BRCORP
TECHNOLOGY | INNOVATION | INDUSTRY

5th World Conference on
Applied Sciences, Engineering & Technology
02-04 June 2016, HCMUT, Vietnam

An Experimental Study on Identifying Obfuscation Techniques in Packer

NGUYỄN MINH HẢI, QUẢN THÀNH THƠ
Department of Software Engineering, HCM city University of Technology, Ho Chi Minh City, VietNam
Email: nguyenmhai1984@gmail.com, qttho@cse.hcmut.edu.vn

Abstract: Malware is one of the most important problems in computer security. There are two main approaches for detecting malware, signature matching and virtual emulation. Signature is a typical bit pattern, which characterizes malwares. Most of industrial malware detection methods depend on regular expression based signature recognition. Virtual emulation prepares a sandbox to explore behaviour of malwares, which requires a deep encoding of system environments to emulate windows APIs [1]. However, emulation requires finding a suitable abstraction level which is very heavy task. Moreover, these techniques are easily defeated by the obfuscation techniques, e.g. indirect jump, self-modifying code, Structured Exception Handling (SEH) and many other techniques which are adopted in packer. In fact, most of modern malware use packers for creating a new variant which cheats the antivirus software, According to a report of Semantic Lab [2], nearly 80% of malware are packed by packer. This paper targets on the problem of identifying the obfuscation techniques which are adopted in some well-known packers. It proposes an experimental study of obfuscation techniques which are used in 7 popular packers which include UPX, FSG, NPACK, ASPACK, PECOMPAT, PETITE, and YODA. We develop our pushdown model generation of malware, BE-PUM as a generic unpacker tool by implementing the anti-anti-analysis techniques against the obfuscation techniques in these packers. During the on-the-fly disassembly, BE-PUM observes and measure the frequency of obfuscation techniques adopted in packers. We have performed the experiments in 8 packers using BE-PUM and achieved very promising results.

Keywords: *Concolic Testing, Pushdown System, Malware Detection, Binary Code Analysis, Self-Modifying Code, Packer Identification, Obfuscation Technique*

Introduction:

Most of the modern popular malwares are packed by a packer. Packer mutates a malware into another executable to evade the signature based technique of anti-virus softwares. For solving this problem, most of anti-virus software focuses on identifying packer based on the packer signature, a binary pattern specific to each packer. However, since malware can obfuscate the packer signature, this approach is easily defeated when dealing with real-world malware.

There is a considerable binary analysis tools, e.g. JakStab [3][4][5], Syman [6] and BINCOA [7]. However, they are quite limited when dealing with packers. In [8][9] we have proposed a tool BE-PUM (Binary Emulator for Pushdown Model generation), for generating a precise control flow graph (CFG). BE-PUM can handle many typical obfuscation techniques of malware, e.g., indirect jump, self-modification, overlapping instructions, structured exception handler (SEH) and many techniques which are covered in packers.

In this paper, we introduce an experimental study on identifying obfuscation techniques in 2000 real-world malware. We have developed BE-PUM as a generic unpacker tool by implementing the anti-anti-analysis techniques against the obfuscation techniques in these packers. During the process of model generation, BE-PUM can observe and measure the frequency of obfuscation techniques adopted in 7 packers which include UPX, FSG, NPACK, ASPACK, PECOMPAT, PETITE, and YODA.

We have performed the experiments in 2000 real-world malware belonged to these packers for checking the effectiveness of our approach.

The rest of this paper is organized as followed. Section “Materials and Methods” briefly describes the obfuscation techniques and the methods for detecting them. Section “Results and Discussions” shows our experiments on 2000 malwares taken from VirusTotal. The final section is the conclusion of our paper.

Materials and Methods:

Inspired by [10], the obfuscation techniques in packer are categorized into 6 groups including *Entry/Code placing obfuscation, Self-modification code, Instruction obfuscation, Anti-tracing, Arithmetic operation* and *Tamper detection*. The first group includes 5 main techniques, *Dynamic Code, Code Layout, Overlapping functions, Code Chunking* and *Anti Rewriting*. The second one composes of 3 main techniques, *Dynamic Code, Code Overwriting* and *Overlapping Blocks*. The third group includes the *Indirect Jump* technique. The fourth group includes techniques of *SEH* and *special APIs*. The fifth group includes 2 techniques, *Obfuscated Constants* and *Checksumming*. And the final group includes *Checksumming, Timing Check* and *Anti-Debugging*.

The table 1 briefly describes the techniques which are supported in each packer.

WCSET 2016041 Copyright © 2016 BASHA RESEARCH CENTRE. All rights reserved

Table 1: Supported techniques in packer

Name	UPX	ASPACK	FSG	NPACK	PECOMPACT	PETITE	YODA
Packing-Unpacking	x	x	x	x	x	x	x
Overwriting	x	x	x	x	x	x	x
Indirect Jump	x	x	x	x	x	x	x
Obfuscated Constants	x	x	x	x	x	x	x
Code Chunking		x	x	x		x	x
Stolen Bytes		x		x	x		
Checksumming	x	x	x	x	x	x	x
SEH					x	x	
2API	x		x	x	x		x
Anti-Debugging							x
Dynamic Code	x	x	x	x	x	x	x
Code Layout	x	x	x	x	x	x	x
Overlapping Function		x	x	x	x	x	x
Overlapping Blocks	x	x	x	x	x	x	x
Anti-Rewriting	x	x	x	x	x	x	x
Timing Check							

Entry/Code Placing Obfuscation

a. Dynamic Code

Dynamic code technique is used in packer in two forms including overwriting and packing/unpacking.

- Overwriting

This technique is also called *self-modifying code* (SMC). Packer exploits SMC to modify binary which is dynamically loaded onto memory. Using this feature, packer evades the technique of signature matching which many anti-virus softwares base on for verifying malwares. Since BE-PUM supports on-the-fly model generation with the capability of capturing the modification of binary code in dynamic way, it can easily detect this technique. Moreover, BE-PUM also implements the procedure to locate the position of memory value for verifying whether it is in code section or not. The code below describes the overwriting technique in YODA packer.

```
4050D3 STOS BYTE PTR ES:[EDI]
4050D6 CMP ECX , EBP
```

Listing 1: Overwriting in YODA

- Packing/Unpacking

This technique is also called Encryption/Decryption. It uses the same idea of SMC technique with the appearance of loop. It can be easily recognized in BE-PUM. The code below describes the packing/unpacking technique in YODA packer.

```
405067 CALL 40506C
40506C POP EBP
40506D SUB EBP, 40286C
405073 MOV ECX, 40345D
. . .
405092 LODS BYTE PTR DS : [ EDI ]
405093 ROR AL, 0DB
. . .
4050C3 STOS BYTE PTR ES : [ EDI ]
4050C4 LOOPD 405092
```

Listing 2: Packing/Unpacking in YODA

b. Code Layout

The code layout technique is presented in packer in three ways, overlapping functions, overlapping blocks and code chunking.

- Overlapping Functions

The main idea of this technique is to interleave code between functions. Each function is placed between *CALL* and *RET* instruction. Using this feature, on encountering each newly-discovered function, BE-PUM checks whether it overlaps with others.

- Overlapping Blocks

This technique uses the same idea with overlapping function. Each block is delimited by the appearance of jump instruction. BE-PUM records these blocks and verifies its location for interleaving. The code below describes the overlapping block features in Yoda.

```
40509D DEC AL
40509F ADD AL, CL
. . .
4050C2 CLC
4050C3 STOS BYTE PTR ES:[EDI]
. . .
4050BD JMP 4050C0
```

Listing 3: Overlapping block in YODA

- Code Chunking

This technique splits code blocks in many groups of small instructions ending with a jump instruction. BE-PUM detects this technique by checking whether packer uses many jump instructions. The distance of these instructions is less than or equals the threshold which is about 20 bytes. This threshold is determined on many observations and testing.

```
40546B JMP 40546E
. . .
40546E STC
. . .
405477 JMP 40547A
. . .
40547A JMP 40547D
. . .
40547D NOP
```

Listing 4: Code chunking in YODA

c. Anti-Rewriting

- Stolen Bytes

This technique allocates a buffer by calling Windows API *VirtualAlloc* and copies unpacked code into this buffer instead of overwriting the original one. BE-PUM detects this technique by recognizing the occurrence of this special API.

```
405899 PUSH EDX
40589A MOV EBP, EAX
40589C PUSH 40
40589E PUSH 1000
4058A3 PUSH DWORD PTR DS:[EBX+4]
. . .
4058AF CALL kernel32.VirtualAlloc
```

Listing 5: Stolen bytes in PECOMPACT

- Checksumming

This technique can be detected in BE-PUM by recognizing 2 stages. In the first stage, there is a loop which calculates the total checksum value. This loop does not modify the memory value in code section. Otherwise this technique will be captured as packing/unpacking technique. In the second stage, BE-PUM detects the occurrence of comparison instruction which compares checksum with memory value.

```
4057F5 XOR EAX, EAX
4057F7 LODS BYTE PTR DS:[EDI]
4057F8 XOR AL , DL
4057FA SHR EAX, 1
405806 INC ECX
405C51 PUSH EAX
405C52 XOR EAX , 7DCC805B
405C57 SUB EAX , 2A5DA2BD
405C63 JNZ 40527B
```

Listing 6: Checksumming in TELOCK

<u>Self-Modification</u>

This technique can be divided into Dynamic Code, Code Overwriting, and Overlapping Block. These techniques are described in the above part.

<u>Instruction Obfuscation</u>

This technique is also called Indirect Jump techniques. It stores the target of jump instruction in a register, memory or stack frame. For detecting indirect call, BE-PUM verifies whether the target of *CALL* instruction is located in a register or memory. For checking indirect jump, BE-PUM applies the same way. With the indirect return, BE-PUM first records the top stack value when it jumps into a function. On encountering *RET* instruction, BE-PUM pops the top stack for comparing with the recorded value. If these values are not equal, BE-PUM captures this technique as indirect return.

```
4053FA CALL DWORD PTR DS:[ESI+503C]
```

Listing 7: Indirect call in UPX

<u>Anti-Tracing</u>

a. Structured Exception Handling (SEH)

BE-PUM detects this technique in two stages. The first stage setups the exception by storing the value of exception address in the *FS:[0]*. The second stage triggers the exception by diving by zero, memory violation of read or write instruction, or causing interrupt.

```
405116 PUSH 4022E3
40511B PUSH DWORD PTR FS:[0]
405122 MOV DWORD PTR FS:[0] , ESP
40521E MOV BYTE PTR DS:[EDI] , AL
```

Listing 8: SEH in PETITE

b. Two Special APIs

Packers use the two APIs, *LoadLibrary* and *GetProcAddress* for getting the necessary dynamic link library. BE-PUM detects this technique by recognizing these two special APIs.

```
4001C5 PUSH EAX
4001C6 CALL kernel32.Load Library
4001D4 PUSH EAX
4001D5 PUSH EBP
4001D6 CALL kernel32.GetProcAddress
```

Listing 9: Using two special APIs in FSG

<u>Arithmetic Operation</u>

This technique composes of Obfuscated Constants and Checksumming. Checksumming technique is described above.

a. Obfuscated Constants

The main idea of obfuscated constant is to replace the constant value with arithmetic instructions which produces the same results. BE-PUM detects this technique by checking whether there are arithmetic instructions with all of operands which are concrete value (not symbolic value).

```
405C51 PUSH EAX
405C52 XOR EAX, 7DCC805B
405C57 SUB EAX, 2A5DA2BD
```

Listing 10: Obfuscated constant in PETITE

Tamper Detection

This technique also includes Checksumming technique as we explained above. Moreover, this technique composes of two other techniques, including timing check and anti-debugging.

a. Timing Check

BE-PUM detects this technique by recognizing the special APIs relating to time of system, e.g. *GetTickCount, GetSystemTime, GetLocalTime*, etc. or special instruction e.g. *RDTSC.*

b. Anti-Debugging:

This technique can be detected by recognizing special APIs which packer exploits for verifying its running environment, e.g. *IsDebuggerPresent*, *CheckRemoteDebuggerPresen, NtQueryInformation Process, NtQuerySystemInformation* and *NtQueryObject*, etc.

```
405888 CALL kernel32.IsDebuggerPresent
```

Listing 11: Anti-debugging in YODA

c. Hardware Breakpoints

Hardware Breakpoints is exploited via usage of the debug registers. By triggering an interrupt exception with *INT3* instruction, packer jumps to a special procedure which modifies the special value of debug registers *DR0, DR1, DR2* and *DR3*. These registers store the values of location where *TELOCK* plans to cause an exception by interrupting. When the control flow is transferred to the locations stored in debug registers, it will be set as hardware breakpoint and causes an *SINGLE STEP EXPCEPTION*.

```
40508C INT3
40508D NOP
40508E MOV EAX, EAX
405090 STC
405099 CLC
40509E CLD
4050A3 NOP
```

Listing 13: Hardware breakpoints in TELOCK

Packer will cause an exception at location *40508C*, then it will setup for debug registers by modifying the values of *DR0* at location *405090*, DR1 at location *405099*, *DR2* at location *40509E*, and *DR3* at location *4050A3*. When the control flow jumps to this location, Telock moves to special routine for setting up a special value in memory for later packing/unpacking routine. Since BE-PUM supports 8 debug registers, it can easily detect this technique.

Results and Discussion:

In this section, we present our experiments of obfuscation technique detection and packer identification on 2000 real-world malware, taken from VirusTotal. Our experiments are performed on Windows XP with AMD Athlon I I X4 635, 2.9 GHz and 8GB. Among 2000 malware, BE-PUM has detected the techniques of Packing/Unpacking, Overwriting, Indirect Jump, Obfuscated Constants, Co de Chunking, Stolen Bytes, Checksumming, SEH, 2API, Anti-Debugging, Dynamic Code, Code Layout, Overlapping Function, Overlapping Blocks, Anti-Rewriting and Timing Check in 617, 1502, 1460, 1761, 965, 183, 1392, 709, 140, 8, 1502, 1460, 1225, 1430, 1399 and 234 malwares respectively. Table 2 below summarizes our result of detecting techniques in BE-PUM

Table 2: Result of technique detection

Name	Number of Malwares
Packing/Unpacking	617
Overwriting	1502
Indirect Jump	1460
Obfuscated Constants	1761
Code Chunking	965
Stolen Bytes	183
Checksumming	1392
SEH	709
2API	164
Anti-Debugging	8
Dynamic Code	1502
Code Layout	1460
Overlapping Function	1225
Overlapping Blocks	1430
Anti-Rewriting	1399
Timing Check	234

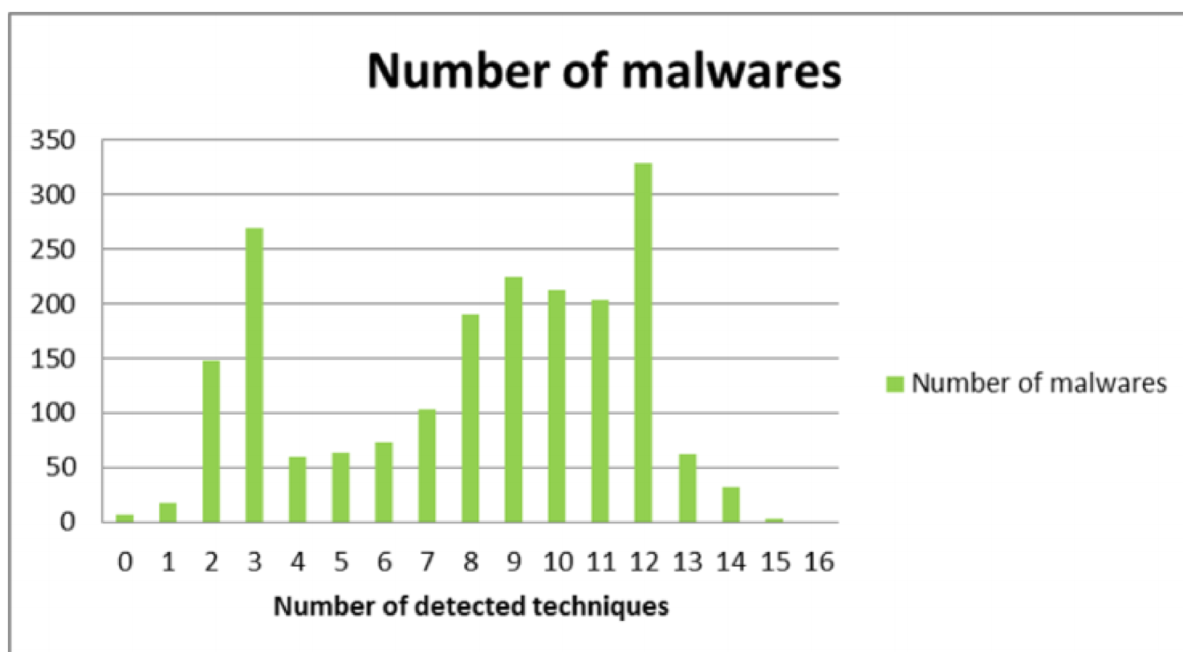

Figure 1: Result of correlation detection

Figure 1 describes the correlation between the number of malwares and the number of techniques which are detected in them.

Conclusion:

This paper proposes an experimental study on detecting obfuscation techniques during disassembly. The method is implemented as BE-PUM (Binary Emulator for PUshdown Model generation). Experiments and observation confirm that BE-PUM correctly handles obfuscation techniques of 7 packers.

References:

[1] A. Mori, T. Izumida, T. Sawada, and T. Inoue. A tool for analyzing and detecting malicious mobile code. In ICSE, pages 831–834, 2006. LNCS 3233

[2] Anti-virus technology whitepaper. Technical report, BitDefender, 2007.

[3] J. Kinder and D. Kravchenko. Alternating control flow reconstruction. In VMCAI, pages 267–282, 2012. LNCS 7148.

[4] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In VMCAI, pages 214–228, 2009. LNCS 5403

[5] Johannes Kinder. Static Analysis of x86 Executables. PhD thesis, Technische Universitat Darmstadt, 2010.

[6] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA framework for binary code analysis. In CAV, pages 165–170, 2011. LNCS 6806.

[7] T.Izumida, K.Futatsugi, and A.Mori. A generic binary analysis method for malware. In International Workshop on Security, pages 199–216, 2010. LNCS 6434.

[8] M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa. A hybrid approach for control flow graph construction from binary code. In IEEE APSEC, pages 159–164, 2013.

[9] M. H. Nguyen, M. Ogawa, and T. T. Quan and. Obfuscation code localization based on cfg generation of malware. In FPS, to appear in LNCS, 2015.

[10] K.A. Roundy and B.P. Miller. Binary-code obfuscations in prevalent packer tools. In ACM Comput. Surv, 46, pages 4:1–4:32, 2013.