Detection of Self-Mutating Computer Viruses

Gilbert Notoatmodjo
gnot002@ec.auckland.ac.nz

Department of Computer Science
University of Auckland, New Zealand

Abstract

This paper presents self-mutating computer viruses as an emerging threat, along with a brief background on obfuscation transformations which are used by virus writers to achieve self-mutation and virus detection as a practical defense. A new static analysis approach which was claimed by the authors, Christodorescu and Jha to be resilient against obfuscated viruses is evaluated, along with a discussion about several problems that were found during an investigation of their experiment. Finally, a future experiment that can be used to determine the viability of their approach is proposed and discussed.

1. Introduction

As people are becoming increasingly dependent on computers, threats on computer systems are exerting more serious problems. Nowadays, computer virus is considered one of the threats that cause devastating effects on computer systems. Fred Cohen, the first person to introduce the word *computer virus*, defined it as “a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself [2]”. The most interesting aspect in Cohen’s definition is that he predicted the possibility of self-mutating viruses long before their existence.

Self-mutation is a technique that is recently used by virus writers to make their viruses more difficult to detect by antivirus software. Before further discussing about self-mutating viruses, it would be helpful to understand how viruses develop. According to Subramanya et al [6], viruses can be classified into five categories based on their generations; each generation incorporates new features and improvements from the previous generation, making them even more difficult to detect.

- *First generation viruses* are the most basic form of viruses. They self replicate, but do not attempt to hide their presence from the system. They can easily be recognized by the noticeable increase in the size of infected files due to re-infection of already infected files.

- *Second generation viruses* use unique signatures to prevent re-infection of a file, which causes unnecessary growth of the infected file and might lead to early detection.

- *Third generation viruses* use stealth techniques to counteract virus-scanning attempts. These viruses intercept the requests to perform selected system service call interrupts. If the operation exposes the presence of the virus, the operation is redirected to return false information.

- *Fourth generation viruses* use simple armoring techniques such as encryption (normally symmetric) [8] and simple obfuscation transformations to avoid detection. In the case where they are encrypted, a decryption routine is usually placed before the encrypted virus body.

- *Fifth generation viruses* are self-mutating viruses. They infect other systems with modified versions of themselves. These viruses can be further classified into two categories, which are:

    - *Polymorphic viruses* are capable of changing its encryption key and generating new instances of their decryption routines with simple obfuscation transformations, such as nop-insertion, code transposition and register reassignment. Antivirus software deals with these viruses by using heuristic analyses, emulation, and sometimes by adding additional features to its signature scanning method, such as wildcards and regular expressions [1].

- *Metamorphic viruses* do not have a decryption routine or a constant virus body, but are able to create new versions that look different by using more complex obfuscation techniques. This is considered the most advanced technique used by virus writers to date [1, 6].

2. Background on Obfuscation Transformations

As previously mentioned, some viruses, particularly fourth and fifth generation viruses use obfuscation techniques to avoid detection. Obfuscation transforms a program so that it becomes more complex to understand (or detect, in case of viruses), while maintaining the function of the original version [3, 5]. Christodorescu and Jha mentioned several obfuscation transformations that are commonly used by virus writers [1], which are:

- *Dead-Code Insertion* adds irrelevant instructions and/or variables (sometimes referred as trash or junk) to a program without affecting its behavior. The resilience of this type of transformation against automatic deobfuscation and overhead cost to implement such transformation depends on the quality of the irrelevant code itself and the nesting depth at which the irrelevant code is inserted [3]. The simplest example and probably the most commonly used trash used by virus writers is nop [1, 8].

- *Code Transposition* scrambles the placement of instructions in the program, so that the order of execution is different form the order of instructions assumed in the binary image. According to Collberg et al, this type of transformation has strong resilience against automatic deobfuscation, and is inexpensive in terms of overhead cost [3].

- *Register Reassignment* replaces the usage of a register with another register in a specific live range, and thus substitutes the register names. This transformation is also known as 'Change Variable Lifetime' [3]. It is interesting to note that while Christodorescu and Jha stated that this transformation offers no real obfuscatory value [1], Collberg et al claimed that although this transformation might fail to confuse human readers, it offers high resilience against automatic deobfuscation [3].

- *Instruction Substitution* replaces instruction sequences in the program with other instruction sequences without changing the functionality of the program based on a dictionary of equivalent instruction sequences. This transformation is similar to 'Table Interpretation' which provides strong resilience against automatic deobfuscation with expensive overhead cost when implemented in high level language such as Java, where additional virtual machines are required [3]. In the case of IA-32 assembly language, this method is even more powerful, because the instruction set is rich enough to provide several ways of performing the same operation [1] without requiring additional virtual machines.

Considering the size of ordinary virus source codes, these transformations might fail to confuse human readers after all, but are sufficient to confuse antivirus analyses that rely upon basic signature-based scanning.

3. Virus Detection

Although isolating our systems seems to be the ultimate defense against computer viruses, complete isolation is impossible in most cases. This situation left us with virus detection as the most practical defense against computer viruses. Virus detection can be classified into three major categories, namely static, dynamic and heuristic analyses [6, 8, 10]. This section will discuss the three main categories, along with some commonly used techniques that fall into each category.

3.1. Static Analysis

Static analysis detects viruses by analyzing the virus codes or infected files and matching them with a set of records [10]. In order for a static analysis to be effective, it has to regularly maintain and update its records. Signature-based scanning is one of the most commonly used techniques.

Signature-based scanning

Signature based scanning maintains a database of sequences of bytes that were extracted from known viruses, but not likely to be found in benign programs. It looks for patterns that match the records in its database in the file system. According to Szor [8], there are enhancements that are sometimes added to basic signature-based scanning, such as:

- *Wildcards and regular expressions* offer better detection of instruction groups. Some encrypted viruses and even polymorphic viruses can easily be detected with this additional feature.

- *Smart scanning* was first introduced to deal with virus mutating kits, which often employ dead code insertion transformation. Smart scanning can skip irrelevant codes such as nop. It can also select an area of the virus body that had no references to data or other subroutines to deal with transformations like code transposition and register reassignment.

- *Cryptographic detection* is useful against encrypted viruses, which often use symmetric encryption that is rather weak and relatively easy to attack. The virus code can be easily detected after being decrypted.

3.2. Dynamic Analysis

Dynamic analysis attempts to detect viruses as they are executed. Code emulation and behavioral analysis are some of the commonly used dynamic analysis methods.

Code emulation

Code emulation implements a virtual machine to mimic the execution of infected files to determine malicious intent [6, 10]. Although this method is quite powerful, it tends to consume enormous amount of system resources [6, 8].

Behavioral analysis

Behavioral analysis monitors the execution of files in real-time environments, and gives the user an opportunity to prevent or undo the actions that might be performed by viruses [6]. This approach has a tendency to slow down system performance [10] and often generates false alarms, which might frustrate end users [6].

3.3. Heuristic Analysis

Heuristic analysis attempts to detect unknown viruses using both static and dynamic analyses. Instead of searching for specific types of viruses, it generalizes the viruses into families, and marks a program as suspicious if it appears to be a closely related variant of known virus families. [6, 8] Although this approach seems to be a potential solution for polymorphic and unknown viruses [1, 8], it is known to have a fairly high false positive rate.

3.4. Discussion

“Not all the techniques can be applied to all computer viruses...It is enough to have an arsenal of techniques, one of which will be a good solution to block, detect or disinfect a particular computer virus. [8]” Szor stated his opinion that it is almost impossible to create an ideal solution that deals with all computer viruses. Cohen also proposed a similar idea that the virus detection problem is undecidable, he mentioned, “several potential countermeasures were examined in some depth, and none appear to offer ideal solutions. [2]” In the case of self-mutating viruses, Szor’s point is rather precise, because each self-mutating virus is unique and has its own weaknesses. Nobody has found a single technique that can be used effectively against all self-mutating viruses.

But apparently, Szor's opinion faces an opposition from "…security professionals and researchers, who might argue that if one technique cannot be used all the time, it is completely ineffective. [8]" Does such ideal technique exist? As security professionals and researchers wage their war against virus writers, existence of such technique remains as axiomatic as the 'Holy Grail'. Thus, the question that we should ask is not "does it exist?" but rather "can it exist?"

4. A New Breed of Static Analysis: The 'Holy Grail'?

Christodorescu and Jha proposed a new static analysis approach [1]. The prototype implementation of this approach is called SAFE (Static Analyzer For Executables). They claimed that SAFE is resilient to common obfuscation transformations, and based on their experiment, SAFE was able to detect several obfuscated versions of viruses that three commercial antivirus software (Norton, McAfee, Command) failed to detect, with zero false negative and false positive rates.

SAFE's detection method is based on a library of abstract representations of the viruses, referred to as 'malicious code automata'. It first transforms a suspected executables into its internal representations in the form of CFG (Control Flow Graph), annotates the CFG with a set of abstraction patterns, to produce an annotated CFG which includes information which indicates where a particular abstraction pattern was found in the program. The annotated CFG is then compared to the abstract representations of viruses. If the pattern described by the abstract representation matches the annotated CFG, it declares that the executable is infected. SAFE utilizes two third party tools, IDA Pro, a commercial interactive disassembler, and CodeSurfer, which provides an API to perform a variety of static analysis.

SAFE's only drawback might be its rather sluggish execution time. Based on their experiment result, scan on a 1 MB benign executable (QuickTimePlayer.exe) took approximately 16 minutes. This doesn’t include the processing time taken by the third party tools (IDA Pro and CodeSurfer), which are incorporated into SAFE.

Apart from the execution time drawback, the result of their experiment demonstrates that SAFE seems to be a promising solution against self-mutating viruses, a tool which is able to detect self-evolving viruses with zero false negative and false positive rates and outperforms three commercial antivirus softwares in terms of accuracy. If their experiment was well conducted and their result was reliable, SAFE could well be the ideal solution whose existence is questioned by Cohen [2] and Szor [8].

However, my investigation suggests that there are some problems with their experiment design and result presentation. Those problems will be discussed in the following section.

5. Investigation of Christodorescu and Jha’s Experiment

5.1. Experiment Description

In their experiment, 4 viruses were used to determine the accuracy of their tool, SAFE, compared to three commercial virus scanners (Norton, Command and McAfee). The viruses that were used are:

- Chernobyl (CIH)

  This virus infects 32 bit Windows 95/98/ME. When a user runs an infected program, the virus will become resident in memory and will try to execute its payload, which destroys data. The second payload will then try to corrupt Flash BIOS, causing permanent damage to the computer. [12]

- zombie-6.b

  I was unable to find a description of this virus apart from the description that was given by Christodorescu and Jha [1], “[zombie-6.b] includes an interesting feature – the polymorphic engine hides every piece of the virus, and the virus code is added to the infected file as a chain of differently-sized routines, making standard signature detection techniques almost useless."

- f0sf0r0

  This virus uses a polymorphic engine and entry point obscuring to hide itself infected files. When an infected file is executed, the virus searches for portable executable files and infects them. According to Virus Library [11], "[f0sf0r0] has bugs, and infected files often become corrupted while infecting. When run, they cause a standard Windows message about an error in application."

- Hare

  This virus infects executable files (COM and EXE), as well as the MBR (Master Boot Record) of the hard drive and bootloader sectors of floppy disks. The virus has a strange way in running its polymorphic routines, which causes polymorphic decryption loops of all files that were infected on the same computer to contain the same data. [1, 4]

They obfuscated these viruses with their own obfuscation tool, which employs four obfuscation transformations that were described in section 2, namely dead-code insertion, code transposition, register reassignment and instruction substitution. Their obfuscation tool is mentioned to be similar to Mistfall, a polymorphic engine written by a Russian virus writer, Zombie. They then tried to detect both obfuscated and unobfuscated versions of the four viruses using the commercial antivirus softwares and SAFE.

They also conducted a small test to determine the false positive of SAFE by trying to scan different versions of each virus with a malicious code automaton corresponding to a different virus. They scanned four benign programs with malicious code automaton corresponding to all four viruses. The false positive rate was found to be zero.

5.2. First Flaw: Result Presentation

They stated their result as follows, "a combination of nop-insertion and code transposition was enough to create obfuscated versions of the viruses that the commercial virus scanners could not detect. Moreover, the Norton antivirus software could not detect an obfuscated version of the Chernobyl virus using just nop-insertions. SAFE was resistant to the two obfuscation transformations. [1]" The result in tabular form is presented below.

		Norton® Antivirus 7.0	McAfee® VirusScan 6.01	Command® Antivirus 4.61.2	SAFE
Chernobyl	original	✓	✓	✓	✓
	obfuscated	✗[1]	✗[1,2]	✗[1,2]	✓
z0mbie-6.b	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓
f0sf0r0	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓
Hare	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓

Obfuscations considered: [1] = nop-insertion (a form of dead-code insertion)
[2] = code transposition

Table 1: Result of Christodorescu and Jha's experiment. [1]

Note that although they mentioned Norton antivirus software failed to detect a version of Chernobyl, which was obfuscated by using only nop-insertion, they did not mention anything about how the other antivirus softwares performed against the viruses that were obfuscated using only nop-insertion. Furthermore, although they stated that SAFE was resistant to the two obfuscation transformation, they did not mention whether these transformations were used separately or in conjunction with each other. The missing data are noted with question marks in the table below:

			Norton Antivirus 7.0	McAfee VirusScan 6.01	Command Antivirus 4.61.2	SAFE
Chernobyl	Original		√	√	√	√
	Obfuscated	NOP insertion	x	?	?	?
		Code transposition	?	?	?	?
		Combination	?	x	x	?
Zombie-6.b	Original		√	√	√	√
	Obfuscated	NOP insertion	?	?	?	?
		Code transposition	?	?	?	?
		Combination	x	x	x	?
f0sf0r0	Original		√	√	√	√
	Obfuscated	NOP insertion	?	?	?	?
		Code transposition	?	?	?	?
		Combination	x	x	x	?
Hare	Original		√	√	√	√
	Obfuscated	NOP insertion	?	?	?	?
		Code transposition	?	?	?	?
		Combination	x	x	x	?

Table 2: Missing data interpretation. Where did all the ticks and crosses go? [after 1]

They did not describe how SAFE performed against register reassignment and instruction substitution, two transformations which they also regarded as common. More importantly, they failed to address the versions of the antivirus software that were used in the experiment. At this point, their result became questionable.

5.3. Second Flaw: Experiment Design

As described above, three of the four viruses that were used in the experiment are polymorphic viruses, which have their own polymorphic routines, along with their bugs and weaknesses. Antivirus software sometimes attempts to identify the viruses based on their weaknesses. Further obfuscation of these polymorphic viruses with a new tool will result in new generations, which are unknown to most antivirus software. In the case of the non-polymorphic virus, obfuscation will result in a new generation, which might be unexpected from a virus that was previously known as non-polymorphic.

On the other hand, SAFE, which relies on a library of malicious code automata, was provided with all automata of the four viruses used in the experiment. In this scenario, it would seem unreasonable to compare the accuracy of the three commercial antivirus software against SAFE's accuracy.

5.4. Discussion

Given that some underlying variables of the experiment and some parts of the resulting data were not clearly described, it would be too early to conclude that SAFE in fact outperforms the three commercial antivirus softwares in terms of accuracy, and is completely resilient to the four obfuscation transformations.

6. Further Studies: An Experiment Proposal

6.1. Mistfall

Christodorescu and Jha used their own obfuscation tool in their experiment, which they described as being similar to Mistfall engine. The original Mistfall was written in Borland C++ by Zombie, a Russian virus writer. This section will discuss the features of Mistfall and one of the viruses that utilize this engine, W95/ZMist, which will be used in the proposed experiment.

Mistfall integrates the virus code to the infected executables by first decompiling them with per-instruction basis. After that, it modifies the code by inserting the virus code into the executables, re-assembles the executable, writes the file to disk, and recalculates the checksum of the executable. To do this operation, Mistfall requires at least 32MB of system memory. One of the possible infection cases will have JMP instructions inserted after every instruction, which points to the next instruction, with the virus code inserted in between. [13, 14]

One of the viruses that utilize this engine is W95/ZMist, which was also written by Zombie. ZMist has all the features from Mistfall. It is able to merge with the executable code, and become a part of the instruction flow. When the virus receives control, it will launch the host program and hide its own process until the infection routine is complete. ZMist only infects Portable Executables with MZ extensions, which is smaller than 448 KB in size. [8]

ZMist can also mutate itself, creating new generations by using obfuscation transformations, such as dead-code insertion, and instruction substitution (reversing branch conditions, replacing MOV instructions with PUSH/POP sequences, alternative opcode encoding, exchanging XOR/SUB and OR/TEST instructions). The mutation is only done once per infection of a computer. [8] According to Symantec [7] and McAfee [9], their products are able to detect W95/ZMist virus.

6.2. Proposed Experiment

In this section, I would like to propose an experiment, which will help to clarify the ambiguity in Christodorescu and Jha's experiment, as discussed in section 5. The main purpose of the proposed experiment is to assess the practicality of SAFE against several obfuscation transformations, in terms of accuracy and detection time. In this experiment, the aforementioned W95/ZMist virus will be used. Two commercial antivirus softwares (Norton and McAfee), which are claimed to be able to detect W95/ZMist will also be used as the control group.

The steps needed to perform the experiment are described below:

1. Implement Christodorescu and Jha's approach.
   Since they did not release the prototype that was used in their experiment, the approach will need to be implemented as close as possible to their description.

2. Obtain an unobfuscated version of W95/ZMist virus, and create a malicious code automaton based on this version.

3. Infect executables in a few machines with W95/ZMist virus.

The reason of using more than one machine is because the virus only creates a new generation once per machine, and more than one generation would be needed to test SAFE’s accuracy against obfuscated viruses.

These machines will be loaded with a number of portable executables that match the condition for Zmist infection. A record of the checksum and size of these executables will be recorded; a backup copy will also be stored for reference. The infection will be done by executing an executable, which has been purposely infected by ZMist virus on the machines. The machines that are used in the experiment will need to be isolated during the infection process and re-formatted immediately.

4. Extract the infected executables.

   The executables from the previous step will be extracted and checked for infection by comparing their checksum and file size with the corresponding records, and with the backup copy, if necessary.

5. Attempt to detect the viruses.

   In this step, a computer, which has Norton and McAfee installed (at least the versions that are confirmed to be able to detect W95/Zmist) will be used. The SAFE implementation will also be installed in this machine along with the malicious code automaton of W95/ZMist, which was obtained in step 2. After that, these three tools (Norton, McAfee, and SAFE) will be used to detect the infected executables that were obtained in steps 3 and 4. At each trial, execution time and detection accuracy of all three tools will be recorded. The records will then be used to assess SAFE’s performance in terms of accuracy and detection time.

Executables		Norton		McAfee		SAFE	
		Detected	Time	Detected	Time	Detected	Time
Machine A	Executable 1						
	Executable 2						
	Executable 3						
	…						
Machine B	Executable 1						
	Executable 2						
	Executable 3						
	…						
Machine C	Executable 1						
	Executable 2						
	Executable 3						
	…						

Table 2: The format of how the data are recorded in the proposed experiment.

7. Final Discussion and Conclusion

The proposed experiment is not intended to determine the real false positive rate of SAFE. As mentioned above, Christodorescu and Jha conducted a small test to determine the false positive rate of SAFE, which was found to be zero. In my opinion, their claim for zero false positive rate in their experiment does not reflect the real false positive rate of SAFE due to insufficient sample size. In static analysis, the false positive rate greatly depends on the number of records used. As the number of records increases, the chance of getting false positive also increases.

A possible way to determine the real false positive of SAFE would be to obtain a list of all viruses that can be detected by several commercial antivirus softwares, create an automaton for each of the viruses, and try to scan several commonly used benign programs with the library of automata. However, enormous amount of time and effort will be needed to perform this study.

There are several issues that might be encountered during the implementation step of the proposed experiment. Firstly, since their description is more intended as a formal description rather than practical implementation, we will not be able to create an exact implementation based on their description. Secondly, this part of the experiment might also be time and resource consuming. Further measurement studies will be required to obtain a more precise approximation of time and effort needed to do the implementation. Lastly, in order for the experiment to be significant, the implementation will also have to be tested to ensure whether it can detect viruses with the same degree of accuracy as their prototype, and thus the implementation will need to be tested against all the viruses used in their experiment.

Personally, I believe that although this experiment might show their approach has a high accuracy in detecting obfuscated viruses, the execution time might be large compared to the commercial antivirus software. This experiment may only contribute to Szor's and Cohen's earlier proposition that no single ideal solution exists. This fact questions the value of time and resources needed to evaluate Christodorescu and Jha's solution, as with the same effort, one could formulate an approach which is as practical if not better.

Acknowledgement

The author would like to first and foremost thank Clark Thomborson for providing his advice, expertise, and constructive review on the earlier drafts of this paper. The author would like to acknowledge Poernomo Notoatmodjo, for raising the author's interest in computer security. The author would also like to thank Angela Halim, Arthur Iskandar, and Jason Chen for their comments, support and positive reviews. This paper was written as a requirement of Software Security (COMPSCI 725 S2C) course held by the Department of Computer Science, University of Auckland, New Zealand.

References

[1] Christodorescu, M., Jha, S. Static analysis of executables to detect malicious patterns. *Proceedings of the 12th USENIX Security Symp., Washington, DC. August 2003.*

[2] Cohen, F. Computer Viruses - Theory and Experiments. *Proceedings of IFIPTC11, Computers and Security, pp. 22-35, 1987.* - http://all.net/books /virus/index.html (Access date: 20 Oct. 2005).

[3] Collberg, C., Thomborson, C., Low, D. A taxonomy of obfuscating transformations. *Technical Report #148.* Department of Computer Science, University of Auckland, 1997.

[4] Hare.7610. Virus Library - http://www.viruslibrary.com/details.htm?id=2924 (Access Date: 20 Oct. 2005).

[5] Low, D. Protecting Java code via code obfuscation. *ACM Crossroads, Volume 4, Issue 3, spring 1998, pp. 21 – 23.*

[6] Subramanya, S.R., Lakshminarasimhan, N. Computer Viruses. *IEEE Potentials Volume 20, Issue 4, Oct-Nov 2001, pp.16 – 19.*

[7] Symantec Security Response - http://smallbiz.symantec.com /avcenter/ venc/ auto/ index/indexW.html (Access Date: 20 Oct. 2005).

[8] Szor, P. The Art of Computer Research and Defense. Addison Wesley, 2005.

[9] W32/Zmist.gen, McAfee Virus Information Library - http://vil.mcafeesecurity.com /vil/content/v_99382.html (Access Date: 20 Oct. 2005).

[10] Wang, J.H., Deng, P.S., Fan, Y., Jaw, L., Liu, Y. Virus detection using data mining techniques. *Proceedings of IEEE 37th Annual 2003 International Carnahan Conference on14-16 Oct. 2003 pp.71 – 76*

[11] Win32.Fosforo. Virus Library - http://www.viruslibrary.com/ virusinfo /Win32. Fosforo.html (Access Date: 20 Oct. 2005).

[12] Yamamura, M. Write-up on W95.CIH, Symmantec Security Response http://www.symantec.com/avcenter/venc/data/cih.html (Access Date: 20 Oct. 2005).

[13] Z0mbie, Automated reverse engineering: Mistfall engine. 2000. Translated from Russian in 2001- http://vx.netlux.org/lib/vzo21.html(Access Date: 20 Oct.2005).

[14] Z0mbie, Mistfall source code - http://vx.netlux.org/vx.php?id=em23 (Access Date: 20 Oct. 2005).