2010 International Conference on Broadband, Wireless Computing, Communication and Applications

Malware Obfuscation Techniques: A Brief Survey

Ilsun You
School of Information Science
Korean Bible University
Seoul, Republic of Korea
isyou@bible.ac.kr

Kangbin Yim
Dept. of Information Security Engineering
Soonchunhyang University
Asan, Republic of Korea
yim@sch.ac.kr

2010 International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA) | 978-1-4244-8448-5/10/$31.00 ©2010 IEEE | DOI: 10.1109/BWCCA.2010.85

***Abstract*—As the obfuscation is widely used by malware writers to evade antivirus scanners, so it becomes important to analyze how this technique is applied to malwares. This paper explores the malware obfuscation techniques while reviewing the encrypted, oligomorphic, polymorphic and metamorphic malwares which are able to avoid detection. Moreover, we discuss the future trends on the malware obfuscation techniques.**

Keywords-Malware, Obfuscation, Metamorphic, Polymorphic

I. INTRODUCTION

The obfuscation is a technique that makes programs harder to understand [1]. For such a purpose, it converts a program to a new different version while making them functionally equal to each other. Originally, this technology aimed at protecting the intellectual property of software developers, but it has been broadly used by malware authors to elude detection [2]-[6]. That is, in order to evade antivirus scanners, malwares evolve their body into new generations through the obfuscation technique. Clearly, it is important to analyze the obfuscation techniques to efficiently address malwares.

In this paper, we explore the malware obfuscation techniques. For this goal, we firstly overview the history of the malwares that have been developed to defeat signature-based antivirus scanners. Then, the malware obfuscation techniques are introduced with examples. The example code is extracted from Win95/Zmist and reversed on the debugger OllyDBG [6]. Also, we discuss the future trends on the malware obfuscation techniques while focusing on the web and smartphone malwares.

This paper is organized as follows. In section 2, we describe the encrypted, oligomorphic, polymorphic and metamorphic malwares. Section 3 explores the obfuscation techniques commonly used by polymorphic and metamorphic malwares, and then section 4 discusses the future trends. Finally, we conclude in section 5.

II. ENCRYPTED, OLIGOMORPHIC, POLYMORPHIC AND METAMORPHIC MALWARES

A. *Encryped Malware*

The first approach to evade the signature based antivirus scanners is to use encryption [2]-[4]. In this approach, an encrypted malware is typically composed of the decryptor and the encrypted main body. The decryptor recovers the main body whenever the infected file is run. For each infection, by using a different key, the malware makes the encrypted part unique, thus hiding its signature. However, the main problem of this approach is that the decryptor remains constant from generation to generation. That makes it possible for the antivirus scanners to detect this kind of malwares based on the descriptor's code pattern.

B. *Oligomorphic and Polimorphic Malwares*

In order to address the shortcoming of the encrypted malwares, malware authors devised technologies, through which malwares can mutate their decryptor from one generation to the next [2]-[4]. The first attempt was the oligomorphic malware capable of changing its decryptor slightly [2][4]. However, this malware can generate at most a few hundreds of different decryptors, thus still being able to be detected with signatures. For overcoming the limitation, the malware authors developed the polymorphic malware [2][4]. The polymorphic malware achieves to create countless number of distinct decryptors with the help of the obfuscation methods including dead-code insertion, register reassignment, and so forth [4]-[6]. Especially, due to the powerful toolkits such as "*The Mutation Engine* (*MtE*)" [2], it was a critical problem. The toolkits help the malware writers to easily convert their non-obfuscated malware into the polymorphic version. Even though the polymorphic malwares can effectively thwart the signature matching, their constant body, which appears after decryption, can be used as an important source for detection. In order to exploit this vulnerability, antivirus tools adopt the emulation technique [2][3]. Through this technique, the tools execute a malware in an emulator (called "*Sandbox*") without resulting in any harm. Once the constant body is loaded into memory after decrypted, the conventional detection, *i.e.*, signature based, can be applied. In order to detect and prevent such emulation, the polymorphic malwares used the armoring technique [2]. However, as the antivirus scanners became matured, they were capable of addressing this technique, thus effectively defeating the polymorphic malwares.

C. *Metamorphic Malware*

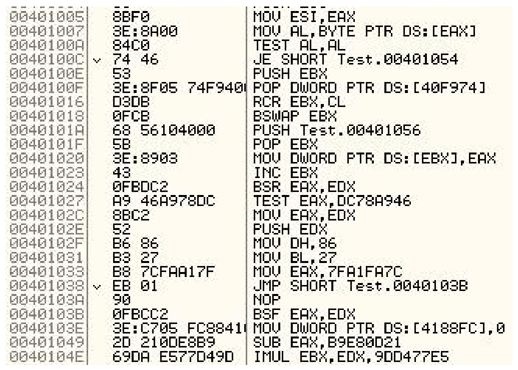
The metamorphic malware was proposed as a novel approach beyond the oligomorphic and polimorphic ones [3]-[6]. Note that this malware makes best use of obfuscation techniques to evolve its body into new generations, which look different but work essentially the same. For such an evolution, it should be able to recognize, parse and mutate its own body whenever it propagates. It is important that the metamorphic malware never reveals its constant body in memory due to not using encryption or packing. That makes it so difficult for the antivirus scanners to detect this malware.

III. OBFUSCATION TECHNIQUES

This section introduces the obfuscation techniques commonly used in the polimorphic and metamorphic malware.

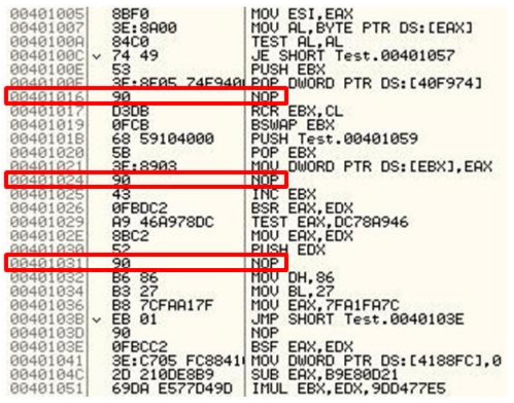
A. *Dead-Code Insertion*

Dead-code insertion is a simple technique that adds some ineffective instructions to a program to change its appearance, but keep its behavior [1][4][6]. An example of such instructions is nop. Figures 1 and 2 show the original code is easily obfuscated through insertion of nop instructions. However, the signature based antivirus scanners can defeat this technique by just deleting the ineffective instructions prior to analysis. Consequently, in order to make detection more difficult, some code sequences were presented as illustrated in Figure 3.



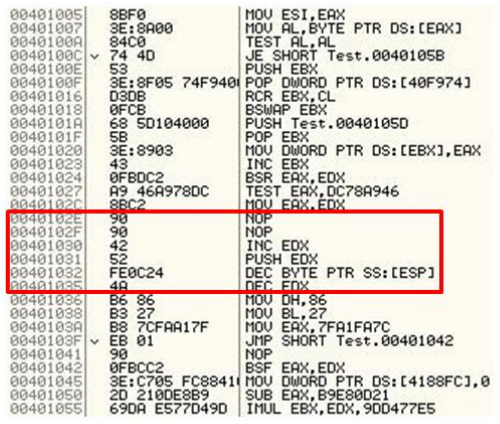
```
00401005    8BF0            MOV ESI,EAX
00401007    3E:8A00         MOV AL,BYTE PTR DS:[EAX]
0040100A    84C0            TEST AL,AL
0040100C  v 74 46           JE SHORT Test.00401054
0040100E    53              PUSH EBX
0040100F    3E:8F05 74F940  POP DWORD PTR DS:[40F974]
00401016    D3DB            RCR EBX,CL
00401018    0FCB            BSWAP EBX
0040101A    68 56104000     PUSH Test.00401056
0040101F    5B              POP EBX
00401020    3E:8903         MOV DWORD PTR DS:[EBX],EAX
00401023    43              INC EBX
00401024    0FBDC2          BSR EAX,EDX
00401027    A9 46A978DC     TEST EAX,DC78A946
0040102C    8BC2            MOV EAX,EDX
0040102E    52              PUSH EDX
0040102F    B6 86           MOV DH,86
00401031    B3 27           MOV BL,27
00401033    B8 7CFAA17F     MOV EAX,7FA1FA7C
00401038  v EB 01           JMP SHORT Test.0040103B
0040103A    90              NOP
0040103B    0FBCC2          BSF EAX,EDX
0040103E    3E:C705 FC8841  MOV DWORD PTR DS:[4188FC],0
00401049    2D 210DE8B9     SUB EAX,B9E80D21
0040104E    69DA E577D49D   IMUL EBX,EDX,9D477E5
```

Figure 1. A Sample Code



```
00401005    8BF0            MOV ESI,EAX
00401007    3E:8A00         MOV AL,BYTE PTR DS:[EAX]
0040100A    84C0            TEST AL,AL
0040100C  v 74 49           JE SHORT Test.00401057
0040100E    53              PUSH EBX
0040100F    3E:8F05 74F940  POP DWORD PTR DS:[40F974]
00401016    90              NOP
00401017    D3DB            RCR EBX,CL
00401019    0FCB            BSWAP EBX
0040101B    68 59104000     PUSH Test.00401059
00401020    5B              POP EBX
00401021    3E:8903         MOV DWORD PTR DS:[EBX],EAX
00401024    90              NOP
00401025    43              INC EBX
00401026    0FBDC2          BSR EAX,EDX
00401029    A9 46A978DC     TEST EAX,DC78A946
0040102E    8BC2            MOV EAX,EDX
00401030    52              PUSH EDX
00401031    90              NOP
00401032    B6 86           MOV DH,86
00401034    B3 27           MOV BL,27
00401036    B8 7CFAA17F     MOV EAX,7FA1FA7C
0040103B  v EB 01           JMP SHORT Test.0040103E
0040103D    90              NOP
0040103E    0FBCC2          BSF EAX,EDX
00401041    3E:C705 FC8841  MOV DWORD PTR DS:[4188FC],0
0040104C    2D 210DE8B9     SUB EAX,B9E80D21
00401051    69DA E577D49D   IMUL EBX,EDX,9D477E5
```

Figure 2. Dead –Code Insertion
(The original code is shown in Figure 1)

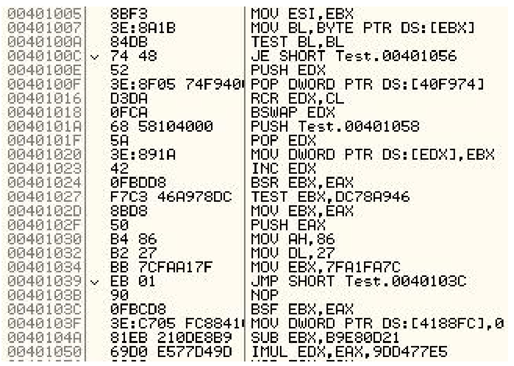


```
00401005    8BF0            MOV ESI,EAX
00401007    3E:8A00         MOV AL,BYTE PTR DS:[EAX]
0040100A    84C0            TEST AL,AL
0040100C  v 74 4D           JE SHORT Test.0040105B
0040100E    53              PUSH EBX
0040100F    3E:8F05 74F940  POP DWORD PTR DS:[40F974]
00401016    D3DB            RCR EBX,CL
00401018    0FCB            BSWAP EBX
0040101A    68 5D104000     PUSH Test.0040105D
0040101F    5B              POP EBX
00401020    3E:8903         MOV DWORD PTR DS:[EBX],EAX
00401023    43              INC EBX
00401024    0FBDC2          BSR EAX,EDX
00401027    A9 46A978DC     TEST EAX,DC78A946
0040102C    8BC2            MOV EAX,EDX
0040102E    90              NOP
0040102F    90              NOP
00401030    42              INC EDX
00401031    52              PUSH EDX
00401032    FE0C24          DEC BYTE PTR SS:[ESP]
00401035    4A              DEC EDX
00401036    B6 86           MOV DH,86
00401038    B3 27           MOV BL,27
0040103A    B8 7CFAA17F     MOV EAX,7FA1FA7C
0040103F  v EB 01           JMP SHORT Test.00401042
00401041    90              NOP
00401042    0FBCC2          BSF EAX,EDX
00401045    3E:C705 FC8841  MOV DWORD PTR DS:[4188FC],0
00401050    2D 210DE8B9     SUB EAX,B9E80D21
00401055    69DA E577D49D   IMUL EBX,EDX,9D477E5
```

Figure 3. Ineffective Code Sequences

B. *Register Reassignment*

Register reassignment is another simple technique that switches registers from generation to generation while keeping the program code and its behavior same [4][6]. Figure 4 describes how this technique is applied. In this example, the original code shown in Figure 1 is evolved by switching the registers. (used by Win95/Regswap virus [4])



```
00401005    8BF3            MOV ESI,EBX
00401007    3E:8A1B         MOV BL,BYTE PTR DS:[EBX]
0040100A    84DB            TEST BL,BL
0040100C  v 74 48           JE SHORT Test.00401056
0040100E    52              PUSH EDX
0040100F    3E:8F05 74F940  POP DWORD PTR DS:[40F974]
00401016    D3DA            RCR EDX,CL
00401018    0FCA            BSWAP EDX
0040101A    68 58104000     PUSH Test.00401058
0040101F    5A              POP EDX
00401020    3E:891A         MOV DWORD PTR DS:[EDX],EBX
00401023    42              INC EDX
00401024    0FBDD8          BSR EBX,EAX
00401027    F7C3 46A978DC   TEST EBX,DC78A946
0040102D    8BD8            MOV EBX,EAX
0040102F    50              PUSH EAX
00401030    B4 86           MOV AH,86
00401032    B2 27           MOV DL,27
00401034    BB 7CFAA17F     MOV EBX,7FA1FA7C
00401039  v EB 01           JMP SHORT Test.0040103C
0040103B    90              NOP
0040103C    0FBCD8          BSF EBX,EAX
0040103F    3E:C705 FC8841  MOV DWORD PTR DS:[4188FC],0
0040104A    81EB 210DE8B9   SUB EBX,B9E80D21
00401050    69D0 E577D49D   IMUL EDX,EAX,9D477E5
```

Figure 4. Register Reassignment
(The original code is shown in Figure 1 and registers EAX, EBX and EDX are reassigned to EBX, EDX and EAX respectively)

Note that the wildcard searching can make this technique useless.

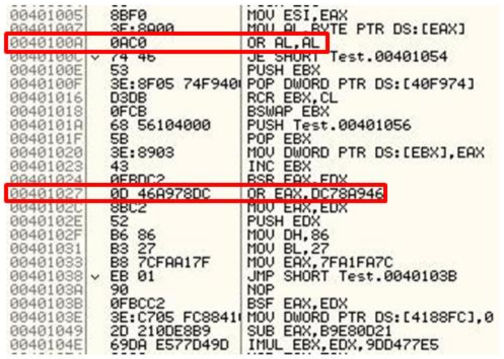
C. *Subroutine Reordering*

Subroutine reordering obfuscates an original code by changing the order of its subroutines in a random way [4]. This technique can generate n! different variants, where n is the number of subroutines. For example, Win32/Ghost had ten subroutines, leading to 10! = 3628800 different generations [4].

D. *Instruction Substitution*

Instruction substitution evolves an original code by replacing some instructions with other equivalent ones [6].

For example, xor can be replaced with sub and mov can be replaced with push/pop as shown in Figure 5.



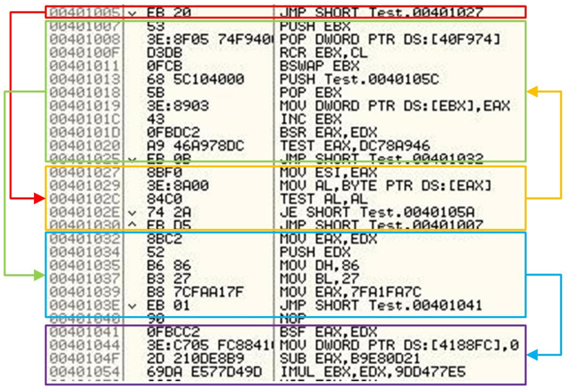
```
00401005    8BF0            MOV ESI,EAX
00401007    3E:8A00         MOV AL,BYTE PTR DS:[EAX]
0040100A    0AC0            OR AL,AL
0040100C  v 74 46           JE SHORT Test.00401054
0040100E    53              PUSH EBX
0040100F    3E:8F05 74F940( POP DWORD PTR DS:[40F974]
00401016    D3DB            RCR EBX,CL
00401018    0FCB            BSWAP EBX
0040101A    68 56104000     PUSH Test.00401056
0040101F    5B              POP EBX
00401020    3E:8903         MOV DWORD PTR DS:[EBX],EAX
00401023    43              INC EBX
00401024    0FBDC2          BSR EAX,EDX
00401027    0D 46A978DC     OR EAX,DC78A946
0040102C    8BC2            MOV EAX,EDX
0040102E    52              PUSH EDX
0040102F    B6 86           MOV DH,86
00401031    B3 27           MOV BL,27
00401033    B8 7CFAA17F     MOV EAX,7FA1FA7C
00401038  v EB 01           JMP SHORT Test.0040103B
0040103A    90              NOP
0040103B    0FBCC2          BSF EAX,EDX
0040103E    3E:C705 FC8841( MOV DWORD PTR DS:[4188FC],0
00401049    2D 210DE8B9     SUB EAX,B9E80D21
0040104E    69DA E577D49D   IMUL EBX,EDX,9DD477E5
```

Figure 5. Instruction Substitution
(The original code is shown in Figure 1)

Note that this technique can effectively change the code with a library of equivalent instructions.

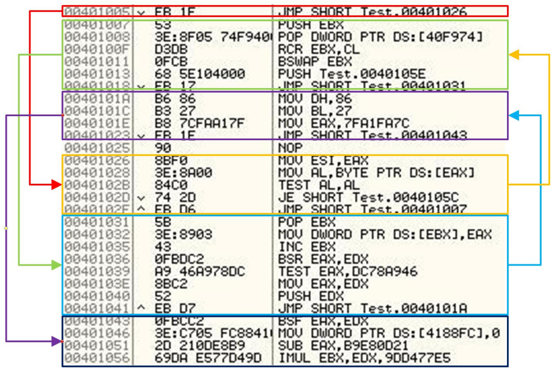
E. *Code Transposition*

Code transposition reorders the sequence of the instructions of an original code without having any impact on its behavior [5]. There are two methods to achieve this technique.



```
00401005  v EB 20           JMP SHORT Test.00401027
00401007    53              PUSH EBX
00401008    3E:8F05 74F940( POP DWORD PTR DS:[40F974]
0040100F    D3DB            RCR EBX,CL
00401011    0FCB            BSWAP EBX
00401013    68 5C104000     PUSH Test.0040105C
00401018    5B              POP EBX
00401019    3E:8903         MOV DWORD PTR DS:[EBX],EAX
0040101C    43              INC EBX
0040101D    0FBDC2          BSR EAX,EDX
00401020    A9 46A978DC     TEST EAX,DC78A946
00401025  v EB 0B           JMP SHORT Test.00401032
00401027    8BF0            MOV ESI,EAX
00401029    3E:8A00         MOV AL,BYTE PTR DS:[EAX]
0040102C    84C0            TEST AL,AL
0040102E  v 74 2A           JE SHORT Test.0040105A
00401030  ^ EB D5           JMP SHORT Test.00401007
00401032    8BC2            MOV EAX,EDX
00401034    52              PUSH EDX
00401035    B6 86           MOV DH,86
00401037    B3 27           MOV BL,27
00401039    B8 7CFAA17F     MOV EAX,7FA1FA7C
0040103E  v EB 01           JMP SHORT Test.00401041
00401040    90              NOP
00401041    0FBCC2          BSF EAX,EDX
00401044    3E:C705 FC8841( MOV DWORD PTR DS:[4188FC],0
0040104F    2D 210DE8B9     SUB EAX,B9E80D21
00401054    69DA E577D49D   IMUL EBX,EDX,9DD477E5
```

Figure 6. Code Transposition based on Unconditional Branches



```
00401005  v EB 1F           JMP SHORT Test.00401026
00401007    53              PUSH EBX
00401008    3E:8F05 74F940( POP DWORD PTR DS:[40F974]
0040100F    D3DB            RCR EBX,CL
00401011    0FCB            BSWAP EBX
00401013    68 5E104000     PUSH Test.0040105E
00401018  v EB 17           JMP SHORT Test.00401031
0040101A    B6 86           MOV DH,86
0040101C    B3 27           MOV BL,27
0040101E    B8 7CFAA17F     MOV EAX,7FA1FA7C
00401023  v EB 1E           JMP SHORT Test.00401043
00401025    90              NOP
00401026    8BF0            MOV ESI,EAX
00401028    3E:8A00         MOV AL,BYTE PTR DS:[EAX]
0040102B    84C0            TEST AL,AL
0040102D  v 74 2D           JE SHORT Test.0040105C
0040102F  ^ EB D6           JMP SHORT Test.00401007
00401031    5B              POP EBX
00401032    3E:8903         MOV DWORD PTR DS:[EBX],EAX
00401035    43              INC EBX
00401036    0FBDC2          BSR EAX,EDX
00401039    A9 46A978DC     TEST EAX,DC78A946
0040103E    8BC2            MOV EAX,EDX
00401040    52              PUSH EDX
00401041  ^ EB D7           JMP SHORT Test.0040101A
00401043    0FBCC2          BSF EAX,EDX
00401046    3E:C705 FC8841( MOV DWORD PTR DS:[4188FC],0
00401051    2D 210DE8B9     SUB EAX,B9E80D21
00401056    69DA E577D49D   IMUL EBX,EDX,9DD477E5
```

Figure 7. Code Transposition based on Independent Instructions

The first method, which is demonstrated in Figure 6, randomly shuffles the instructions, and then recovers the original execution order by inserting the unconditional branches or jumps. Clearly, it is not difficult to defeat this method because the original program can be easily restored by removing the unconditional branches or jumps. On the other hand, the second method creates new generations by choosing and reordering the independent instructions that have no impact on one another. Because it is a complex problem to find the independent instructions, this method is hard to implement, but can make the cost of detection high. Figure 7 shows an example of this method.

F. *Code Integration*

In code integration, introduced by the Win95/Zmist malware (called Zmist), a malware knits itself to the code of its target program [6]. In order to apply this technique, Zmist firstly decompile its target program into manageable objects, seamlessly adds itself between them, and reassembles the integrated code into a new generation. As one of the most sophisticated obfuscation techniques, code integration can make detection and recovery so difficult.

IV. Future Trends

As shown in the advanced malwares such as Zmist, the malware obfuscation technologies have become sophisticated and complex. Clearly, such a tendency is expected to be retained based on the growth of the hardware and software technologies. Also, they will be revised to be suit for the popular infrastructures such as web and smartphone.

In this section, we describe the future trends in the malware obfuscation techniques while focusing on web and smartphone malwares.

A. *Web Malware*

Due to the abundance and popularity of web applications, web malwares have considerably increased, thus being the main security threats nowadays [7]. It is natural that the authors of web malwares apply the obfuscation technologies to make it so difficult for their malware to be analyzed. Note that web malwares are generally distributed by exploiting web browsers' vulnerabilities and malicious (or compromised) websites. Thus, the current obfuscation technologies will be revised for such exploitation and adapted to web environment. Especially, the obfuscation technologies for the malicious JavaScript will be continually presented and sophisticated because JavaScript is mainly used as a vehicle for malware distribution [7]. For example, a new web malware, called "JS_VIRTOOL", was recently found [8]-[10]. In order to make analysis difficult, the malware uses a code obfuscation method where the malware body for each infected page is encrypted with a unique secret key derived from that page's URL. Thus, it is impossible to recover the encrypted malware body without knowing the original URL.

B. *Smartphone Malware*

Smartphones have gained considerable success, but been the most attractive target for malware writers [11]-[14]. Thus, it is not surprising that the smartphone malwares are being increased. For instance, recently, an iPhone malware named "Rickrolling" was found [13]. Once successfully infecting a device, the malware silently sends the device owner's privacy information such as e-mail, contacts, SMSs, calendars, photos and so forth to its host machine.

It is obvious that the obfuscation technologies will be actively considered and applied for these malwares. We expect that in addition to just using the current obfuscation technologies, malware authors will develop new ones that are not only energy and resource efficient, but also appropriate for their target platform such as iPhone or Android.

C. *Virtual Machine-based Malware*

One of the most difficult issues to be solved by malware writers is to hide the behavior of the extractor and the plain body of the malwares after extraction. Instructions to be fetched for execution need to be loaded first in the primary memory as they are designed based on the Von Neumann architecture. This means that a thorough dynamic analysis on the dedicated memory region can give hints to understanding the body [18][19].

Emulating multiple personalized virtual processors has been considered as an ultimate solution to hiding a plain code body between small groups of researchers. For malwares in this approach, the code body is reprogrammed or recompiled into world-unique instructions prior to release. To understand the behavior of the body, analyzers need to understand the unknown architecture and accordant unknown code of the selected virtual processor and program.

This job requires too much overhead because the executing context of the native code in the emulator is really far from that of the original unknown code that the emulator interprets. Especially, the instruction sets can be selected randomly. Even though the analyzers completely understand the functionality of the code after several days or weeks, the code will be already updated for another unknown virtual processor. Several examples for the virtual machine obfuscation can be found in documents and practical applications recently [15]-[17].

V. Conclusions

In this paper, we briefly surveyed the malware obfuscation technologies such as dead-code insertion, register reassignment, subroutine reordering, instruction substitution, code transposition and code integration, which have been mainly used by polymorphic and metamorphic malwares to evade antivirus scanners. As a future trend, these obfuscation techniques will be more sophisticated and complex while being combined with one another. Especially, these obfuscation techniques will be revised to be appropriate for the web and smartphone malwares.

References

[1] A. Balakrishnan and C. Schulze, "Code Obfuscation Literature Survey," http://pages.cs.wisc.edu/~arinib/writeup.pdf, 2005.

[2] M. Schiffman, "A Brief History of Malware Obfuscation: Part 1 of 2 ," *http://blogs.cisco.com/security*, Feb. 2010.

[3] M. Schiffman, "A Brief History of Malware Obfuscation: Part 2 of 2 ," *http://blogs.cisco.com/security*, Feb. 2010.

[4] W. Wong and M. Stamp, "Hunting for Metamorphic Engines," Journal in Computer Virology, vol. 2, no. 3, pp. 211-229, Dec. 2006.

[5] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," Proceedings of the 12th conference on USENIX Security Symposium, Vol. 1, pp. 169-186, Aug. 2003.

[6] E. Konstantinou, "Metamorphic Virus: Analysis and Detection," RHUL-MA-2008-02, Technical Report of University of London, Jan. 2008. http://www.rhul.ac.uk/mathematics/techreports

[7] R. Chakraborty, "Increase in Web Malware Activity," http://maliciousbrains.blogspot.com/2009/11/increase-in-web-malware-activity.html, Nov. 2009.

[8] L. Constantin, "Web Malware Employs New Obfuscation Technique," http://news.softpedia.com/news/Web-Malware-Employs-New-Obfuscation-Technique-115349.shtml, June 2009

[9] J. San Jose, "New Anti-analysis Technique for Script Malware," http://blog.trendmicro.com/new-anti-analysis-technique-for-script-malware/, June 2010.

[10] P. Likarish, E. Jung and I. Jo, "Obfuscated Malicious Javascript Detection using Classification Techniques," Proceedings of the 4th International Conference on Malicious and Unwanted Software, pp. 47-54, Oct. 2009.

[11] A. Schmidt et Al., "Smartphone Malware Evolution Revisited: Android Next Target?," Proceedings of Malware 2009, Oct. 2009.

[12] M. Hypponen, "Malware Goes Mobile," Scientific American, pp. 70-77, Nov. 2006.

[13] C. Foresman, "Truly malicious iPhone malware now out in the wild," http://arstechnica.com/apple/news/2009/11/truly-malicious-iphone-malware-now-out-in-the-wild.ars, Nov. 2009.

[14] T. Kessler, "Jailbreakers beware: iPhone malware evolves rapidly," http://reviews.cnet.com/8301-13727_7-10395178-263.html, Nov. 2009.

[15] Rolf Rolles, "Unpacking Virtualzation Obfuscators," Proceedings of the 3rd USENIX Workshop On Offensive Technologies, Aug. 2009.

[16] Craig Smith, "Creating Code Obfuscation Virtual Machines," Tutorial in RECON08

[17] Jean Borello, Eric Filiol and Ludovic Me, "Are current antivirus programs able to detect complex metamorphic malware? An empirical evaluation," Proceedings of the 18th EICAR Annual Conference. May 2009.

[18] Min Gyung Kang, Pongsin Poosankam, and Heng Yin, "Renovo: A Hidden Code Extractor for packed Executables," Proceedings of WORM07, Nov. 2007.

[19] Kyungroul Lee, Ilsun You and Kangbin Yim, "A Hint to the Analysis of the Packed Malicious Codes," Proceedings of the WISA2010, Aug. 2010.