Morphological Detection of Malware

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

Abstract

In the field of malware detection, method based on syntactical consideration are usually efficient. However, they are strongly vulnerable to obfuscation techniques. This study proposes an efficient construction of a morphological malware detector based on a syntactic and a semantic analysis, technically on control flow graphs of programs (CFG). Our construction employs tree automata techniques to provide an efficient representation of the CFG database. Next, we deal with classic obfuscation of programs by mutation using a generic graph rewriting engine. Finally, we carry out experiments to evaluate the false-positive ratio of the proposed methods.

Introduction

Since there are no obvious technologies to prevent fully and precisely the spreading of viruses and more generally of malwares, the issue of detection cannot be simply thrown out. There are several approach in detection, some of them put the accent on syntactic features, some others on behavioural considerations. Debar, Filiol and Jacob have proposed a taxonomy for detection technologies [12]. Our technique is essentially syntactic, but we take into account some more semantical features of programs.

Generally speaking, string signature based detection uses a database of malware signatures made of regular expressions and a string matching engine to scan files and detect infected ones. There are (at least) three difficulties, which are bound to this kind of detection 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, 15]. Third, as the quantity of malware increases, the ratio of false positive augments. One may remove old viruses from databases, but such a technique lets the door open to new (old) malware.

Thus, a current trend in the community is to design a new generation of malware detectors based on semantical aspects [11, 9] or [17]. However, a major difficulty of these approaches is the efficiency of the detection. Heuristics can be very complex as it is illustrated in the field of computer safety. We will put the accent on these issues.

In [5] , we proposed a new morphological analysis to detect viruses. The idea is to recognize the shape of the control flow graphs of malicious program. Unlike string signature detection, we are not only considering a program as a flat text, but rather as a semantical object, adding in some sense a new dimension to the analysis. To sum up, 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 is based on control flow graphs (CFG) of programs. We use 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. As we see, the design is closed to a string signature based detector, so that both approaches may be combined in a close future. Moreover, it is important to notice that this framework make the signature extraction easier. Indeed, the CFG can be used directly as a witness of the program.

This detection strategy is close to [9, 6]. However, we use a different notion of CFG, technically, we make the an other abstraction of instruction flow graph. Second point, we put our strengths to optimize the efficiency of algorithms, a key point for ”real” applications. For that sake, we use tree automata, a generalization to trees of finite state automata over strings [10]. Here, we transform CFG into trees with, intuitively, 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 Myhill-Nerode construction, 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 we take into account several classical obfuscation methods. 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, 17] where program data flow 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%.

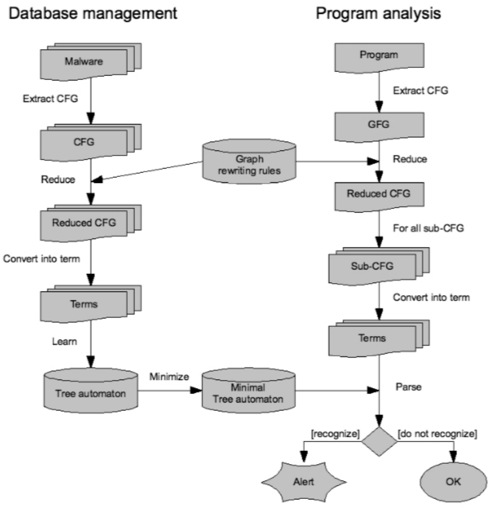


Figure 1: Design of the control flow detector

1 CFG in x86 languages

Roadmap. We consider an assembly language and we explain how its Control Flow Graph (CFG) can be extracted. We underline the difficulties that can be encountered and we outline how they can be overcome with classic techniques. Finally, we present a generic method to normalize mutation of the CFG using graph rewriting rules.

An x86 assembly language. We present the grammar of an assembly language which is close to the x86 assembly language from which we developed our malware detection system.

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	$\mathbb{P} ::= \mathbb{I}^d \mid \mathbb{I}^f \mid \mathbb{P} ; \mathbb{P}$

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. The program entry point is the first instruction $\mathbf{i}_0$.

Prerequisites. The extraction of the CFG from a program is tied to several difficulties. First, since we need access to the instructions of the program, packing and encryption techniques can thwart the extraction. This problem is well known, and classical string signature detectors also suffer from those techniques. Many solutions such as sandboxes 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 [14, 13, 16].

Second, the extraction process should take care of obscure sequences of instructions such as `push a; ret` which has the behavior of the instruction `jmp a`. Again, this is also part of the folklore in the domain and we will suppose that such sequences 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, to build the control flow graph at instruction `jmp eax`, one needs the value of the register `eax` in order to follow the control flow transfer. In such cases, our current procedure rely on a heuristic $\llparenthesis e \rrparenthesis$ which provides the value of the expression e by static analysis. If the value cannot be computed then $\llparenthesis e \rrparenthesis = \bot$. Such an heuristic can be based on partial evaluation, emulation or any other static analysis technique.

The extraction procedure. We suppose that we have access to the code of programs and that we have an heuristic $\llparenthesis\ \rrparenthesis$ to evaluate expressions. Table 1 presents a procedure to abstract the control flow of programs. If an expression cannot be evaluated then the extraction yields an `end` node. The entry point of the program correspond to the root of the CFG. We remark that a CFG is a rooted directed graph with ordered successors as a result any CFG can be represented by a flow graph.

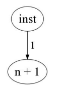
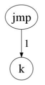
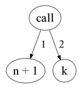
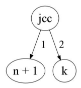
Instruction	Graph
$\mathbf{i}_n \in \mathbb{I}^d$	
$\mathbf{i}_n = \texttt{jmp}\ e$ $\llparenthesis e \rrparenthesis = k$	
$\mathbf{i}_n = \texttt{call}\ e$ $\llparenthesis e \rrparenthesis = k$	
$\mathbf{i}_n = \texttt{jcc}\ x$	
Otherwise	

Table 1: Control flow graph extraction

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

We make the CFG even more sound with respect to classic mutation techniques considering other abstractions.

- 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. From now on $\mathcal{CFG}(\mathbf{p})$ denotes the flow graph which correspond to the reduced CFG of the program **p**. 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
```

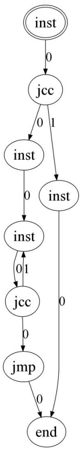


Figure 2: A program and its CFG

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 mutation through rewriting rules is a generic principle that could be applied on sophisticated cases. Then, the issues of termination and confluence will be carefully considered.

2 Efficient database

Roadmap. 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 infection.

From graphs to terms. A path is words over $\{1,2\}^*$, we write ϵ the empty path. We define the path order for any path $\rho, \tau \in \{1,2\}^*$ and any integer $i \in \{1,2\}$

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

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

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

A tree over a set of symbol $\mathbb{F}$ 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 $\mathbb{F}$.

From now on, let the set of symbols be $\mathbb{F} = \{\mathtt{inst}, \mathtt{jmp}, \mathtt{call}, \mathtt{jcc}, \mathtt{ret}\} \cup \{1,2\}^*$. In the corresponding trees, a node labelled by word/path $\rho = \{1,2\}^*$ is thought of as a pointer to the corresponding node of the tree. Then, a tree have two kind of nodes: the inner nodes labelled by symbols of $\{\mathtt{inst}, \mathtt{jmp}, \mathtt{call}, \mathtt{jcc}, \mathtt{ret}\}$ and the pointer nodes labelled 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

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

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

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

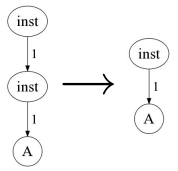
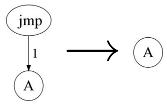
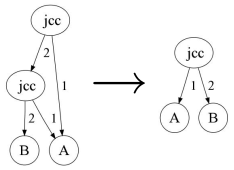
Concatenate instructions	Realign code	Merge jcc
		 jcc 2 1 jcc 2 1 B A jcc 1 2 A B

Table 2: Control flow graph reductions

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 $a(q_1 \dots q_i) \rightarrow q$ with $a \in \mathbb{C}$ has arity i and $q, q_1, \dots, q_i \in \mathbb{Q}$.

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 labelled by q if $a \rightarrow q \in \Delta$. Next, if the direct sub-trees $t_1, \dots, t_n$ of a tree $t = a(t_1, \dots, t_n)$ are respectively labelled by states $q_1, \dots, q_n$ then the tree t is labelled by the state q if $a(q_1, \dots, q_n) \rightarrow q \in \Delta$. A tree t is *accepted* by the automaton if the run labels the root node of t with a final state. We observe that a run on a tree t can be computed in linear time, that is $O(|t|)$.

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 by means of tree automata.

Theorem 1 (From [10])**.** *For any tree automaton $\mathcal{A}$ which recognizes 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 $\{t_1, \dots, t_n\}$ 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 have the same CFG as a known malware.

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

Detecting infections. Actually, when a malicious programs 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 sub-graph of the global CFG of the infected program. As a result, we can detect such an infection by deciding the sub-graph isomorphism problem within the context of CFG.

So, our problem is a classical problem of subgraph isomorphism property, a property which is NP-complete in general. However, due to the fact that the successor relation is ordered, in the present terms, the problem is polynomial. Indeed a CFG composed of n vertices has only n distinct sub-CFG of at most n vertices[1]. Then to detect sub-CFG it is sufficient to run the automaton on the tree representations of all the sub-CFG.

3 Experiments

Roadmap. We consider the win32 binaries of VX Heavens [2] malware collection. This collection is composed of 10156 malicious programs. Then, we have collected 2653 win32 binaries from a fresh installation of Windows Vista$^{\text{TM}}$. 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 the CFG extraction we have chosen the following solutions

- We use partly the unpacking procedure of ClamAV$^{\text{TM}}$ [3].
- We have implemented a dynamic disassembler based on the disassembler library Udis86 [1].
- We reduce the obtained CFG according to the rules of Table 2.

Figure 3 presents the result of the CFG extraction from the malware database. 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 5 nodes from about 80% of the program of the database. The remaining 15% produce a CFG which have between 1 and 5. We think that those graphs are too small to be relevant.

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.

Evaluation. As said above we dispose of a collection of 10156 malicious programs and 2653 sane programs. Figure 3 gives the sizes of the reduced CFG extracted form 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.

We are interested by false positives, that is sane programs detected as malicious. For that, we have collected 2653 programs from a fresh installation of `Windows Vista`$^{\text{TM}}$. 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

[1] Take care that we speak about sub-CFG, and not sub-graph. Otherwise, the result is incorrect.

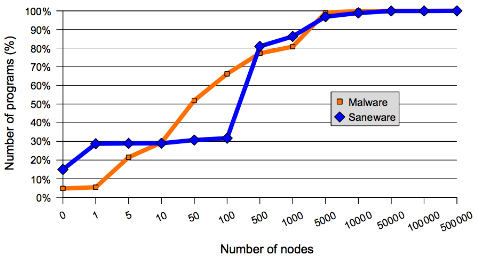


Figure 3: Sizes of control flow graphs

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.

Building the database. 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 malware CFG. 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 sub-graph of $\mathcal{CFG}(\mathbf{p})$ 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 the canonical tree of 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.

Experimental results. We have built tree automata from the malware sample. 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 analyse the collection of saneware, this represents the analysis of $2'319'294$ sub-CFG. Table 3 presents the results. The first column indicates the considered detector according to the lower bound N. The second column indicates the number of false negatives, those are malicious programs whose CFG have sizes lower than the bound. The ratio is computed with respect to the whole database of 10156 malicious programs. The last column indicates the number of false positives and the ratio with respect to the collection of 2653 sane programs.

4 Conclusion

In this paper, we wanted to show that issues about the efficiency of detectors based on semantical features could be overwhelmed. From that point of view, the result we got are somewhat promising, and we still work on it. But now, our main issue is to get a better evaluation of the precision of our system. Indeed, the use of Vista distribution as saneware witnesses may provoke a bias in our analysis. We are currently working on this issue.

References

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

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%

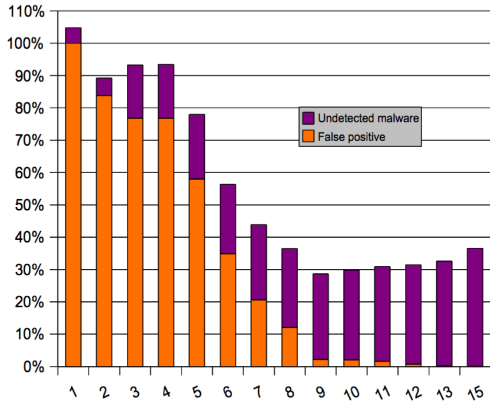


Table 3: Results of the experiments

[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] H. Debar, E. Filiol, and Jacob. G. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 2008.

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

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

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

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

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