HAL
open science

Architecture of a Morphological Malware Detector

Guillaume Bonfante, Matthieu Kaczmarek, Jean-Yves Marion

To cite this version:

Guillaume Bonfante, Matthieu Kaczmarek, Jean-Yves Marion. Architecture of a Morphological Malware Detector. Journal in Computer Virology, 2009, 5 (3), pp.263-270. 10.1007/s11416-008-0102-4 . inria-00330022

HAL Id: inria-00330022

https://inria.hal.science/inria-00330022v1

Submitted on 13 Oct 2008

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

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

Architecture of a Morphological Malware Detector

Guillaume Bonfante, Matthieu Kaczmarek and Jean-Yves Marion
Nancy-Université - Loria - INPL - Ecole Nationale Supérieure des Mines de Nancy
B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France

July 15, 2008

Abstract

Most of malware detectors are based on syntactic signatures that identify known malicious programs. Up to now this architecture has been sufficiently efficient to overcome most of malware attacks. Nevertheless, the complexity of malicious codes still increase. As a result the time required to reverse engineer malicious programs and to forge new signatures is increasingly longer.

This study proposes an efficient construction of a morphological malware detector, that is a detector which associates syntactic and semantic analysis. It aims at facilitating the task of malware analysts providing some abstraction on the signature representation which is based on control flow graphs (CFG).

We build an efficient signature matching engine over tree automata techniques. Moreover we describe a generic graph rewriting engine in order to deal with classic mutations techniques. Finally, we provide a preliminary evaluation of the strategy detection carrying out experiments on a malware collection.

Introduction

The identification of malicious behavior is a difficult task. Until now, no technologies have been able to automatically prevent the spread of malware. Several approaches have been considered but neither syntactic analysis nor behavioral consideration were really effective. Presently, human analysis of malware seems to be the best strategy, next malware detectors based on string signature remains the most reliable solution. From this point of view, we have tried to easier the task which consist in finding a good signature within a malicious programs. Our technique has been inspired from the article [6] where control flow graphs are used to detect the different instances of the computer virus `MetaPHOR`.

Generally speaking, detection strategies based on string signatures uses a database of regular expressions and a string matching engine to scan files and to detect infected ones. Each regular expression of the database is designed to identify a known malicious program. There are at least three difficulties tied to this approach. First, the identification of a malware signature requires a human expert and the time to forge a reliable signature is long compared to the time related to a malware attack. Second, string signature approach can be easily bypassed by obfuscation methods. Among recent work treating this subject, we propose to see for example [4, 7, 14]. Third, as the quantity of malware increase, the ratio of false positives becomes a crucial issue. And removing old malware signatures would open doors for outbreaks of reengineered malware.

Thus, a current trend in the community proposes to design next generation of malware detectors over semantic aspects. [11, 9, 20]. However, most of semantic properties are difficult to decide and even heuristics can be very complex as it is illustrated in the field of computer safety. For those reasons, in [5] we try to propose and to construct a *morphological analysis* in order to detect malware. The idea is to recognize the shape of a malicious program. That is, unlike string signature detection, we are not only considering a program as a flat text, but rather as a semantics object, so adding in some sense a dimension to the analysis. Our approach tries to combine several features: (a) to associate

syntactic and semantic analysis, (b) to be efficient and (c) to be as automatic as possible.

Our morphological detector uses a set of CFG which plays the role of a malware signature database. Next, the detection consists in scanning files in order to recognize the shape of a malware. This design is closed to a string signature based detector and so we think that both approaches may be combined in a near future. Moreover, it is important to notice that this framework make the signature extraction easier. Indeed, either the extraction is fully automatic when the malware CFG is relevant or the task of signature makers is facilitated since they can work on an abstract representation of malicious programs.

This detection strategy is close to the ones presented in [9, 6] but we put our strengths to optimize the efficiency of algorithms. For that sake, we use tree automata, a generalization to trees of finite state automata over strings [10]. Intuitively, we transform CFG into trees with pointers in order to represent back edges and cross edges. Then, the collection of malware signatures is a finite set of trees and so a regular tree language. Thanks to the construction of Myhill-Nerode, the minimal automaton gives us a compact and efficient database. Notice that the construction of the database is iterative and it is easy to add the CFG of a newly discovered malicious program.

Another issue of malware detections is the soundness with respect to classic mutation techniques. Here, we detect isomorphic CFG and so several comon obfuscation methods are canonically removed. Moreover, we add a rewriting engine which normalizes CFG in order to have a robust representation of the control flow with respect to mutations. Related works are [6, 8, 20] where data flow of programs is also considered.

The design of this complete chain of process is summarized by Figure 1.

We also provide large scale experiments, with a collection of 10156 malicious programs and 2653 sane programs. Those results are promising, with a completely automatic method for the signature extraction we have obtained a false positive ratio of 0.1%.

This study is organized as follows. First we expose the principles of CFG extraction and normalization. Then, we present a matching engine for CFG that is based on tree automata. Finally we carry out some experiments to validate our method.

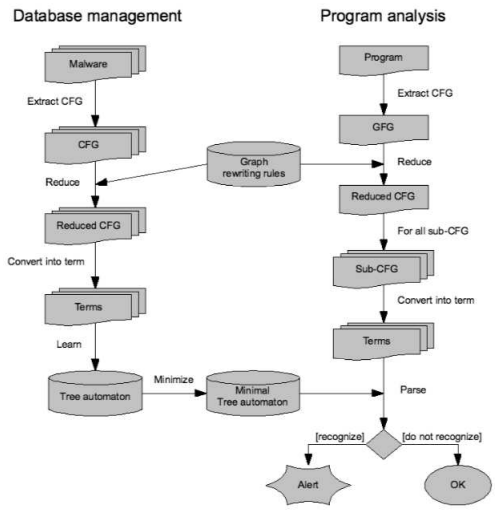


Figure 1: Design of the control flow detector

1 CFG in x86 languages

Road-map. Since we focus on practical aspects we choose to work on a concrete assembly language. This language is close to the x86 assembly language. We detail how to extract CFG from programs, we underline the difficulties that can be encountered and we outline how they can be overcome with classic methods. Finally, we study the problem of CFG mutations. We propose to normalize the extracted CFG according to rewriting rules in order to remove common mutations.

An x86 assembly language. We present the grammar of the studied programming language. The computation domain is the integers and we use a subset of the commands of the x86 assembly language. The important feature is that we consider the same flow instructions as in x86 achitectures, as a result the method that we develop can be directly applied to concrete programs.

Addresses	$\mathbb{N}$
Offsets	$\mathbb{Z}$
Registers	$\mathbb{R}$
Expressions	$\mathbb{E} ::= \mathbb{Z} \mid \mathbb{N} \mid \mathbb{R} \mid [\mathbb{N}] \mid [\mathbb{R}]$
Flow instructions	$\mathbb{I}^f ::= \texttt{jmp}\ \mathbb{E} \mid \texttt{call}\ \mathbb{E} \mid \texttt{ret} \mid \texttt{jcc}\ \mathbb{Z}$
Sequential instructions	$\mathbb{I}^d ::= \texttt{mov}\ \mathbb{E}\ \mathbb{E} \mid \texttt{comp}\ \mathbb{E}\ \mathbb{E} \mid \ldots$
Programs	$\mathbb{P} ::= \mathbb{I}^d \mid \mathbb{I}^f \mid \mathbb{P}\,;\mathbb{P}$

Next, a program is a sequence of instructions $\mathbf{p} = \mathbf{i}_0\,;\ldots;\mathbf{i}_{n-1}$. The address of the instruction $\mathbf{i}_k$ is k. In order to ease the reading and without loss of generality, we suppose that $\mathbf{i}_0$ is the first instruction to be executed, the address 0 is the so called entry point of the program.

We observe that the control flow of programs is driven by only four kinds of flow instructions. Given an instruction $\mathbf{i}_k \in \mathbb{I}^f$, the possible transfers of control are the following.

- If $\mathbf{i}_k$ is an unconditional jump `jmp` e. The control is transferred to the address given by the value of the expression e.
- If $\mathbf{i}_k$ is a conditional jump `jcc` x. If its associated condition is true, the control is transferred to the address $k + x$. Otherwise, the control is transferred to the address $k + 1$.
- If $\mathbf{i}_k$ is a function call `call` e. The address $k+1$ is pushed on the stack and the control is transferred to the the value of the expression e.
- If $\mathbf{i}_k$ is a function return `ret`. An address is popped from the stack and the control is transferred to this address.

Prerequisites. The extraction of the CFG from a program is tied to several difficulties. First, we need access to the instructions of the program. As a result packing and encryption techniques can thwart the extraction. This problem is part of the folklore, indeed classical string signature detectors also have to face those techniques. Many solutions such as sand-boxes and generic unpackers have been developed to overcome this difficulty. The presentation of those solutions exceeds the scope of the current study then we refer to the textbooks [13, 12, 18].

Second, we are confronted to obscure sequenceq of instructions such as `push a; ret` which have the same behavior as the instruction `jmp a`. This is also part of the folklore and we will suppose that such sequence of instructions are normalized during the disassembly phase of the extraction.

Third, the target addresses of jumps and function calls have to be dynamically computed. For example, when we encounter the instruction `jmp eax` we need the value of the register `eax` in order to follow the control flow. In such cases we rely on an heuristic $(\!|e|\!)$ which provides the value of the expression e if it can be statically computed. If the value cannot be computed then $(\!|e|\!) = \bot$. Such an heuristic can be based on partial evaluation, emulation or any other static analysis technique.

The extraction procedure. The control flow consists in the different paths that might be traversed through the program during its execution. It is frequently represented by a graph named a control flow graph (CFG). The vertices stand for addresses of instructions and the edges represent the possible paths that the control flow can follow.

We suppose that we have access to the code of programs and that we have an heuristic $(\!|\ |\!)$ to evaluate expressions. Table 1 presents a procedure to extract CFG from programs. We observe that this procedure closely follows the semantics of flow instructions. Indeed, the vertices of the CFG are labeled accordingly to the instruction at the night address and the nodes are linked according to the possible control transfers.

- The symbol `inst` of arity 1 labels addresses of sequential instructions. There is only one successor: the address of the next instruction.
- The symbol `jmp` of arity 1 labels addresses of unconditional jumps. There is only one successor: the address to jump to.
- The symbol `jcc` of arity 2 labels addresses of conditional jumps. There is two successors: the address to jump to when the condition is true and the address of the next instruction where the control is transfered when the condition is false.
- The symbol `call` of arity 2 labels addresses of function calls. There is two successors: the address of the function to call and the return

address, that is the address of the next instruction.

- The symbol `end` of arity 0 labels addresses of function returns and undefined instructions. There is no successor.

The entry point of the program correspond to the root of the CFG.

Instruction	Graph
$\mathbf{i}_n \in \mathbb{I}^d$	
$\mathbf{i}_n = \texttt{jmp}\ e$ $(\!\lvert e \rvert\!) = k$	
$\mathbf{i}_n = \texttt{call}\ e$ $(\!\lvert e \rvert\!) = k$	
$\mathbf{i}_n = \texttt{jcc}\ x$	
Otherwise	

Table 1: Control flow graph extraction

Normalizing mutations. Our CFG representation is a rough abstraction of programs. Indeed we do not make any distinction between the different kinds of sequential instruction, all of them are represented by nodes labeled with the symbol `inst`. This first abstraction level makes the CFG sound with respect to mutations which substitutes instructions with the same behavior. For example the replacement of the instruction `mov eax 0` by the instruction `xor eax eax` does not impact our CFG representation.

However, the soundness with respect to classic mutations techniques remains an important issue. Indeed, some well know mutation techniques can alter the CFG of malicious programs. In order to recover a sound representation of the control flow we apply reductions on CFG. A reduction is defined by a graph rewriting rule. As a case study, we consider three reductions associated to classic mutation techniques. Of course several other reductions can be defined in order to handle more mutations techniques. We use the following reductions

- Concatenate consecutive instructions into blocks of instructions.
- Realign code removing superfluous unconditional jumps.
- Merge consecutive conditional jumps.

Those abstractions can be defined through the graph rewriting rules of Table 2. Figure 2 presents an assembly program and its reduced CFG.

0: **cmp** eax 0
1: **jne** +7
2: **mov** ecx eax
3: **dec** ecx
4: **mul** eax ecx
5: **cmp** ecx 1
6: **jne** −3
7: **jmp** +2
8: **inc** ecx
9: **ret**

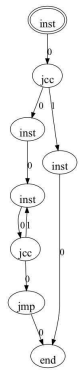


Figure 2: A program and its CFG

We remark that each rewriting rule impose a diminution of the size of the rewritten graph then the reduction clearly terminates. Moreover, since there is no critical pair we have no problem of confluence. Nevertheless, normalizing mutations through rewriting rules is a generic principle that could be applied on sophisticated cases. Then, the issues of termination and confluence shall be carefully considered.

Table 3 presents mutations of the program of Figure 2. All of them have the same reduced CFG as the original program.

2 Efficient database

Road-map. Morphological detection is based on a set of malware CFG which plays the role of malware signatures. This collection of CFG is compiled into a tree automaton thanks to a term representation. Since tree automata fulfill a minimization property, we obtain an efficient representation of the database. Next, we apply this framework for the sub-CFG isomorphism problem in order to detect malware infections.

From graphs to terms. A path is a word over $\{1, 2\}^*$, we write ϵ the empty path. We define the

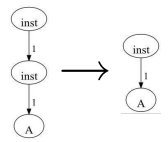
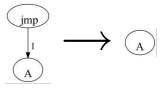
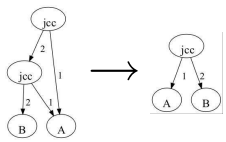
Concatenate instructions	Realign code	Merge jcc
		

Table 2: Control flow graph reductions

Instruction substitution	Block substitution	Block permutation	`jcc` obfuscation	All in one
0: **cmp** eax 0 1: **jne** +7 2: **mov** ecx eax 3: *sub ecx 1* 4: **mul** eax ecx 5: **cmp** ecx 1 6: **jne** −3 7: **jmp** +2 8: *mov eax 1* 9: **ret**	0: **cmp** eax 0 1: **jne** +8 2: *push eax* 3: *pop ecx* 4: **dec** ecx 5: **mul** eax ecx 6: **cmp** ecx 1 7: **jne** −3 8: **jmp** +2 9: **inc** ecx 10: **ret**	0: **cmp** eax 0 1: **jne** +7 2: **mov** ecx eax 3: **dec** ecx 4: **mul** eax ecx 5: **cmp** ecx 1 6: **jne** −3 9: *ret* 8: **inc** ecx 9: *jmp −2*	0: **cmp** eax 0 1: **jne** +9 2: **mov** ecx eax 3: **dec** ecx 4: **mul** eax ecx 5: **cmp** ecx 2 6: *ja −3* 7: **cmp** ecx 1 8: *jne −5* 9: **jmp** +2 10: **inc** ecx 11: **ret**	0: **cmp** eax 0 1: *je +2* 2: *jmp +10* 2: *push eax* 3: *pop ecx* 4: *sub ecx 1* 5: **mul** eax ecx 6: *cmp ecx 2* 7: *ja −3* 8: **cmp** ecx 1 9: *jne −5* 10: *ret* 11: *mov ecx 1* 12: *jmp −2*

Table 3: Control flow graph mutations

path order for any path $\rho, \tau \in \{1,2\}^*$ and any integer $i \in \{1,2\}$ as follows.

$$\rho 1 < \rho 2 \qquad \rho < \rho i \qquad \rho < \tau \Rightarrow \rho\rho' < \tau\tau'$$

A tree domain is a set $d \subset \{1,2\}^*$ such that for any path $\rho \in \{1,2\}^*$ and any integer $i \in \{1,2\}$ we have

$$\rho i \in d \Rightarrow \rho \in d$$

A tree over a set of symbols $\mathbb{F}$ is a pair $t = (d(t), \hat{t})$ where $d(t)$ is a tree domain and $\hat{t}$ is a function from $d(t)$ to $\mathbb{F}$.

We consider the set of symbols $\mathbb{F} = \{\texttt{inst}, \texttt{jmp}, \texttt{call}, \texttt{jcc}, \texttt{ret}\} \cup \{1,2\}^*$ and the trees overs this set. In such trees, a nodes labeled by a path $\rho = \{1,2\}^*$ is thought as pointers to the night node of the tree. Then, a tree have two kinds of nodes: the inner nodes labeled by symbols of $\{\texttt{inst}, \texttt{jmp}, \texttt{call}, \texttt{jcc}, \texttt{ret}\}$ and the pointer nodes labeled by path in $\{1,2\}^\rho$. In the following we write $\mathring{d}(t)$ the set of inner nodes of the tree t, that is

$$\mathring{d}(t) = \left\{ \rho \;\middle|\; \begin{array}{l} \rho \in d(t) \\ \hat{t}(\rho) \in \{\texttt{inst}, \texttt{jmp}, \texttt{call}, \texttt{jcc}, \texttt{ret}\} \end{array} \right\}$$

Next a tree t is well formed if for any paths $\rho, \tau \in d(t)$

$$\big(\hat{t}(\rho) = \tau\big) \Rightarrow \big(\tau \in \mathring{d}(t) \text{ and } \rho \leq \tau\big)$$

We observe that any CFG can be represented by a unique well formed tree.

Tree automata. A *finite tree automaton* is a tuple $\mathcal{A} = (\mathbb{Q}, \mathbb{F}, \mathbb{Q}_f, \Delta)$, where $\mathbb{Q}$ is a finite set of states, $\mathbb{F}$ is a set of symbols, $\mathbb{Q}_f \subset \mathbb{Q}$ is a set of final states and Δ is a finite set of transition rules of the type $a(q_1 \ldots q_i) \rightarrow q$ with $a \in \mathbb{F}$ has arity i and $q, q_1, \ldots, q_i \in \mathbb{Q}$.

A run of an automaton on a tree t starts at the leaves and moves upward, associating a state with

each sub-tree. Any symbol a of arity 0 is labeled by q if $a \to q \in \Delta$. Next, if the direct sub-trees $t_1, \ldots, t_n$ of a tree $t = a(t_1, \ldots, t_n)$ are respectively labeled by states $q_1, \ldots, q_n$ then the tree t is labeled by the state q if $a(q_1, \ldots, q_n) \to q \in \Delta$. A tree t is *accepted* by the automaton if the run labels t with a final state. We observe that a run on a tree t can be computed in linear time, that is $O(n)$ where n is the size of t, that is the number of its nodes.

For any automaton $\mathcal{A}$, we write $\mathcal{L}(\mathcal{A})$ the set of trees accepted by $\mathcal{A}$. A language of trees $\mathbb{L}$ is *recognizable* if there is a tree automaton $\mathcal{A}$ such that $\mathbb{L} = \mathcal{L}(\mathcal{A})$. We define the size $|\mathcal{A}|$ of an automaton $\mathcal{A}$ as the number of its rules.

Tree automata have interesting properties. First, it is easy to build an automaton which recognize a given finite set of trees. This operation can be done in linear time, that is $O(n)$ where n is the sum of the sizes of the trees in the language. Second, we can add new trees to the language recognized by an automaton computing a union of automata, see [10]. Given an automaton $\mathcal{A}$, the union of $\mathcal{A}$ with an automaton $\mathcal{A}'$ can be computed in linear time, that is $O(|\mathcal{A}'|)$.

Finally, for a given recognizable tree language, there exists a unique minimal automaton in the number of states which recognizes this language. This property ensures that the minimal automaton is the best representation of the tree language.

Theorem 1 (From [10])**.** *For any tree automaton $\mathcal{A}$ which recognize a tree language $\mathbb{L}$ we can compute in quadratic time ($O(|\mathcal{A}|^2)$) a tree automaton $\widehat{\mathcal{A}}$ which is the minimum tree automaton recognizing $\mathbb{L}$ up to a renaming of the states.*

Building the database. We explain how this framework can be used to detect malware infections. Suppose that we have a set $\{t_1, \ldots, t_n\}$ of malware CFG represented by trees. Since this set is finite, there is a tree automaton $\mathcal{A}$ which recognizes it.

Next, consider the tree representation t of a given program. Computing a run of $\mathcal{A}$ on t, we can decide in linear time if this tree is one of the the trees obtained from malware CFG. This means that that we can efficiently decide if a program has the same CFG as a known malware.

Finally, we can speed-up the detection computing the minimal automaton which recognize the language $\{t_1, \ldots, t_n\}$. From a practical point of view this is the most efficient representation of the malware CFG database.

Detecting infections. Actually, when a malicious program infects an other program, it includes its own code within the program of its host. Then, we can reasonably suppose that the CFG of the malicious program appears as a subgraph of the global CFG of the infected program. As a result, we can detect such an infection by deciding the subgraph isomorphism problem within the context of CFG.

First we have to observe that we are not confronted with the general sub-graph isomorphism since CFG are graphs with strong constraints. In particular the edge labeling property implies that a CFG composed of n nodes accepts at most n subgraphs. As a result, the sub-CFG isomorphism problem is not NP-complete. Then to detect sub-CFG it is sufficient to run the automaton on the tree representations of any sub-CFG.

3 Experiments

Road-map. We consider the win32 binaries of VX Heavens malware collection [2]. This collection is composed of 10156 malicious programs. Then, we have collected 2653 win32 binaries from a fresh installation of Windows Vista™. This second collection is considered as sane programs.

Using those samples we experiments with our implementation of the morphological detector. We focus our attention on false positive ratios in order to validate the our method. Indeed, we have to know if it is possible to discriminate sane programs from malicious ones only considering their CFG. The following experimental results agree with this hypothesis.

CFG extraction in practice. To overcome the difficulties of CFG extraction we have chosen the following solutions

- In order to deal with crypted and packed samples, we use the unpacking capabilities of ClamAV™ [3].
- We have implemented a dynamic disassembler based on the disassembler library Udis86 [1]. Our module is able to follow the control flow

and it keeps track of the stack in order to remove `push a; ret` sequences.

- The evaluation heuristic $(\!|e|\!)$ proceed as follows. When we encounter a dynamic flow instruction, we emulate the preceding block of sequential instructions in order to recover the value of the expression e. Our emulation technology is also build over Udis86. It is limited to a subset of x86 assembly instruction, interruptions and system calls are not taken into account.

- We reduce the obtained CFG according to the rules of Table 2.

Figure 3 gives the sizes of the reduced CFG extracted from the programs of those collections. On the X axis we have the upper bound on the size of CFG and on the Y axis we have the percentage of CFG whose size is lower than the bound.

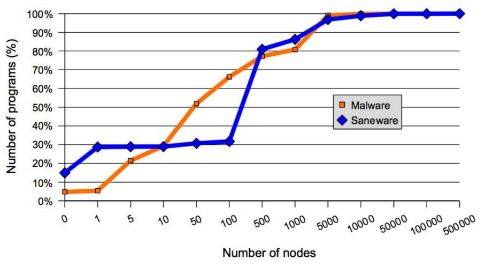


Figure 3: Sizes of control flow graphs

We obeserve that about 5% of the database are programs with a non valid PE header, they produce an empty graph. Then we are able to extract a CFG of more that 15 nodes from about 65% of the samples. The remaining 30% produces a CFG which have between 1 and 15. We think that those graphs are too small to be relevant. We are curruntly working on this part of the samples to improve our extraction procedure.

Building the database. The size of malware control flow graphs clearly impact the accuracy of the control flow detector. We have observed that the graphs extracted from some malware were too small to be relevant and the resulting detector makes many false alerts because of a few such graphs. As a result, we impose a lower bound on the size of the graphs that we include in the database. Next, we have done several tests using different lower bounds.

Let $N \in \mathbb{N}$ be the lower bound on the size of CFG. We build the minimized automaton $\mathcal{A}_{\mathbb{M}}^{N}$ which recognizes the set of tree representations of reduced malware CFG that are composed of more than N nodes. We define the morphological detector $D_{\mathbb{M}}^{N}$ as a predicate such that for any program $\mathbf{p} \in \mathbb{P}$ we have $D_{\mathbb{M}}^{N}(\mathbf{p}) = 1$ if a malware CFG appears as a subgraph of the CFG of $\mathbf{p}$ and $D_{\mathbb{M}}^{N}(\mathbf{p}) = 0$ otherwise. We have seen in the previous sections that $D_{\mathbb{M}}^{N}$ can be decided using $\mathcal{A}_{\mathbb{M}}^{N}$.

This design has several advantages. First, when a new malicious program is discovered, one can easily add its CFG to the database using the union of tree automata and a new compilation to obtain a minimal tree automaton.

The computation of the 'not minimal' automata takes about 25 minutes. The minimization takes several hours but this delay is not so important. Indeed, within the context of an update of the malware database, during the minimization we can release the 'not minimal' automaton. Indeed, even if this is not the best automaton it still recognize the malware database and it could be used until the minimization is terminated.

Evaluation. We are interested in false positives, that is sane programs detected as malicious. For that, we have collected 2653 programs from a fresh installation of `Windows Vista`™. Let us note $\mathbb{S}$ this set of programs. Let $N \in \mathbb{N}$ be a lower bound on the size of malware CFG, we consider the following approximation of the false positives of the detector $D_{\mathbb{M}}^{N}$

$$\text{False positives} \quad \{\mathbf{p} \mid D_{\mathbb{M}}^{N}(\mathbf{p}) = 1 \text{ and } \mathbf{p} \in \mathbb{S}\}$$

We do not evaluate false negatives, that is undetected malicious programs. Indeed, by construction all malicious programs of our malware collection are detected by the morphological detector. Nevertheless, this methods seems promising for this aspect. Indeed, the study [6] has shown that a CFG based detection allows to detect the high-obfuscating computer virus `MetaPHOR` with no false negative.

However, the presence of the lower bound on the size of malware CFG that enter in the database implies that some malware are undetected. Those can

be considered as false negatives, even if they more related to the technical problem of CFG extraction than to the methods of morphological detection. The results of the experiments mention this ratio of undetected malware.

Experimental results. We have built tree automata from the malware samples considering different lower bounds N on the size of CFG. According to the previous section we obtain the morphological detectors $D_{\mathbb{M}}^{N}$. We have tested those detectors on the collection of saneware in order to evaluate the false positives. It takes about 5 h 30 min to analyze the collection of saneware, this represents the analysis of $2'319'294$ sub-CFG. Table 4 presents the results. The first column indicates the considered the lower bound N. The second column indicates the ratio of false positives that is the number of sane programs detected as malicious out of the toalt number of sane programs. The third column indicates the ratio of undetected malicious programs that is the number of malware samples with a CFG size lower than the bound out of the total number of malware samples.

Lower Bound	False positives	Undetected
1	100.00%	4.80%
2	83.78%	5.43%
3	76.82%	16.43%
4	76.77%	16.66%
5	57.98%	20.01%
6	34.84%	21.50%
7	20.57%	23.34%
9	12.06%	24.43%
10	2.17%	26.47%
11	2.04%	27.78%
12	1.60%	29.35%
13	0.71%	30.74%
15	0.09%	36.52%

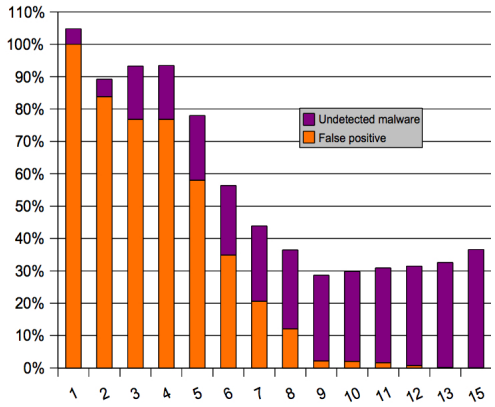


Table 4: Results of the experiments

3.1 Analysis.

As expected, we observe that the false positives decrease with the lower bound on the size of CFG. Over 15 nodes, the CFG seems to be a relevant criterium to discriminate malware.

Concerning the remaining false positives. The libraries `ir41_qc.dll` and `ir41_qcx.dll`, and the malicious program `Trojan.Win32.Sechole` have the same CFG composed of more than $1'000$ nodes. We have tested those programs with commercial antivirus software and the libraries `ir41_qc.dll` and `ir41_qcx.dll` are not detected whereas the program `Trojan.Win32.Sechole` is detected as malicious. The malicious programs seems to be based on the dynamic library and the extraction algorithm was not able to extract the CFG related to the malicious program.

Concerning the ratio of undetected malware, The only way to improve the detector is to implement a better heuristic for control flow graph extraction. In its current version our prototype only use a few heuristics.

For comparison, statistical methods used in [16] induce false negatives ratios between 36 % and 48 % and false positive ratios between 0.5 % and 34 %. A detector based on artificial neural networks developed at IBM [19] presents false negatives ratios between 15 % and 20 % and false positive ratios lower than 1 %. The data mining methods surveyed in [17] present false negatives ratios between 2.3 % and 64.4 % and false positive ratios between 2.2 % and 47.5 %. Heuristics methods from antivirus industry tested in [15] present false negatives ratios between 20.0 % and 48.6 % and false positive ratios lower than 0.2 %.

References

[1] http://udis86.sourceforge.net.

[2] http://vx.netlux.org.

[3] http://www.clamav.net.

[4] Ph Beaucamps and E Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1):3–21, April 2007.

[5] G. Bonfante, M. Kaczmarek, and J.Y. Marion. Control Flow Graphs as Malware Signatures. *WTCV, May*, 2007.

[6] D. Bruschi, Martignoni, L., and M. Monga. Detecting self-mutating malware using control-flow graph matching. Technical report, Università degli Studi di Milano, September 2006.

[7] M. Christodorescu and S. Jha. Testing malware detectors. *ACM SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004.

[8] M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith. Software transformations to improve malware detection. *Journal in Computer Virology*, 3(4):253–265, 2007.

[9] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. *IEEE Symposium on Security and Privacy*, 2005.

[10] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. *Available on: http://www. grappa. univ-lille3. fr/tata*, 10, 1997.

[11] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A Semantics-Based Approach to Malware Detection. In *POPL'07*, 2007.

[12] E. Filiol. *Computer Viruses: from Theory to Applications*. Springer-Verlag, 2005.

[13] E. Filiol. *Advanced viral techniques: mathematical and algorithmic aspects*. Berlin Heidelberg New York: Springer, 2006.

[14] E. Filiol. Malware pattern scanning schemes secure against black-box analysis. In *15th EICAR*, 2006.

[15] D. Gryaznov. Scanners of the Year 2000: Heuristics. *Proceedings of the 5th International Virus Bulletin*, 1999.

[16] J.O. Kephart and W.C. Arnold. Automatic Extraction of Computer Virus Signatures. *4th Virus Bulletin International Conference*, pages 178–184, 1994.

[17] M.G. Schultz, E. Eskin, E. Zadok, and S.J. Stolfo. Data Mining Methods for Detection of New Malicious Executables. *Proceedings of the IEEE Symposium on Security and Privacy*, page 38, 2001.

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

[19] GJ Tesauro, JO Kephart, and GB Sorkin. Neural networks for computer virus recognition. *Expert, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 11(4):5–6, 1996.

[20] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. Normalizing metamorphic malware using term rewriting. *scam*, 0:75–84, 2006.