mathbb{E} \to \wp(\mathbb{B})$ that returns for a given summary and an expression a set of bitfields overapproximating the expression value in the concrete semantics. Translation to a

$$\omega[\![i : \texttt{Assign}[e_1], e_2]\!] = \left\{ \begin{array}{l} \texttt{jmp } i+1 \ if \neg IsWrite(e_1) \\ or \ \texttt{write } \gamma_S(Value(i), e_2) \end{array} \right\}$$
$$\omega[\![i : \texttt{Guard } e, n_1, n_2]\!] = \texttt{jND } n_1, n_2$$
$$\omega[\![i : \texttt{Skip}]\!] = \left\{ \begin{array}{ll} \texttt{call } f & if (i, f) \in L_{calls} \\ \texttt{ret} & if \ i \in L_{rets} \\ \texttt{jmp } i+1 & otherwise \end{array} \right\}$$

Figure 7: DBA programs to CF-Programs

CF program P_{CF} is done through a straightforward translation function ω from DBA instructions to CF-program instructions, given figure 7. Note that since we resolved dynamic jumps, the DBA `Jump` instruction needs not to be translated.

Since $Value$ returns an overapproximation of the concrete program state, P_{CF} will also be an overapproximation of P, in the sense that $L(P) \subseteq L(P_{CF})$. A grammar G recognizing $L(P_{CF})$ is then extracted using algorithm 1. A particular care is taken when the concretization function γ_S returns the whole set of bitfields $\mathbb{B}$, i.e. when the numerical analysis has approximated a value as $\top$. In that case, a special rule will be added to the grammar that will match the language $\{0, 1\}^k$, where k is the maximum length of the bitfields. For example, the grammar matching the language produced by the code figure 5 is given figure 8.

$AStart = A \mid \epsilon$
$T = 0[0:1[\mid 1[0:1[$
$A = T\ T\ T\ \texttt{058}[3:8[\mid \texttt{0C483}[0:16[\ \texttt{00}[0:2[\ T\ T\ T\ \texttt{00}[0:3[$

Figure 8: Simplified grammar for code figure 5

The grammar G is then simplified (automatically) and used to generate a GLR parser recognizing $L(P_{CF})$ on the binary alphabet $\Sigma = \{0, 1\}$. This parser can be used to detect all variants of any virus using P as mutation engine. In the following section, we will show results concerning the precision of this parser, when tested against real-life malwares.

V. Implementation and preliminary results

A. Implementation

We have developed a proof of concept tool to apply our analysis provided the initial conditions are met (cf. III-A). It is written in `C++` and takes as input x86 and ARM infected executables in PE, ELF or Mach-O format. Additional information must be given to the analyzer:

1) the mutation engine location in the executable;
2) the memory address where the mutated code is written;
3) the random number generator seed variable location, as well as other instance-dependant input variables.

Items (1) and (2) are mandatory for the analysis to succeed, and have already been discussed. The third information is also important. Most mutation engines embed a pseudo-random number generator (PRNG). Because we analyze only one instance of a given virus, the seed of the generator will be fixed in this instance, and may be considered as a numerical constant by our analysis, leading to a signature matching only this particular instance of the virus. By pinpointing the location of the PRNG procedure in the code, the analysis makes sure the returned value is computed as $\top$, simulating the fact that this value is modified at each virus infection. Other instance-dependant values may also be concerned, e.g. virus size or virus virtual address. Note that most of the time those instance-dependent values are given as input to the mutation engine and are thus automatically overapproximated to $\top$ by our analysis.

Our tool has been tested against six mutation engines: the `CRPE` engine (used in some anonymous PE appenders), the `Bolzano` engine (used in `W32/Bolzano` virus), the `NBKPE` mutation engine, the `Offensive Poly Engine` (used in `W32/Annunaki` virus), the `Voltage` polymorphic engine (used in `W32/Norther`), the `FINE` mutation engine (used in `W32/Atix`) and the `ETMS` mutation engine (used in `Aldebaran` and `Antares` families). The mutation engines were either gathered from live virus samples or from the `vx.netlux` website [32]. Our tool was able to extract a grammar from all these engines. The output of the tool is a simple text file containing the extracted grammar in EBNF notation. Resulting grammars can be found online [33]. Processing times vary between a few seconds and 10 minutes, depending on the size of the engine and the chosen precision. The manual analysis never exceeded one man-hour.

B. The scanner

After the analysis process, the resulting grammar is further simplified and automatically "beautified" by additional python scripts. The goal of this step is to reduce the size of the grammar and parsing time as well as to increase grammar readability. The simplified grammar is then used by our scanner. The scanner is a standard GLR parser (we used LEPL [34] and Piggy [35] parsers), that is aware of the executable file format.

Scanning is performed on binary programs, starting at the entry point of the executable. Files are scanned against the grammar G produced by our tool, and are flagged as infected if there exists a word w inside the file, such that $w \in L(G)$. The actual implementation of the scanner has been written in python and parsing performances are not optimal at the moment.

C. Results

We tested our scanner against the seven mutation engines. For each mutation engine, tests were run against 1000 infected executables and 1000 different clean files taken from a Windows XP developer station. The results are presented figure 9. Scanning times ranged from 2 seconds to approximately 4 minutes per binary file, except for the ETMS engine. The

Name	Instrs	Rules	Ambgs.	FP	FN
CRPE	359	20	41	0	0
Bolzano	578	17	15	0	0
NBKPE	697	68	115	0	0
VoltagePE	1044	152	221	0	0
OffensivePE	1053	173	1203	22%	0
ETMS	1100	240	2792	-	-

Figure 9: Scanning results

four first mutation engines have been perfectly detected by the scanner. The Offensive Polymorphic Engine, while being correctly detected, lead to a huge false positives rate (22%). Our analysis was very imprecise against this mutation engine, mainly because of the complexity of its code: many values were computed inside loop bodies, and in a high context-sensitive manner. The ETMS mutation engine was correctly analyzed, and the grammar seemed correct. Nonetheless, the complexity of the grammar (in particular the high number of ambiguities) lead to a very long parsing time. Finally, The FINE mutation engine was not correctly analyzed as it does not respect our preconditions. Mutated code in FINE mutation engine is not written sequentially, but in two passes: a first code production step is then followed by a fix-up phase, where the produced code is further modified.

It should be noted that both the analyser and the scanner are proof-of-concept tools, and several improvements can be made regarding the precision of the analysis and the performance of the scanner. Moreover, we have not investigated grammar disambiguation techniques to cope with the large scanning time for the ETMS mutation engine.

VI. Conclusion and future work

In this paper, we designed a process able to extract a signature from a given polymorphic malware respecting realistic preconditions. Instead of learning the language of the virus, we designed an automated static analysis able to infer the malware signature by analyzing the binary code of its mutation engine. Our solution requires only one sample of a malware in order

to extract its signature. The result is a context-free grammar, matching a superlanguage of all instances of the virus (no false negative). The false positives rate of our detection scheme depends solely on the precision of the malware mutation engine static analysis. Tests have been performed on seven mutation engines, showing encouraging results for the six polymorphic viruses that met our preconditions.

Performance problems have emerged from our tests, as very ambiguous grammars lead to significant parsing times. These drawbacks came from the fact that our scanner is just a proof-of-concept, the lack of a proper grammar disambiguation pass in the framework as well as from the inherent complexity of parsing context-free languages.

We believe that the resulting context-free grammar should not be used as is in deployed antiviral solutions, but could serve as an additional security. Antivirus could use their efficient NFA scanners for early virus detection and then rely on GLR parser to decrease the false positives rate of their software. Another application of our solution could be to infer regular expression based signatures recognizing a superlanguage of our context-free language [36]. Using regular expressions as signature offer the great performances we are used to in current antivirus. While the overall detection precision may be less than GLR scanners, signatures generated by this method would benefit from the zero false negative rate of the original context-free signature, while suppressing the pitfalls of learning-based signatures extraction schemes.

The grammar of a polymorphic virus could also be a useful tool for virus classification. The obfuscation power of a polymorphic virus could then be quantified by, for example, counting the number of rules of its grammar in CNF form.

REFERENCES

[1] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[2] E. Filiol, "Malware pattern scanning schemes secure against black-box analysis," *Journal in Computer Virology*, vol. 2, pp. 35–50, 2006.

[3] J. O. Kephart and W. C. Arnold, "Automatic extraction of computer virus signatures," in *Proceedings of the 4th Virus Bulletin International Conference*. Virus Bulletin Ltd., 1994, pp. 178–184.

[4] H. Sverdlove. (2011, Feb) 20 years of malware security. Http://blog.bit9.com/bid/40340/What-s-the-Score-20-Years-of-Malware-Security.

[5] E. Filiol, "Metamorphism, formal grammars and undecidable code mutation," *International Journal of Computer Science*, vol. 2, pp. 70–75, 2007.

[6] E. Gold, "Language identification in the limit," *Information and control*, vol. 5, pp. 447–474, 1967.

[7] F. Cohen, "Computer viruses : Theory and experiments," *Computers & Security*, vol. 6, no. 1, pp. 22 – 35, 1987.

[8] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, Tech. Rep. 148, Jul 1997, http://www.cs.auckland.ac.nz/ collberg/Research/Publications/CollbergThomborsonLow97a/index.html.

[9] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.

[10] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," *Security and Privacy, IEEE Symposium on*, vol. 0, pp. 226–241, 2005.

[11] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static analyzer of vicious executables (save)," in *Proceedings of the 20th Annual Computer Security Applications Conference*, ser. ACSAC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 326–334. [Online]. Available: http://dx.doi.org/10.1109/CSAC.2004.37

[12] D. P. Shaohua, W. Jau-Hwang, S. Wen-Gong, Y. Chin-Pin, and T. Cheng-Tan, "Intelligent automatic malicious code signatures extraction," in *Security Technology, 2003. Proceedings. IEEE 37th Annual 2003 International Carnahan Conference*, 2003, pp. 600–603.

[13] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '04. New York, NY, USA: ACM, 2004, pp. 470–478.

[14] Y. Ye, D. Wang, T. Li, and D. Ye, "Imds: intelligent malware detection system," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 1043–1047.

[15] K. Griffin, S. Schneider, X. Hu, and T.-c. Chiueh, "Automatic generation of string signatures for malware detection," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, E. Kirda, S. Jha, and D. Balzarotti, Eds. Springer Berlin / Heidelberg, 2009, vol. 5758, pp. 101–120.

[16] W. Yan and E. Wu, "Toward automatic discovery of malware signature for anti-virus cloud computing," *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, vol. 4, pp. 724–728, 2009.

[17] J.-H. Wang, P. Deng, Y.-S. F. L.-J. Jaw, and Y.-C. Liu, "Virus detection using data mining techinques," in *Proceedings of the IEEE 37th Annual 2003 International Carnahan Conference on Security Technology*, 2003.

[18] M. Christodorescu and S. Jha, "Testing malware detectors," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 34–44, July 2004.

[19] ——, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 169–186.

[20] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "Control flow graphs as malware signatures," in *International Workshop on the Theory of Computer Viruses TCV'07*, Eric Filiol, Jean-Yves Marion, and Guillaume Bonfante, Eds., Nancy France, 2007.

[21] P. Ször and P. Ferrie, "Hunting for metamorphic," in *In Virus Bulletin Conference*, 2001, pp. 123–144.

[22] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science, K. Julisch and C. Kruegel, Eds. Springer Berlin / Heidelberg, 2005, vol. 3548, pp. 514–515.

[23] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, ser. Lecture Notes in Computer Science, D. Kozen, Ed. Springer Berlin / Heidelberg, 1982, vol. 131, pp. 52–71.

[24] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 5, pp. 1–54, 2008.

[25] F. Leder, B. Steinbock, and P. Martini, "Classification and detection of metamorphic malware using value set analysis," in *Malicious and Unwanted Software (MALWARE), 2009 4th International*, 2009, pp. 39 – 46.

[26] P. Zbitskiy, "Code mutation techniques by means of formal grammars and automatons," *Journal in Computer Virology*, vol. 5, pp. 199–207, 2009.

[27] M. D. Preda, R. Giacobazzi, S. Debray, K. Coogan, and G. M. Townsend, "Modelling metamorphism by abstract interpretation," in *Proceedings of the 17th international conference on Static analysis*, ser. SAS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 218–235. [Online]. Available: http://portal.acm.org/citation.cfm?id=1882094.1882108

[28] J. E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Pearson Addison Wesley, 2007.

[29] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The bincoa framework for binary code analysis," in *Submitted to CAV*, 2011.

[30] D. R. Boccardo, "Context-sensitive analysis of x86 obfuscated executables," Ph.D. dissertation, Universidade Estadual Paulista, 2009.

[31] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[32] Hermit. (vx.netlux.org/vx.php?id=eidx) Vx netlux website. [Online]. Available: vx.netlux.org/vx.php?id=eidx

[33] R. Tabary. (2011, May) Context-free signatures for some in-the-wild polymorphic viruses. [Online]. Available: www.labri.fr/perso/tabary/publis/polysigns.pdf

[34] A. Cooke. (www.acooke.org/lepl/) Lepl 4.4. [Online]. Available: www.acooke.org/lepl/

[35] T. Newsham. (www.lava.net/ newsham/pyggy/) Pyggy glr parser. [Online]. Available: www.lava.net/~newsham/pyggy/

[36] M.-J. Nederhof, "Practical experiments with regular approximation of context-free languages," *Comput. Linguist.*, vol. 26, pp. 17–44, March 2000. [Online]. Available: http://dx.doi.org/10.1162/089120100561610