Control Flow to Detect Malware

Guillaume Bonfante, Matthieu Kaczmarek, Jean-Yves Marion

Control Flow to Detect Malware

Guillaume Bonfante, Matthieu Kaczmarek and Jean-Yves Marion

Nancy-Université - INPL - Ecole Nationale Supérieure des Mines de Nancy - Loria
B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France

Abstract

This study proposes a malware detection strategy based on control flow. It consists in searching in the control flow graph of the analysed program for an induced sub-graph which corresponds to the control flow graphs of a malicious program. The resulting detector is build over a strong theoretical framework.Finally, experiments are carried out in order to evaluates the proposed detection strategy.

Introduction

Malware detection aims at detecting malicious programs or programs which have been infected by malicious programs. Up to now, anti-malware software have satisfied with syntactic detection based on signature matching engines [14]. However, recent advances in code obfuscation [10, 2] have suggested that this detection strategy is likely to be overtaken by modern mutation techniques.

The term 'malicious' refers to an undesirable behavior [8], this implies that malware detection has to decide semantic aspects of programs. Unfortunately, most of semantic properties are not decidable, then malware detection face computability limits [1, 5]. Nervertheless, recent studies have proposed detection strategies based on decidable semantic aspects: execution trace [7], control flow [4, 3] or data flow [16]. Such strategies are able to handle most of known mutation techniques.

We interest in a detection strategy based on control flow graphs. More precisely, we show how flow graphs can be used as signatures. This is the first step of a broader semantics based approach to detection. This preleminary study follows three steps. First, the Section 1 recalls basics about control flow graphs in `assembly x86` and it describes how to compute such a graph. Then, the Section 2 formally defines the control flow graph detector and its design. Finally, the Section 3 exposes the protocol to evaluate a malware detector and it provides experimental results about the previously defined detector.

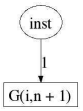
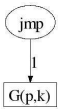
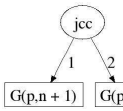
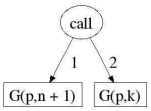
$\mathbf{i}_n$	$\mathbf{i}_n \in \mathbb{I}^d$	$\mathbf{i}_n = \texttt{jmp}\ e$ $(\!\lvert e \rvert\!) = k$	$\mathbf{i}_n = \texttt{jcc}\ e$ $(\!\lvert e \rvert\!) = k$	$\mathbf{i}_n = \texttt{call}\ e$ $(\!\lvert e \rvert\!) = k$	Otherwise
					

Table 1: Control flow graph extraction

1 Control flow graphs

Programming language. We consider a subset of the `assembly x86` defined by the following grammar

$$
\begin{array}{rll}
\text{Expressions :} & \mathbb{E} & := \mathbb{N} \mid \texttt{eax} \mid \ldots \mid [\mathbb{E}] \\
\text{Flow instructions :} & \mathbb{I}^f & := \texttt{jmp}\ \mathbb{E} \mid \texttt{jcc}\ \mathbb{E} \mid \texttt{call}\ \mathbb{E} \mid \texttt{ret} \\
\text{Data instructions :} & \mathbb{I}^d & := \texttt{mov}\ \mathbb{E}\ \mathbb{E} \mid \texttt{comp}\ \mathbb{E}\ \mathbb{E} \mid \ldots \\
\text{Instructions :} & \mathbb{I} & := \mathbb{I}^f \mid \mathbb{I}^d \\
\text{Programs :} & \mathbb{P} & := \mathbb{I} \mid \mathbb{P}\,;\mathbb{I}
\end{array}
$$

A program is a sequence of instructions $\mathbf{p} = \mathbf{i}_1 ; \ldots ; \mathbf{i}_n$ and we suppose that $\mathbf{i}_1$ is the first instruction to be executed. For any instruction $\mathbf{i}_k$, we say that k is its address and we write $|\mathbf{p}| = n$ the length of $\mathbf{p}$.

The control flow is driven by the flow instructions of $\mathbb{I}^f$. We describe the intuition behind the different possible flow instructions: we suppose that the current instruction is at the address k.

- An unconditional jump `jmp` e, moves the flow to the value of e.
- A conditional jump `jcc` e moves the flow to the value of$e$ if its associated condition is validated otherwise it continues on the address $k + 1$.
- A function call `call` e, pushes the address $k + 1$ and moves the flow to the the value of e.
- A function return `ret` pops an address and moves the flow to this address.

Extraction of the CFG. The *control flow graph (CFG)* represents all paths that might be traversed through a program during its execution.

The extraction of the exact CFG of an `assembly x86` program is not computable in general. This problem can easily be reduced to the halting problem: consider a program $\mathbf{p}$ which takes as arguments an other program $\mathbf{q}$ such that $\mathbf{p}$ executes $\mathbf{q}$ and it jumps to the address computed by $\mathbf{q}$. The CFG of $\mathbf{p}$ can be extracted if and only if $\mathbf{q}$ terminates.

We conclude that only an approximation of the CFG can be computed. We define a procedure G to build such an approximation. It is described by the Table 1 where for any expression $e \in \mathbb{E}$, $(\!\lvert e \rvert\!)$ is an oracle which return the value of e when it can be evaluated. For any program $\mathbf{p}$ we obtain its CFG by computing $G(\mathbf{p}, 1)$.

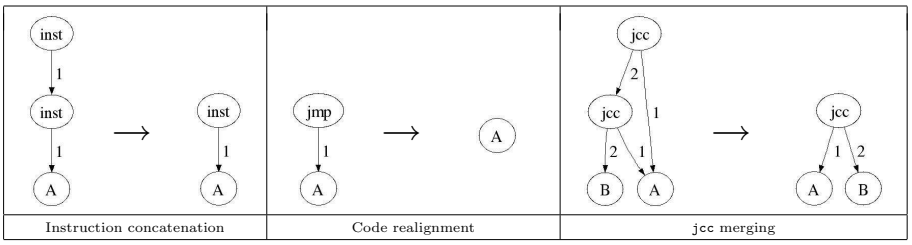


Table 2: Control flow graph reduction

```
1: call 3
2: jmp 9
3: mov eax, 1
4: mov ecx, 8
5: mul ecx
6: dec ecx
7: cmp ecx, 0
8: jne 5
9: ret
```

```
1: call 3
2: jmp 10
3: push 1
4: pop eax
5: mov ecx, 9
6: mul ecx
7: dec ecx
8: cmp ecx, 0
9: jne 6
10: ret
```

```
1: jmp 5
2: mul ecx
3: dec ecx
4: jmp 10
5: call 7
6: jmp 12
7: mov eax, 1
8: mov ecx, 8
9: jmp 2
10: cmp ecx, 0
11: jne 2
12: ret
```

```
1: call 3
2: jmp 10
3: mov eax, 1
4: mov ecx, 8
5: mul ecx
6: dec ecx
7: cmp ecx, 0
8: ja 5
9: jne 5
10: ret
```

Figure 1: Left to right, a code fragment, a mutation of data instructions, a code permutation, a mutation of conditional jumps.

Reductions. Some classic obfuscation techniques are able to mutate the CFG. To handle those mutations we apply the reductions defined in the Table 2. The ideas behind those graph rewriting rules are the following.

- Concatenate consecutive instruction vertex, to handle mutations which change the number of contiguous data instructions.
- Realign code removing the linking jumps, to handle code permutation.
- Merge consecutive conditional jumps.

The Figure 1 provides examples of mutated programs which are semantically equivalent but which have different CFG. However, all those programs have the same reduced CFG. It is worth to mention that the presented mutations are used by the computer virus `MetaPHOR` [9].

2 The detection strategy.

The CFG detector. Following the formalism of [7], a *detector* is a predicate over $2^{\mathbb{P}} \times \mathbb{P}$. We say that a detector D *detects* a program $\mathbf{p} \in \mathbb{P}$ on a set of malware $\mathbb{M} \subset \mathbb{P}$ if $D(\mathbb{M}, \mathbf{p})$ is true.

We define $\mathbf{p} \mapsto \bar{\mathbf{p}}$ as the application which maps any program $\mathbf{p}$ to its reduced CFG. We interest in the particular class of detectors $D_n(\mathbb{M}, \mathbf{p})$ defined as

$$\exists \mathbf{m} \in \mathbb{M} : |\bar{\mathbf{m}}| \geq n \text{ and } \bar{\mathbf{m}} \text{ is an induced sub-graph of } \bar{\mathbf{p}}$$

Where $|\bar{\mathbf{m}}|$ denotes the number of nodes in the graph $\bar{\mathbf{m}}$.

The intuition is that the detector searches in the CFG of the analysed program for an induced sub-graph which corresponds to the CFG of a malicious program.

The parameter n in D_n allows to handle malicious programs which have a too small CFG to be relevant. In the following, we experiments with different lower bound on the size of CFG. The intuition is that the greater n is, the more false negatives there are and the less false positives there are.

The design. Automata theory have proved to be a sound theoretical framework for the classic string signature detection. An equivalent base for the CFG detector would be beneficial. Unfortunately, there is no commonly accepted framework for graph automata then we fall back upon tree automata. We establish the following protocol to design the detector.

- Define a one-one mapping $g \mapsto \hat{g}$ from graphs to trees.
- Compile the trees $\{\hat{\mathbf{m}} \mid \mathbf{m} \in \mathbb{M}\}$ into a tree automata.
- Run the automaton on each sub-graph of $\bar{\mathbf{p}}$.

It is worth to metion that this design works only if $\{\hat{\mathbf{m}} \mid \mathbf{m} \in \mathbb{M}\}$ is a regular set. In practice $\mathbb{M}$ is finite, then the associated set of CFG is regular.

The resulting detector is built on strong theoretical basis and it uses classic techniques [6] to minimize the representation of the database and to speed-up the detection. More details about this design will be given in a forthcoming paper.

3 Evaluation

Evaluation of a detector. A detector can do two kinds of errors: false negatives or false positives. A *false negative* is an undetected malicious program. In other terms it is a program $\mathbf{m}$ known as malicious and such that $D(\mathbb{M}, \mathbf{m}) = 0$. Conversely, a *false positive* is a sane program $\mathbf{p}$ detected as malicious, that is $D(\mathbb{M}, \mathbf{p}) = 1$.

The problem of this definition is that the notions of malicious and sane are ubiquious. As a result, detectors are evaluated experimentally considering samples of malware and sane programs. The protocol is the following. We take a set of malware $\mathbb{M}$ and a set of sane programs $\mathbb{S}$, $\mathbb{M}$ and $\mathbb{S}$ are suposed disjoint. Then, we compute the false negatives defined by the Equation (1) and the false positives defined by the Equation (2), with respect to $\mathbb{M}$ and $\mathbb{S}$.

$$\{\mathbf{m} \mid D(\mathbb{M}, \mathbf{m}) = 0 \text{ and } \mathbf{m} \in \mathbb{M}\} \tag{1}$$

$$\{\mathbf{p} \mid D(\mathbb{M}, \mathbf{p}) = 1 \text{ and } \mathbf{p} \in \mathbb{S}\} \tag{2}$$

The challenge is to minimize the false negatives and the false positives for extensive sets of malware and sane programs.

Experiments. We have implemented and evaluated this detector on a sample of 10156 malicious programs collected from public sources and 2653 programs taken from a fresh installation of `Windows Vista`™. The Table 3 presents the results. The first column indicates the instance of n in D_n, that is the lower bound imposed on the size of CFG. 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.

Detector	False negatives	Ratio	False positives	Ratio
D_1	493	4.9 %	2617	98.6 %
D_2	526	5.2 %	2197	82.8 %
D_3	1233	12.1 %	2189	82.5 %
D_4	1290	12.7 %	2151	81.1 %
D_5	1331	13.1 %	2145	80.9 %
D_6	1573	15.5 %	2107	79.4 %
D_7	1635	16.1 %	796	30.0 %
D_8	1679	16.5 %	754	28.4 %
D_9	1755	17.3 %	461	17.4 %
D_{10}	1842	18.1 %	378	14.2 %
D_{11}	1942	19.1 %	332	12.5 %
D_{12}	2009	19.8 %	73	2.7 %
D_{13}	2080	20.5 %	38	1.4 %
D_{14}	2171	21.4 %	31	1.2 %
D_{15}	2258	22.2 %	19	0.7 %
D_{18}	2580	24.6 %	13	0.5 %
D_{19}	2635	25.9 %	3	0.1 %

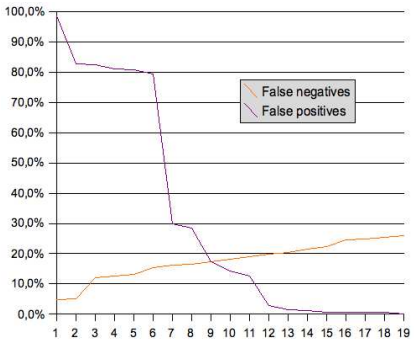


Table 3: Results of the experiments

Analysis. As expected, we observe that the false negatives and the false positives respectively increase and decrease with the lower bound on the size of CFG. The increase in false negatives is moderate wheras there are clear treshold on the decrease in false positives. Over 18 nodes, the CFG seems to be a relevant criterium to descriminate malware.

Concerning the three remaining false positives with more than 19 nodes. The library `sqlwid.dll` and the malicious program `Trojan-Proxy.Win32.Agent.x` have the same CFG composed of 81 nodes. The libraries `ir41_qc.dll` and `ir41_qcx.dll`, and the malicious program `Trojan.Win32.Sechole` have the same CFG composed of 1226 nodes. A preliminaries study of the machine code have shown that the programs are mostly identical but we are still investigating.

For comparison, statistical methods used in [12] 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 [15] presents false negatives ratios between 15 % and 20 % and false positive ratios lower than 1 %. The data mining methods surveyed in [13] 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 [11] present false negatives ratios between 20.0 % and 48.6 % false positive ratios lower than 0.2 %.

References

[1] L. Adleman. An abstract theory of computer viruses. In *Advances in Cryptology – CRYPTO'88*, volume 403. Lecture Notes in Computer Science, 1988.

[2] P. 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.

[3] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Control flow graphs as malware signatures. In *WTCV*, May 2007.

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

[5] F. Cohen. Computational aspects of computer viruses. *Computers and Security*, 8:325–344, 1989.

[6] 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.

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

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

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

[10] E. Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology*, 2(1):35–50, 2006.

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

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

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

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

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

[16] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. *scam*, 0:75–84, 2006.