Packer identification based on metadata signature

Nguyen Minh Hai
HoChiMinh City University of Technology
hainmmt@cse.hcmut.edu.vn

Mizuhito Ogawa
Japan Advanced Institute of Science and Technology
mizuhito@jaist.ac.jp

Quan Thanh Tho
HoChiMinh City University of Technology
qttho@hcmut.edu.vn

ABSTRACT

Malware applies lots of obfuscation techniques, which are often automatically generated by the use of packers. This paper presents a packer identification of packed code based on *metadata signature*, which is a frequency vector of occurrences of classified obfuscation techniques. First, BE-PUM (Binary Emulator for PUshdown Model generation) disassembles and generates the control flow graph of malware in an on-the-fly manner, using *concolic testing*. Second, obfuscation techniques in the generated control flow graph are detected based on the formal criteria of each obfuscation technique. Last, the used packer is identified with the chi-square test on the *metadata signature* of a packed code. The precision is evaluated with experiments on 12814 malware from VX heaven and Virusshare, in which 608 examples are detected inconsistent with commercial packer identification at PEiD, CFF Explore, and VirusTotal. We manually confirm that, except for 1 example, BE-PUM is correct. The only case that BE-PUM misunderstands is between MEW and FSG, which are quite similar packers and current BE-PUM extension does not support MEW.

Keywords

concolic testing, malware, binary code analysis, packer

1. INTRODUCTION

Over the past decades, malware or *malicious software* has been becoming a real threat to computer users. Malware transparently infects nasty, unwanted and malicious code inside victim machine for illegal financial gain, personal information thief and other malicious actions. In 2014, according to a report from International Data Corporation (IDC) and the National University of Singapore (NUS), more than 491 billion dollars has been consumed for the war against malicious softwares [1]. To evade firewall and antivirus (AV) scanners, malware authors utilize packer, a binary software which employs code obfuscation techniques. According to [2, 3, 4, 5], over 80% of malware is obfuscated using packers for protecting against anti-virus systems, e.g. ASPACK[1], FSG[2], PECOMPACT[3], TELOCK[4], UPX[5], and YODA's Crypter[6]. Packers apply a lot of obfuscation techniques to avoid the detection [25]. Hence, the identification and unpack of packing techniques are becoming vital for revealing a malicious intention and conducting further analysis.

The packer transforms the targeted file into another compressed executable, which preserves the original functionality. Its first aim is to reduce the size of binary files. Another aim is to evade reverse engineering, or protect the licensed software from crackers, which is also favorable for generating malware.

A packed binary contains an unpacking code (called *a restoration loader stub*), which decrypts the original file with different algorithms specific to each packer. After unpacking the payload, it transfers the control flow to the original entry point (OEP). Moreover, many packers, e.g., TELOCK, YODA's Crypter, provide armored stubs for protecting against reverse engineering, cracking, and tampering with anti-debugging and anti-reversing techniques.

When the used packer is identified, we can use specialized unpackers, as well as generic unpackers [13, 14, 16, 20, 26] to obtain the original binary code, which closely relates to the detection of the OEP. Most of the packer identification are based on the binary signature, which is a packer-specific binary sequence in a packed code. For instance, CFF Explorer[7] and PEiD[8] are popular tools for the packer identification by finding exact matching with the binary signature. There are several attempts to improve binary pattern matching [7, 10, 15, 30, 34].

Contributions

Our key contributions are summarized as follows.

1. We newly propose the *metadata signature* of packers, as an alternative to the binary signature, which is the frequency vector of classified obfuscation techniques in the unpacking code of a packed binary. First, we extend a binary model generator BE-PUM [22, 23], available online at `bepum.jaist.ac.jp`, for the obfuscation

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

[1]`http://www.aspack.com`
[2]`http://fsg.soft112.com`
[3]`https://bitsum.com/pecompact`
[4]`http://www.telock.com-about.com`
[5]`http://upx.sourceforge.net`
[6]`http://www.yodas-crypter.com-about.com`
[7]`http://www.ntcore.com/exsuite.php`
[8]`http://www.secretashell.com/codomain/peid`

technique detection. BE-PUM has strong disassemble feature, e.g., it automatically detects the destination server of information leak by EMDIVI[9]. Second, the obfuscation techniques used in packers are classified following to the survey [25] and chosen as indicators. The formal criteria to detect them are carefully defined based on manual observation on more than 40 malwares. Lastly, inspired by [27, 29, 28], we apply the chi-square test for the packer identification based on the *metadata signature*. The average and the membership thresholds for the chi-square test are set by training on test sets, which are taken from packed toy examples and real-world malware. BE-PUM generates a control flow graph along the execution paths in an on-the-fly manner, and it computes the membership at each generation step. When the membership exceeds the threshold, it identifies the used packer. This occurs around the end of the unpacking code. Thus, BE-PUM detects a near region of the OEP, which is often investigated by analyzing dynamic behavior [16, 20, 26] or statistic features [13, 14].

2. We have performed the experiments for evaluating the precision of our approach on 5374 malware from VX heaven[10] and 7440 malware from Virusshare[11], in which 608 inconsistent examples are detected with commercial packer identification at PEiD, CFF Explore, and VirusTotal. We manually investigate all and observe that, except for 1 example, BE-PUM is correct. The only case that BE-PUM misunderstands is between MEW and FSG, which are quite similar packers and the current BE-PUM does not cover MEW.

3. There are three different techniques for unpacking a packed file, manual unpacking, static unpacking and generic unpacking [8]. Manual unpacking is time consuming. Static unpacking can be evaded by unknown or custom packer. Due to the nature of binary emulator, we enhance BE-PUM as a generic unpacking tool which is very important for unpacking custom packer. BE-PUM can simultaneously unpack and detect custom packer based on the occurrence of packing/unpacking and 2API techniques.

The paper is organized as follows. Section 2 and 3 present preliminaries and BE-PUM, respectively. Section 4 gives the formal criteria of the obfuscation techniques in packed code. Section 5 defines the metadata signature. Section 6 shows the experiments on the packer identification. Section 7 concludes the paper.

Related Work

There are two main targets in the packer analyses. The first one is packer identification, which is mostly by the binary signature detection [15, 34, 30, 7]. The binary signature is often located around the OEP, and this closely relates to the OEP detection and unpacking. Li Sun, et. al. observed the randomness profile on the whole binary code [34], applying various pattern matching techniques, e.g., the k-nearest neighbor, the best-first decision tree, the sequential minimal optimization, and the naive Bayes to classify packers. It shows high accuracy, more than 95%. Unfortunately, the datasets used for the experiments are no longer available for comparison. Our approach focuses more on semantic features based on precise control flow graphs, which is almost 100% precise at the cost of the execution time. We expect our approach can be applied for automatic training set generation.

The second goal is to unpack the packed files. Some remarkable tools following this goal include OllyBonE[12], Renovo [16] and CoDisasm [11]. *OllyBonE*[13], a plugin of *OllyDbg*[14] aims to unpack code by finding the OEP. This tools consists of a Windows kernel driver for implementing the page protection of an arbitrary region. The method to find the OEP consists of several steps. It first selects the memory region and sets exception break-on-execute to protect this region. Then, it waits for unpacking stub to finish. If the control flow transfers to the address inside the protected area, the exception occurs and the OEP is found. However, this plugin fails in many cases, when packers employ anti-debugging techniques, e.g., exploiting the API *IsDebuggerPresent@kernel32.dll*.

Renovo [16] is built on the top of an emulated environment, TEMU[15], which is a dynamic analysis component of BitBlaze [31]. Renovo stores a shadow copy of memory space of the targeted file for observing and monitoring program execution to write on memory at run time. Renovo finds the OEP by extracting the newly generated code and data. The detection of the OEP is also used in [14, 13] with statistical analyses. Polyunpack [26] and Omniunpack [20] detect the original payload by observing dynamic behavior.

There are several static malware analyses, such as model checking and symbolic execution. For instance, malware detection by model checking is found in [32, 33], but they simply use IDApro to disassembly, which limits the capability to handle packed code. Another example is McVeto [35], which first collects candidate destinations of an indirect jump by a static analysis, and check whether each is feasible by symbolic execution. However, it also limits the handling capability of packed codes, since static analyses are easily confused by arithmetic operations. BE-PUM applies dynamic symbolic execution (concolic testing) to decide the destinations of an indirect jump, which is more robust.

More dynamic use of symbolic execution is *CoDisasm* [11], which is built on *Intel/PIN*. Its ideas overlap with BE-PUM, but more with a dynamic analysis. It proposed *concatic disassembly*, which first dynamically executes on Intel/PIN and retraces the path with symbolic execution. If possible branches are found, it confirms with concolic testing.

BINSEC/SE [36] built on BINSEC applies syntactic disassembly (similar to IDApro) to an intermediate representation *DBA*. BINSEC is prepared as a binary code analyzer framework, and BINSEC/SE adds dynamic symbolic execution on BINSEC. Because of the syntactic disassembly, it seems to target more on compiled binaries.

2. PRELIMINARIES

[9]`http://blog.jpcert.or.jp/2015/11/emdivi-and-the-rise-of-targeted-attacks-in-japan.html`
[10]`http://vxheaven.org`
[11]`https://virusshare.com`
[12]`http://www.joestewart.org/ollybone`
[13]`http://www.joestewart.org/ollybone`
[14]`http://www.ollydbg.de`
[15]`http://bitblaze.cs.berkeley.edu/temu.html`

2.1 Chi-square test

When we observe the frequency of occurrences of features of interest, the chi-square test is one of the standard statistical methods to profile and classify based on these features. The chi-square test assumes that such features have the degree k of freedom, which means the number of independent random variables. Then, the standard chi-square distribution is obtained, and the probability of failures (the statistical significance) is a function of the given threshold and the degree of freedom. For instance, the significance 0.05 (which is often a standard choice) is obtained with the threshold $\epsilon = 3.84$ when the degree of freedom is 1.

Let $f = (f_1, f_2, \cdots, f_n)$ be a natural number vector of the dimension n. Given n-features, f_i is the number of the occurrences of the i-th feature. For instance, critical API calls are such example features, used for malware profiling in [27]. We will set obfuscation techniques as such features for packer identification. Either case tries to say whether an objective belongs to a specified class of malware or packed code, respectively. Thus, k is set to 1 and their significance become the allowance of the failure on the classification.

We first prepare the training set T_r, which are already classified into some class $\mathcal{C}$. The average vector $E = (E_1, E_2, \cdots, E_n)$ of occurrences is defined by

$$E_i = \frac{\Sigma_{x \in Tr} f_i(x)}{|T_r|}$$

The chi-square test proceeds for a sample $O = (O_1, O_2, \cdots, O_n)$ as follows. Let

$$\chi_i^2(O) = \frac{(O_i - E_i)^2}{E_i} \qquad \lambda(O) = \frac{|U|}{n}$$

where $U = \{i \mid \chi_i^2(O) \leqslant \epsilon \text{ and } 1 \leqslant i \leqslant n\}$. The degree of membership $\lambda(O)$ presents how likely it is for O to belong to $\mathcal{C}$.

Second, prepare the test set T_e, which are also already known to belong to $\mathcal{C}$ and $T_r \cap T_e = \varnothing$. The membership threshold $\bar{\lambda}$ is set to the average of the degree of membership in T_e, i.e.,

$$\bar{\lambda} = \frac{\Sigma_{O \in T_e} \lambda(O)}{|T_e|}$$

When a new sample x is examined, $\lambda(x) \geqslant \bar{\lambda}$ concludes that x is in $\mathcal{C}$.

2.2 Structure of packed code and binary signature

Roughly speaking, a packed code has a PE (Portable Executable) header, the unpacking code, and the packed payload. Their structures differ depending on used packers. For instance, UPX produces the packed code below (Fig. 1 in [7]).

A packer packs the payload, and when the packed code is executed, it first unpacks the packed payload. The OEP is the entry of the packed payload. We focus on the unpacking code, in which the use of obfuscation techniques is quite independent from the payload content. The end of the unpacking code is near to the OEP.

A popular approach of the packer identification is by the binary signature, which is a binary pattern appearing in a packed code. *PEiD* and *CFF Explorer* are popular examples of such tools. For instance, they check the use of the packer *UPX Protector v1.0x* by detecting the binary signature below in the entire PE header. However, when the self-modification mutates and/or the code layout of a file is modified, they are confused. After the packing, the binary signature can be easily modified by changing some bytes manually.

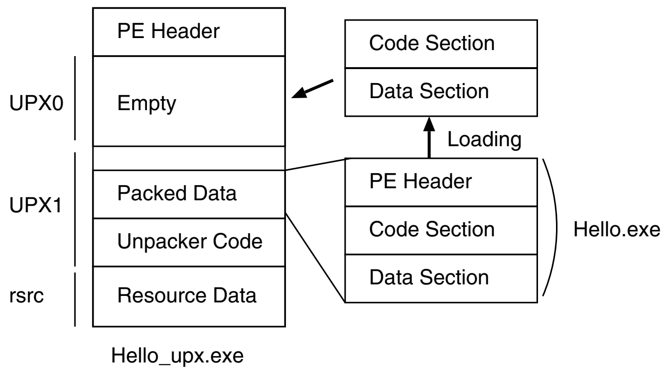


Figure 1: Example of packed code in UPX

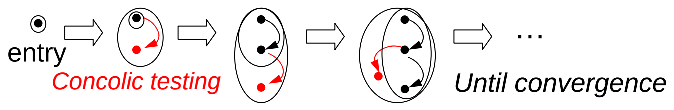


Figure 2: Model generation using concolic testing

```
EB ?? ?? ?? ?? ?? 8A 06 46 88 07 47 01 DB 75 07 8B 1E 83
EE FC 11 DB
```

3. BE-PUM

3.1 Disassembly by BE-PUM

A tool BE-PUM (Binary Emulator for PUshdown Model generation) generates a precise control flow graph (CFG) [23], as well as precise disassembly of x86 binary code. It can handle typical obfuscation techniques of malware, e.g., *indirect jump*, *self-modification*, *overlapping instructions*, and *structured exception handler (SEH)*, which cover obfuscation techniques introduced by a *packer* (see Section 4).

BE-PUM generates a model of binary code in an on-the-fly manner, following the execution paths. The concolic testing is used at each step for extending a model as presented in Figure 2. Compared to McVeto [35], which statically detects possible destinations of indirect jumps and confirms their feasibility by symbolic execution, BE-PUM decides the destinations of indirect jumps by concolic testing as described in Figure 3.

The Figure 4 shows the architecture of BE-PUM, which consists of three components: *symbolic execution*, *binary emulation*, and *CFG storage*. It applies *JakStab 0.8.3* [19, 18, 17] as a preprocessor to compute a single step disassembly, and an SMT solver

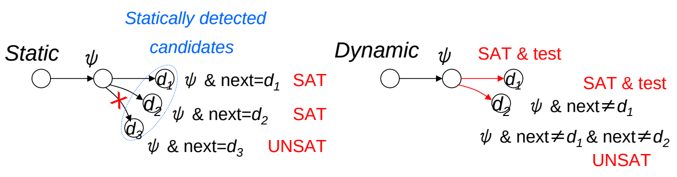


Figure 3: Comparison between static symbolic execution and dynamic symbolic execution

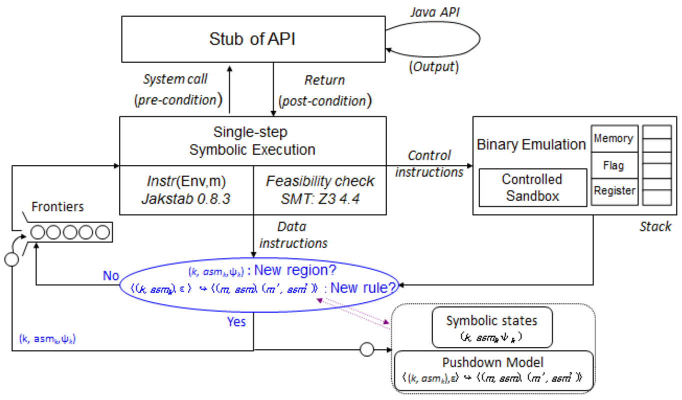


Figure 4: The architecture of BE-PUM

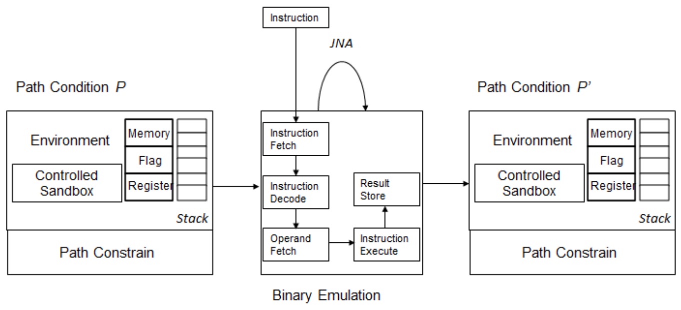


Figure 5: One-step concolic testing in BE-PUM

Z3 4.4 as a backend engine to generate test instances for concolic testing.

The Figure 5 shows how BE-PUM executes one-step concolic test. The *binary emulation* either interprets an x86 instruction, or spawns to a Windows API stub, where the Windows API stub calls JNA to execute a native shared library in real Windows environment to obtain the return value and the environment update.

The *binary emulation* also transfers the *pre-condition* P to the *post-condition* P'. The *path condition* consists of arithmetic constraints on symbolic values only, and the *memory model* describes the environment, which set up the value of registers, memory locations, and flags by arithmetic expressions of symbolic values.

If an exception like the *division by zero* occurs, BE-PUM detects it at the binary emulator, which passes them to the Windows system error handler.

3.2 Extensions of BE-PUM for packer identification

We observe that x86 instructions and APIs used in the packer techniques are quite typical, but sometimes tricky. For instance, the unpacking code first allocates a new virtual memory using *VirtualAlloc@kernel32.dll* or reuses the empty section in a file. It then decompresses the packed payload (including the import table of Windows APIs that are used in the payload) into the new section, and dynamically loads the imported libraries, using the two APIs, *GetProcAddress@kernel32.dll* and *LoadLibrary@kernel32.dll*.

Some packers also contain techniques for the integrity checking, the protection, and the anti-analysis, by using special x86 instructions and APIs. Timing check, which checks timing anomaly, e.g., the execution is too slow, uses the *RDTSC* instruction. The basic debugger detection is to call *IsDebuggerPresent@kernel32.dll* to detect whether currently executed in the debugged mode.

Instead of directly calling *IsDebuggerPresent@kernel32.dll*, sophisticated packers, like Yoda, directly access TIB (Thread Information Block) to get the address of PEB (Process Environment Block) at the offset $0x30$, and check the *BeingDebugged* flag at the offset $0x02$ in the PEB.

To cover them, various extensions (compared to [23]) have been contributed for improving BE-PUM.

- The binary emulation of BE-PUM additionally support 200 x86 instructions and 140 windows API stubs, which are typically used in packers. Some of them needs special care for specific obfuscation techniques. For instance, the anti-tracing requires the stubs for *GetProcAddress@kernel32.dll* and *LoadLibrary@kernel32.dll*. For the anti-tampering, like timing check, an appropriate return value of the *RDTSC* instruction is prepared, not to be judged in an emulation environment.
- The memory model of the binary emulator in BE-PUM accurately simulates the real memory allocation, e.g., PEB and TIB. The PEB is created at the process initialization, and contains the information, e.g., the global context, the startup parameters, the image loader, and the image base address. The TIB stores the information on the currently running thread, e.g., the current thread ID, the PEB address, and the environment pointer.
- The trap flags and the debug registers are introduced to the binary emulator of BE-PUM. For instance, the hardware breakpoint and the structured exception handler (SEH) with the trap flags are used in TELOCK.

Currently, BE-PUM is extended to support 300 popular IA32 (x86) instructions (among > 1000) and 1500 Windows APIs (among > 4000), which are sufficient to trace the unpacking codes of many packers.

3.3 BE-PUM as a generic unpacker

There are three different techniques for unpacking a packed file, manual unpacking, static unpacking and generic unpacking [8]. Manual unpacking is time-consuming and requires deep understanding of kernel. Static unpacking with signature detection can be applied for unpacking files packed by known packers. However, virus developers can evade them using modified or custom packers. Thus, generic unpacking which emulates unknown packed executables until they reveal their malicious behavior in memory is becoming increasingly important for anti-virus providers [8]. Due to the nature of of symbolic dynamic emulator, BE-PUM also works as a generic unpacker. In [9], Xabier, et. al. proposed a taxonomy for run-time packers to measure their structural complexity. Following the classification, current BE-PUM can unpack the Type-III packer containing PE Compact, ASPACK, FSG. For the Type-IV, V and VI packers, BE-PUM needs more observed features for unpacking and classifying. For example, current version of BE-PUM does not support unpacking with multiple threads and interleaved execution in Obsidium 1.2, a Type-IV packer.

Our notable example is *EMDIVI*, which was firstly explored on October 2014 by Kaspersky Laboratory. It caused huge information leak from Japanese governmental pension fund in 2015. BE-PUM disassembles a variation indexed by MD5 code `e9302fe774e22e2b34a395f8e56c6976fe354bb88b5dcfda4ee36984eebd9340`. Until interrupted at `PStoreCreateInstance@Pstorec.dll` (which is currently unsupported in BE-PUM), 20,192 instructions are explored in approximately 2 minutes. During the disassembly, BE-PUM automatically detects the destination server name, the user name, and the password as the arguments of `InternetConnectW@WinINet.dll`. They are obtained by inspecting the stack as

```
ServerName:www.n-fit-sub.com, ServerPort:80,
Username:null, Password:null, Service:3.
```

Current generic unpackers [13, 14, 16, 20, 26] focus on finding the OEP by either statistical or dynamic analysis. BE-PUM directly disassembles without investigating the OEP, and as a result of the packer identification, it will find a near region of the OEP.

For instance, after BE-PUM identifies that *EMDIVI* is packed with UPX v3.0, BE-PUM detects the exact OEP by using the fact that UPX v3.0 ends its unpacking code with the instruction `POPA`. BE-PUM finds the `POPA` instruction in the near region,

```
0x0044d236: popa
0x0044d244: jmp 0x0041eec0
```

and the destination of the next JMP at 0x0044d244 shows the OEP 0x0041eec0.

4. OBFUSCATION TECHNIQUES

4.1 Typical Obfuscation Techniques

Packers apply many obfuscation techniques, which make binary code difficult to explore. Packer techniques are investigated in [25], and with our observation on malware, we focus on 14 obfuscation techniques, which are categorized into 6 groups. We will briefly explain each technique with an example code.

1. **Entry/code placing obfuscation** (*Code layout*): *overlapping functions*, *overlapping blocks*, and *code chunking.*
2. **Self-modification code** (*Dynamic code*: *overwriting* and *packing/unpacking.*
3. **Instruction obfuscation**: *Indirect jump.*
4. **Anti-tracing**: *SEH* (structural exception handler) and *2API* (the use of special APIs, `LoadLibrary` and `GetProcAddress` in `kernel32.dll`).
5. **Arithmetic operation**: *Obfuscated constants* and *checksumming.*
6. **Anti-tampering**: *Timing check, anti-debugging*, *anti rewriting*, and *hardware breakpoints. Anti-rewriting* consists of *stolen bytes* and *checksumming.*

Code layout

IA-32 (x86) has variable-length instruction sets, and overlapping fragments of a binary sequence may be executed in different ways. For instance, `b8 eb 07 b9 eb` and `0f 90 eb` are read as `mov eax, ebb907eb` and `seto bl`, respectively. However, the fragment `eb 0f` is also read as `jmp 45402c`. They are *overlapping instructions.*

When overlapping instructions occur among functions (resp. blocks), they are called *overlapping functions* (resp. *overlapping blocks*). The example below of the *overlapping functions* is generated by FSG. It has two functions F_1 and F_2. F_1 ranges 004001E8 to 004001F1. At 004001EA, 75 05 8A 16 is interpreted as `JNZ SHORT api_test.004001F1` and `MOV DL,BYTE PTR DS:[ESI]`. F_2 ranges 004001EB to 004001F6. At 004001EB, 05 8A 16 46 12 is interpreted as `ADD EAX,1246168A`.

```
00400160 FF13                                CALL 004001EB
....
004001DF FF13 CALL 004001E8
004001E8 02D2 ADD DL,DL
004001EA 7505 JNZ SHORT 004001F1
004001EB                      058A164612 ADD EAX,1246168A
004001EC 8A16 MOV DL, DS:[ESI]
004001EE 46   INC ESI
004001EF 12D2 ADC DL,DL
004001F0                      D2C3       ROL BL,CL
004001F1 C3   RETN
004001F2 4B                              DEC EBX
004001F3 45                              INC EBP
004001F4 52                              PUSH EDX
004001F5 4E                              DEC ESI
004001F6 C3                              RETN
```

When a code is divided into fragments that are connected by the jump instructions, it is called *code chunking.* The examples below are generated by YODA.

```
40446B JMP 40446E
...
40446E STC
...
404474 JMP 404477
...
404477 JMP 40447A
...
40447A JMP 40447D
...
40447D NOP
```

Dynamic code

Dynamic code consists of the *overwriting* and the *packing/unpacking.* The latter is the same to the *encryption/decryption.* The difference is that the latter occurs in a loop. The examples below are generated by YODA. In *Overwriting*, the `STOS` instruction at `4040C3` modifies the `CMP` instruction at `4040C6` to `MOV`.

```
Overwriting              Packing/unpacking
4040C3 STOS ES:[EDI]     404067 CALL 40406C
4040C6 CMP ECX, EBP      40406C POP EBP
                         40406D SUB EBP, 40286C
                         404073 MOV ECX, 40345D
                         404078 SUB ECX, 4028C6
                         ...
                         404092 LODS BYTE PTR DS:[EDI]
                         404093 ROR AL, ODB
                         ...
                         4040C3 STOS BYTE PTR ES:[EDI]
                         4040C4 LOOPD 404092
```

Indirect jump

Indirect jump is a call, a jump, or a return of which the target destination is stored in a register, a memory address, or a stack frame, respectively. An indirect return is the case when the return destination stored in the stack is modified. `54053FA CALL DWORD PTR DS:[ESI+503C]` is an example of indirect call in UPX.

Anti-tracing

- **SEH**: When an exception, e.g., the division by zero and the write-on a protected memory area, occurs, the control is spawn to the system error handler, and the stack is switched to another memory area in a user process. When the system error handler ends, the control returns to the instruction next to the exception and the stack is switched to the original. *Structured Exception Handler* (SEH) is an exception handler written in a user process, which prepares for an exception and post-processes when it occurs. TELOCK uses an SEH with the *trap flag* `AL`. It set `AL` to *true* and causes a *single step exception*, as in the code.
- **2API**: APIs, `LoadLibrary` and `GetProcAddress` in `kernel32.dll`, are used to get the necessary dynamic link library, whose name is stored in `eax`.

```
SEH                           2API
404116 PUSH 4022E3            4001C5 PUSH EAX
40411B PUSH FS:[0]            4001C6 CALL LoadLibrary
404122 MOV FS:[0], ESP        ...
40421E MOV DS:[EDI], AL       4001D4 PUSH EAX
                              4001D5 PUSH EBP2
                              4001D6 CALL GetProcAddress
```

Arithmetic operation

- **Obfuscated Constants**: *Obfuscated constant* replaces a constant with arithmetic instructions resulting the same value. The example below, generated by PETITE, is equivalent to `MOV eax 532F114C`.

```
404C4B MOV EAX, DWORD PTR SS:[EBP+40D280]
404C51 PUSH EAX
404C52 XOR EAX, 7DCC805B
404C57 SUB EAX, 2A5DA2BD
```

- **Checksumming**: A typical example is the CRC checksum. The example below is generated by TELOCK.

```
4047F5 XOR EAX, EAX
4047F7 LODS BYTE PTR DS:[EDI]
4047F8 XOR AL, DL
4047FA SHR EAX,1
404806 INC ECX
404812 JG 4047F5
404C4B MOV EAX, DWORD PTR SS:[EBP+40D280]
404C51 PUSH EAX
404C52 XOR EAX, 7DCC805B
404C57 SUB EAX, 2A5DA2BD
404C63 JNZ 40527B
```

Anti-tampering

- **Anti-Debugging**: Probe whether an execution is in the debug mode. A typical instruction is `CALL kernel32.IsDebuggerPresent`.
- **Stolen bytes** *Stolen bytes* allocates a buffer by calling `VirtualAlloc` and copies the unpacked code there, instead of overwriting the original one. The code below is generated by PECOMPACT.
- **Timing Check**: Timing Check is used to detect timing anomaly compared to the native Windows environment, e.g., an execution is too slow.
- **Hardware breakpoint** *Hardware breakpoints* uses the debug registers $DR0$, $DR1$, $DR2$, and $DR3$, as jump destinations, instead of the standard ones, e.g., *eax*, *ebx*, and *ecx*. The `INT3` instruction prepares a hardware break point, which set these debug registers and triggers a single step exception. The code below is generated by TELOCK, in which an exception is caused at `40408C` and set the debug registers `DR0`, `DR1`, `DR2`, and `DR3` with `404090`, `404099`, `40409E`, and `4040A3`, respectively. These instructions prepare an environment for later *packing/unpacking*.

```
Stolen bytes                  Hardware breakpoint
404899 PUSH EDX               40408C INT3
40489A MOV EBP, EAX           40408D NOP
40489C PUSH 40                40408E MOV EAX, EAX
40489E PUSH 1000              404090 STC
4048A3 PUSH DS:[EBX+4]        404099 CLC
4048A6 PUSH 0                 40409E CLD
...                           4040A3 NOP
4048AF CALL kernel32.VirtualAlloc
```

4.2 Obfuscation Technique Identification

By manually observing the disassembly results over 40 malware by BE-PUM with the aid of *OllyDbg*[16], we set the formal criteria (listed below) of each obfuscation technique, so that BE-PUM automatically locates them during the disassembly.

Code layout

- **Overlapping functions**: Each function body is placed between `CALL` and `RET` instructions. When the overlapping instructions are found between the pairs of `CALL` and `RET`, they are identified as the *overlapping functions*.
- **Overlapping blocks**: Each block is delimited by the occurrences of jump instructions. When the destination of a jump instruction overlaps with previously explored instructions, it is classified into *overlapping blocks*.
- **Code chunking** For identifying *code chunking*, our criteria is whether there are 3 jump instructions with the distance less than or equal to 20 bytes. This threshold (20 bytes) is carefully decided from testing results.

Dynamic code

When modifications of binary code by values at some memory addresses are detected, they are classified as *overwriting*. If further they occur in a loop, they are classified as *packing/unpacking*. Note that BE-PUM observes the memory area whether modification occurs in the code section.

[16]`http://www.ollydbg.de`

Table 1: List of obfuscation techniques

0	*overlapping function*	1	*overlapping block*	2	*code chunking*
3	*overwriting*	4	*packing/unpacking*	5	*indirect jump*
6	*SEH*	7	*2API*	8	*obfuscated constant*
9	*checksumming*	10	*timing check*	11	*anti-debugging*
12	*stolen bytes*	13	*hardware break point*		

Indirect jump

The identification of an *indirect call* and *jump* is straightforward. When the target is either a register or the dereference of a memory address, it is classified to them. For an *indirect return*, the modification of the return destination is detected by recording the top stack value when `CALL` occurs. When `RET` is encountered, the top stack value is compared with the recorded one. If they differ, it is classified into an *indirect return*.

Anti-tracing

- **SEH**: We observe that *SEH* always occurs with the following instruction sequences (up to the use of different registers).
  1. The sequence of the two instructions,

     ```
     PUSH DWORD PTR FS:[0]
     MOV  DWORD PTR FS:[0],ESP
     ```

     which replace the original address of the exception handler, later followed by an exception.
  2. An exception by the *division by zero*, or the *single step exceptions*, `INT1` and `INT3`.
- **2API**: *2API* is identified by detection of the consecutive uses of `LoadLibrary` and `GetProcAddress` in `kernel32.dll`.

Arithmetic operation

- **Obfuscated constants**: We identify the *obfuscated constants* when there are sequences of arithmetic instructions, in which all operands are concrete values (not symbolic values) in symbolic execution.
- **Checksumming**: We identify the *checksumming* by two features. First, a loop does not modify values in the code section. (Otherwise, such a loop is identified as *packing/unpacking*.) Second, the loop contains a comparison instruction that compares a register or a memory value with a constant.

Anti-tampering techniques

We observe straightforward identification.

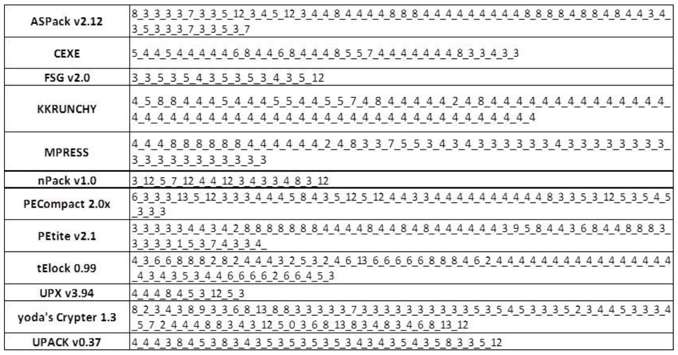
- **Anti-debugging**: An occurrence of a special API to probe a system. They are, `NtQueryInformationProcess`, `NtQuerySystemInformation`, `IsDebuggerPresent`, `NtQueryObject`, and `CheckRemoteDebuggerPresen`.
- **Stolen byte**: An occurrence of a call of the API `VirtuaAlloc`.
- **Timing check**: An occurrence of either a special instruction `RDTSC`, or an API on time, e.g., `GetTickCount`, `GetSystemTime`, and `GetLocalTime`.
- **Hardware breakpoint**: When the debug registers are used, the *hardware breakpoint* is identified.

5. METADATA SIGNATURE

5.1 Obfuscation technique identification and its observation

For the obfuscation technique detection, we focus on the 14 obfuscation techniques in Section 4.2, which are numbered from 0 to 13 as in Table 1.

First, we prepare 4 toy assembly codes (without loops and obfuscation techniques), and pack them with 12 different packers,



ASPack v2.12	8_3_3_3_3_7_3_3_5_12_3_4_5_12_3_4_4_8_4_4_4_4_8_8_8_4_4_4_4_4_4_4_4_4_8_8_8_8_4_8_8_4_8_4_4_3_4_ 3_5_3_3_3_7_3_3_5_3_7
CEXE	5_4_4_5_4_4_4_4_4_6_8_4_4_6_8_4_4_4_8_5_5_7_4_4_4_4_4_4_4_8_3_3_4_3_3
FSG v2.0	3_3_5_3_5_4_3_5_3_5_3_4_3_5_12
KKRUNCHY	4_5_8_8_4_4_4_5_4_4_4_5_5_4_4_5_5_7_4_8_4_4_4_4_4_2_4_8_4_4_4_4_4_4_4_4_4_4_4_4_4_4_ 4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4
MPRESS	4_4_4_8_8_8_8_8_8_4_4_4_4_4_4_2_4_8_3_3_7_5_5_3_4_3_4_3_3_3_3_3_3_4_3_3_3_3_3_3_3_3_ 3_3_3_3_3_3_3_3_3_3_3
nPack v1.0	3_12_5_7_12_4_4_12_3_4_3_3_4_8_3_12
PECompact 2.0x	6_3_3_3_13_5_12_3_3_3_4_4_4_5_8_4_3_5_12_5_12_4_4_3_3_4_4_4_4_4_4_4_4_4_4_8_3_3_5_3_12_5_3_5_4_5 _3_3_3
PEtite v2.1	3_3_3_3_3_4_4_3_4_2_8_8_8_8_8_8_8_4_4_4_4_8_4_4_8_4_8_4_4_4_4_4_4_3_9_5_8_4_4_3_6_8_4_4_8_8_8_3_ 3_3_3_3_1_5_3_7_4_3_3_4_
tElock 0.99	4_3_6_6_8_8_8_2_8_2_4_4_4_3_2_5_3_2_4_6_13_6_6_6_6_6_8_8_8_4_6_2_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4 _4_3_4_3_5_3_4_4_6_6_6_6_2_6_6_4_5_3
UPX v3.94	4_4_4_8_4_5_3_12_5_3
yoda's Crypter 1.3	8_2_3_4_3_8_9_3_3_6_8_13_8_8_3_3_3_3_3_7_3_3_3_3_3_3_3_3_3_3_5_3_5_4_5_3_3_3_5_2_3_4_4_5_3_3_3_4 _5_7_2_4_4_4_8_8_3_4_3_12_5_0_3_6_8_13_8_3_4_8_3_4_6_8_13_12
UPACK v0.37	4_4_4_3_8_4_5_3_8_3_4_3_5_3_5_3_5_3_5_3_4_3_4_3_5_4_3_5_8_3_3_5_12

Figure 6: Obfuscation technique sequences for the observed packers

ASPACK v2, CEXE v1.0b, FSG v2.0, KKRUNCHY v0.23a4, MPRESS v2.19, NPACK v1.0, PECOMPACT v2.0x, PETITE v2.1, TELOCK v0.99, UPX v3.0, YODA's Crypter v1.3, and UPACK v0.37-0.39. Following to the criteria in Section 4.2, we observe that BE-PUM detects the series of the obfuscation techniques (in Table 1).

We manually confirmed with *OllyDbg* that the criteria correctly detect the obfuscation techniques. The series coincide among all toy examples, which encourages our expectation that the occurrences of obfuscation techniques characterize packers.

5.2 Metadata signature

The *metadata signature* of a packed binary is, the *frequency vector* of the numbers of occurrences of obfuscation techniques in the unpacking code.

We select the 14 obfuscation techniques listed in Table 1. The training set T_r and the test set T_e with $T_r \cap T_e = \varnothing$ are selected from binaries of which the used packers are already identified. During the process of the on-the-fly model generation, BE-PUM counts the number of obfuscation techniques. If the frequency vector has the membership beyond the membership threshold of the chi-square test at certain generation step, BE-PUM identifies the used packer.

This also indicates the end of the unpacking code, and as byproduct BE-PUM detects a near region of the OEP. With additional knowledge on each packer, we can identify the exact OEP. For instance, UPX v3.0 always ends its unpacking code with the instruction `POPA`. Thus, by finding near `POPA` when the packer identification has done, the destination of the next `JMP` instruction indicates the exact OEP.

We denote the target obfuscation techniques $T = \{T_1, T_2, \cdots, T_n\}$, the target packer set $M = \{M_1, M_2, ..., M_m\}$, the average vector $E^i = (E_1, E_2, ..., E_n)$ and the membership threshold $\bar{\lambda}_i$ for each packer M_i. $O(B)$ is the frequency vector of obfuscation techniques in the current control flow graph (CFG) of B.

$On\_the\_fly\_Model\_Generation(B)$ extends a CFG of B stepwise by concolic testing. $Model\_Generation\_Stop(B)$ judges whether the CFG generation terminates (possibly by *unsupported instructions*, *unsupported APIs*, or *timeout*).

$Calculate\_Membership\_Degree(O(B), E^i)$ computes the degree of membership of $O(B)$ for the average metadata signature E^i of the packer M_i by the chi-square test. The membership threshold $\bar{\lambda}_i$ for each packer M_i is set to the average of the degree of membership in the test set T_e as described in Sections 2.1.

5.3 Statistical setting

To evaluate the effectiveness of the metadata signature, we select 12 packers, ASPACK v2, CEXE v1.0b, FSG v2.0, KKRUNCHY v0.23a4, MPRESS v2.19, NPACK v1.0, PECOMPACT v2.0x, PETITE v2.1, TELOCK v0.99, UPX v3.0, YODA's Crypter v1.3, and UPACK v0.37-0.39. Although the set of packers is quite small, our method can be extended. Since the training set is taken from packed toy examples, BE-PUM can determine the entry point of these toy examples for calculating the statistics of

```
Input: A packed binary $B$.
Output: $M_i$ if the used packer is identified as $M_i$;
$NONE$ otherwise.
Algorithm:
$O(B) = (O_1, O_2, \cdots, O_n) := (0, 0, \cdots, 0)$;
while $TRUE$ do
    On_the_fly_Model_Generation(B);
    if $Found\_New\_Obfuscation\_Technique() = T_j$ then
        $O(B) := (O_1, \cdots, O_j + 1, \cdots, O_n)$;
        foreach $i := 1$ to $m$ do
            $\bar{\lambda}_T =$
            $Calculate\_Membership\_Degree(O(B), E^i)$;
            if $\bar{\lambda}_T \geqslant \bar{\lambda}_i$ then
                Return $M_i$;
            end
        end
    end
    if $Model\_Generation\_Stop(B)$ then
        Return $NONE$;
    end
end
```

the metadata signature. Then, we can apply the same method for detecting new packer.

For each packer, the training set, the test set, and the membership threshold $\bar{\lambda}_i$ of M_i are set as below.

Table 2: Training and test sets, and threshold

Packer	Training set size	Test set size	Membership threshold
ASPACK v2.4	199	67	0.940299
CEXE	316	79	1
FSG v2.0	216	71	1
KKRUNCHY 0.23a4	143	56	1
MPRESS 2.19	181	63	1
NPACK v1.0	192	63	0.807738
PECOMPACT v2.0x	177	56	0.811847
PETITE v2.1	197	66	0.985294
TELOCK v0.99	201	64	0.98214
UPX v3.94	205	62	0.939421
YODA v1.3	208	65	0.90257
UPACK v0.37-0.39	187	60	0.96405

The training sets are taken from packed toy examples and real world malware whose packers are identified. The latter are taken from *VX heaven* and modern malware. Table 3 shows the average of the metadata signatures in the training set, i.e., the average frequency vectors of the 14 obfuscation techniques listed in Table 1. Note that on our observation, the frequency vector of obfuscation is quite the same regardless of different versions of packers, e.g. UPX v3.0 and UPX v3.94 have the same frequency vector of obfuscation.

Table 3: Metadata signatures for 12 packers

Packer	**Average frequency vector of obfuscation techniques (Table 1) in the test set**													
ASPACK	(0,	0,	0,	13.261,	14.101,	0,	0,	11.206,	24.729,	0,	2.211,	0,	2.201,	0)
CEXE	(0	0	0	3	14.16667	4	2	1	4.16667	0	0	0	0	0)
FSG	(0,	0,	0,	12,	0,	0,	0,	1,	3.348,	0,	0,	0,	1,	0)
KKRUNCHY	(0	0	1	0	13.60185	6	0	1	4	0	0	0	0	0)
MPRESS	(0	0	1.94475	2.9558	5.75138	2	0	1	8.68508	0	0	0	0	0)
NPACK	(0,	0,	0,	2,	6.114,	0,	0,	3.543,	4.829,	0,	2.286,	0,	3.057,	0)
PECOMPACT	(0,	0,	0,	17.045,	9.611,	0,	0,	9.452,	24.420,	1,	2.299,	0,	1.962,	1)
PETITE	(0,	1,	10,	7.882,	16.647,	0,	0,	9.765,	22.706,	2,	0,	0,	0.882,	0)
TELOCK	(0,	0,	15,	21.286,	15.287,	0,	0,	0,	38.143,	20,	1.714,	0,	0.857,	1)
UPX	(0,	0.893,	0,	3.735,	1.995,	0,	0,	2.923,	4.556,	0,	0,	0,	0.985,	0)
YODA	(0.959,	1,	28.948,	65.763,	12.923,	0,	4.907,	1.155,	14.026,	3.933,	0,	1.644,	1.381,	1.072)
UPACK	(0,	0.843,	0,	19.314,	3.902,	0,	0,	1.961,	7.941,	0,	0,	0,	0.941,	0)

5.4 Detection of custom packer

Most of the modern popular malwares are packed by packers. As a counter solution, commercial anti-virus software tends to detect packer signature for detecting and unpacking the packed malware. However, this approach suffers the shortcoming that signatures for packer detection can be easily modified and it is not effective for detecting unknown or custom packers. This is also a major problem since 35% of malware are packed by custom packers [37, 38].

Inspired by [39], a custom packer might be identified by the facts that it contains encrypted or compressed sections which is a strong indication for a packed executable. A packer must also have a bootstrap section containing a stub to decrypt or decompress the encrypted sections. BE-PUM detects the behavior of decrypting as packing/unpacking technique. This indication shows that if the malware contains packing/unpacking techniques, it might be considered as packed by custom packer by BE-PUM. The Universal PE Unpacker[17] plug-in of IDA Pro tends to recognize the two APIs, LoadLibrary and GetProcAddress for unpacking file in generic way. These APIs is mostly used in packers to restore the original executable's imports. BE-PUM detects the employment of these two APIs as 2API technique. In summary, if the malware contains packing/unpacking and 2API techniques, BE-PUM detects it as custom packer.

6. EXPERIMENTS

All experiments are performed on Windows XP built on VMware workstation 10.0. The host OS is Windows 8 Pro with AMD Athlon II X4 635, 2.9 GHz and 8GB memory.

We focus on 12 packers, which are ASPACK v2, CEXE v1.0b, KKRUNCHY v0.23a4, MPRESS v2.19, FSG v2.0, NPACK v1.0, PECOMPACT v2.0, PETITE v2.1, TELOCK v0.99, UPX v3.0, YODA v1.3, and UPACK v0.37-0.39. The experiments are carried out on totally 15031 files in two types of the dataset, manually packed non-malware and real world malware.

6.1 Checking the accuracy with manually packed non-malware

For checking the accuracy of our approach, we apply 12 packers on 418 Windows executables taken from System32 in Windows XP SP3. We obtain 2217 packed files, where 2019 cases are failed to pack. For instance,

- ASPACK reports "*Unrecognized file format*" on `append.exe`, `debug.exe`.
- FSG reports "*FSG does not support* `IMAGE_SUBSYSTEM_NATIVE` *file types*" on `edlin.exe`.
- NPACK causes an error on `dosx.exe`.
- PETITE reports "*Petite cannot compress this file*" on `accwitz.exe`.
- TELOCK reports "*This is no valid PE file*" on `edlin.exe`, `exe2bin.exe`.
- UPX fails to pack a file smaller than 1KB, e.g., `share.exe` (882 bytes).

Table 4 shows the numbers of the successful packing, and BE-PUM correctly identifies the used packer for all of successfully packed files.

Table 4: Data set of manually packed files

Packer	Version	Number of successful packing
ASPack	2	183
CEXE	1.0b	52
FSG	2	348
KKRUNCHY	0.23a4	241
MPRESS	2.19	131
Npack	1	77
PECOMPACT	2	76
PETITE	2	12
TELOCK	0.99	37
UPX	3.94	360
YODA	1.3	308
UPACK	0.37	392

6.2 Packer identification on real malware

We have collected 5374 real malware from the VX Heaven database[18] and 7440 samples from Virusshare[19]. For comparison, each file is scanned by the three popular packer scanners, PEiD, CFF Explorer, and VirusTotal. Unfortunately, we cannot access to the dataset and source code of other generic unpackers [13, 14, 20, 26] for comparison.

PEiD is considered as the most popular signature-based detector for packed files. VirusTotal is a free on-line malware scanner, which combine the results from many AntiVirus sources, e.g., Kaspersky, Microsoft, and AVG. CFF Explorer is also a popular tool, but its database is quite obsolete.

The time limit is set to 1 hour for each file, and the whole experiments take about 10 weeks. Among 12814 examples, BE-PUM reports

- 499 cases with 296 from VX Heaven and 203 from Virusshare cause *timeout*. For instance, `Email-Worm.Win32.Bagle.dw` has 4,294,867,296 iterations in its loop. Currently, BE-PUM simply unholds loops during symbolic execution, which causes *timeout*.
- 5923 with 1419 from VX Heaven and 4504 from Virusshare are detected not packed, consistent with PEiD, CFF Explorer, and VirusTotal.
- 6392 are detected packed.

The details of the 6392 packed files are further observed.

- 5459 with 3270 from VX Heaven and 2189 from Virusshare are detected packed by one of 12 packers, consistent with PEiD, CFF Explorer, and VirusTotal. The details are summarized in Table 5.

Table 5: Detection result on similar cases

Packer	Version	Number of files
ASPack	2	340
CEXE	1.0b	1
FSG	2	880
KKRUNCHY	0.23ab	15
MPRESS	2.19	2
Npack	1	14
PECOMPACT	2	410
PETITE	2	78
TELOCK	0.99	15
UPX	3.94	3432
YODA	1.3	16
UPACK	0.37	256

- 402 with 137 from VX Heaven and 265 from Virusshare are detected packed by one of 12 packers, inconsistent among PEiD, CFF Explorer, VirusTotal, and BE-PUM. Details are analyzed in Section 6.3.
- 325 with 216 from VX Heaven and 109 from Virusshare are classified as packed with *custom* packers, i.e., it contains packing/unpacking and 2API techniques but the frequency vectors match none of 12 packers. PEiD, CFF Explorer and VirusTotal consistently identify their packers i.e., ACProtect 1.3x (11 cases), AHpack 0.1 (1 case), ARM Protector v0.1 (1 case), Armadillo v4.x (142 cases), BJFnt v1.1b (2 cases), CExe v1.0a (3 cases), CreateInstall Stub (1 case), Crunch/PE (25 cases), EncryptPE 1.2003.5.18 (2 cases),

[17]https://www.hex-rays.com/products/ida/support/tutorials/unpack_pe/unpacking.pdf

[18]`http://vxheaven.org`

[19]https://virusshare.com

Table 6: Determination of packer name for real malware

Malware	CFF Explorer	PEiD	VirusTotal	BE-PUM
Backdoor Win32.Rbot.apj	NONE	NONE	UPX	UPX v3.0
Backdoor Win32.VB.yo	NONE	FSG v1.10	NONE	UPX v3.0
Backdoor Win32.Rbot.xf	NONE	FSG v1.10	UPX	UPX v3.0
Trojan-Dropper Win32.Agent.uq	NONE	yoda's Protector v1.02	UPX	UPX v3.0
Trojan-PSW Win32.LdPinch.ei	NONE	Morphine v1.2	UPX	UPX v3.0
Email-Worm Win32.NetSky.ab	PECompact 2.x	PECompact 2.x	PecBundle	PECompact v2.0
Email-Worm Win32.NetSky.ac	PECompact 2.x	PECompact 2.x	NONE	PECompact v2.0
Email-Worm Win32.Brontok.c	NONE	FSG v1.10	MEW	FSG v2.0

Enigma protector 1.10/1.11 (29 cases), EXE Stealth v2.71 (9 cases), EXE32Pack v1.37 (38 cases), EXECryptor 2.2.4 (38 cases), StarForce V3.X (11 cases) and Xtreme-Protector v1.05 (12 cases). These packers are not supported in current BE-PUM.

- 206 with 36 from VX Heaven and 170 from Virusshare are detected packed by BE-PUM as custom packer, while PEiD, CFF Explorer, VirusTotal detect NONE. Details are analyzed in Section 6.3.

6.3 Manual inspection on inconsistency

There are 402 inconsistent examples among results of PEiD, CFF Explorer, VirusTotal, and BE-PUM. We manually investigate all disassembled results by BE-PUM of the 327 examples, and observe that each unpacking code has

- a modified binary signature, and
- the same unpacking code with different offsets, except for different prefixes of less than 5% code.

Table 6 picks up 8 inconsistent samples among 402 examples.

Modified binary signature

The binary signature of UPX v3.0 is compared with the binary prefix of `Backdoor.Win32.Rbot.apj`.

UPX v3.0 (in *CFF explorer*)	`60 BE 00 E0 95 00 8D BE 00 30 AA FF 57`
`Backdoor.Win32.Rbot.apj`	`68 C4 C2 41 00 67 64 FF 36 00 00`

The latter is disassembled as `push 0x41c2c4; push dword fs:[0x0]`. Actually, the disassembly by BE-PUM clarifies that `Backdoor.Win32.Rbot.apj` has extra 8 instructions at the top, which mislead PEiD and CFF Explorer.

```
(a) Backdoor.Win32.Rbot.apj  (b) common prefix of UPX
PUSH 41C2C4
PUSH FS:[0]
MOV FS:[0], ESP
NOP
PUSH 491110
JMP 4922F6
NOP
RET

PUSHA                        PUSHA
MOV ESI, 476000              MOV ESI, 405000
LEA EDI, -479232(ESI)        LEA EDI, -16384(ESI)
PUSH EDI                     PUSH EDI
JMP 491132                   JMP 4052FA
```

Another example `Backdoor.Win32.VB.yo` replaces the first instruction `PUSHA` in the common prefix of the code packed by UPX, with `JMP 42C791`. In both cases, the following unpacking codes are the same except for the differences of offsets.

FSG v2.0	3-5-5-5-5-4-5-5-5-5-5-4-5-5-12
MEW 11 SE v1.1	5-3-5-5-5-4-5-5-5-5-4-5-5-12

Figure 7: Comparison between FSG and MEW

```
(a) Backdoor.Win32.VB.yo      (b) common prefix of UPX
JMP 42C791
                              PUSHA
MOV ESI, 41B000               MOV ESI, 405000
LEA EDI, -106496(ESI)         LEA EDI, -16384(ESI)
PUSH EDI                      PUSH EDI
JMP 42C7B2                    JMP 4052Fa
MOV EBX, (ESI)                MOV EBX, (ESI)
SUB ESI, FFFFFFFC             SUB ESI, FFFFFFFC
ADC EBX, EBX                  ADC EBX, EBX
JB 42C7A8                     JB 42C7A8
MOV AL, (ESI)                 MOV AL, (ESI)
```

One example is malware 01ad1a71389bf3f0ed5ae2558033c5102 7fb750df8c75a2d28119af1d15fc91d 6 (MD5: 990fd836de4291909e 8bdef1c0b55efe) which modifies the second instruction `MOV ESI, 407000` in the common prefix of the code packed by UPX, with `MOV ESI, 26781000`. The instruction `MOV ESI, 407000` takes 5 bytes, BE 00 50 40 00 while `MOV ESI, 26781000` contains the hex value, BE 00 10 78 26. Note that the difference in the last byte between 00 and 26 leads to the failure of detecting this malware. In both cases, the following unpacking codes are also the same except for the differences of offsets.

```
(a) 01ad1a71389bf3f0ed5ae2558033c5  (b) common prefix of UPX
1027fb750df8c75a2d28119af1d15fc91d6
PUSHA                         PUSHA
MOV ESI, 26781000             MOV ESI, 405000
LEA EDI, -65536(EDI)          LEA EDI, -16384(ESI)
PUSH EDI                      PUSH EDI
JMP 2678fA0A                  JMP 4052FA
MOV EBX, (ESI)                MOV EBX, (ESI)
SUB ESI, ffffffC              SUB ESI, ffffffC
ADC EBX, EBX                  ADC EBX, EBX
JB 2678FA00                   JB 4052F0
MOV AL, (ESI)                 MOV AL, (ESI)
```

Unsupported packers in BE-PUM

We emphasize that BE-PUM misunderstands only `Email-Worm.Win32 .Brontok.c`. VirusTotal correctly labels *packed by MEW*, whereas BE-PUM labels *packed by FSG*. The first reason is that the current BE-PUM does not cover MEW, and we further observe that

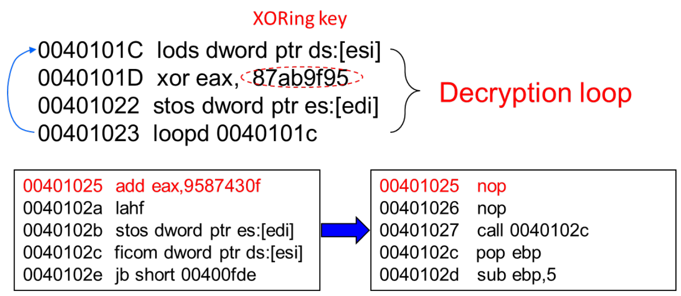
- MEW and FSG use exactly the same set of obfuscation techniques as presented in Figure 7.
- The unpacking codes of MEW and FSG consist of 107 and 86 lines, respectively, and 72 lines among them are shared. Below, the common prefixes of MEW and FSG are compared with the disassembled code of `Email-Worm.Win32.Brontok.c`.

```
(a) Brontok.c            (b) Prefix of MEW        (c) Prefix of FSG
                         JMP 400154
                                                  XCHG 4052D4, ESP
                                                  POPA
                                                  XCHG ESP, EAX
                                                  PUSH EBP
MOV ESI, 42501C          MOV ESI, 40401C
MOV EBX, ESI             MOV EBX, ESI
LODS EAX, DS:[ESI]       LODS EAX, DS:[ESI]
PUSH EAX                 PUSH EAX
LODS EAX, DS:[ESI]       LODS EAX, DS:[ESI]
XCHG EDI, EAX            XCHG EDI, EAX
MOV DL, 80h              MOV DL, 80h
MOVS [EDI], [ESI]        MOVS [EDI], [ESI]        MOVS [EDI], [ESI]
MOV DH, 80h              MOV DH, 80h              MOV DH, 80h
CALL 40012C              CALL 40012C              CALL 4001E8
ADD DL, DL               ADD DL, DL               ADD DL, DL
...                      ...                      ...
```

Note that BE-PUM labels 325 examples with *packed with unknown packers*. PEiD, CFF Explorer and VirusTotal consistently label all of them with different packers from what BE-PUM supports.

Detection of custom packer

BE-PUM detect custom packer on 206 cases while other tools detect NONE. Consider the malware, plt127.bin (MD5: bfdd1b7c6b 501c517c3ae9a3ef03a43f) in the figure below. It is embedded in a malicious PDF file(MD5: aaf8534120b88423f042b9d19f1c59ab). This malware is packed by a custom packer which is unknown to VirusTotal. This malware utilizes the encryption/decryption technique. The encryption/decryption technique occurs at decryption loop from 40101c to 401023 with XORing key 87AB9F95 that modifies the opcodes from 401025 to 4017A5. Thus, the instruction ADD EAX, 9587430F at address 00401025 is converted to NOP. Note that this modification only occurs at run time, allowing the malware to hide the main intension. The malware continues to restore the import tables by calling 2 special APIs LoadLibrary@KERNEL32.dll at 401073 and GetProcAddress@Kernel32.dll at 401097. Since there are packing/unpacking and 2API technique, BE-PUM detects this malware as custom packer.



```
0040101C  lods dword ptr ds:[esi]
0040101D  xor eax, 87ab9f95
00401022  stos dword ptr es:[edi]
00401023  loopd 0040101c
```

```
00401025  add eax,9587430f
0040102a  lahf
0040102b  stos dword ptr es:[edi]
0040102c  ficom dword ptr ds:[esi]
0040102e  jb short 00400fde
```

```
00401025  nop
00401026  nop
00401027  call 0040102c
0040102c  pop ebp
0040102d  sub ebp,5
```

Another example is malware VirusShare_12ac2650ff96a37e602a944 12ba3b757 (MD5: 12ac2650ff96a37e602a94412ba3b757). This malware is detected NONE by VirusTotal, CFF Explorer and PEID. However, it contains the the decryption loop as described below.

```
404623          IMUL    EDI, 343FDh
404629          ADD     EDI, 269EC3h
40462F          MOV     EDX, EDI
404631          SHR     EDX, 10h
404634          XOR     [EAX], DL
404636          INC     EAX
404637          DEC     ESI
404638          JNZ     404623
```

This malware also employs 2API technique for restoring the import table.

```
403DA5          PUSH    OFFSET ProcName
403DAA          PUSH    OFFSET ModuleName
403DAF          CALL    GetModuleHandleA
403DB5          PUSH    EAX
403DB6          CALL    GetProcAddress
```

BE-PUM detects this malware as custom packer.

7. CONCLUSION

This paper proposed the *metadata signature* of packers, as an alternative to the binary signature. It is the frequency vector of the classified obfuscation techniques in the unpacking code of a packed binary. A binary model generator BE-PUM was extended to support the formal criteria to detect obfuscation techniques, which are carefully defined by observing more than 40 real world malware. The chi-square test was applied to decide the likelihood of the metadata signature of a packed code.

Experiments were performed on 12814 real malware with 12 packers. The accuracy of the metadata signature outperforms the state-of-the-art tools, like *PEiD*, *CFF Explorer*, and *Virus Total*, as long as these 12 packers are concerned. Note that extensions to cover more packers are not difficult by giving enough packed examples as the training set. They are easily generated by packing toy programs if we have the same packer.

The drawback is that BE-PUM is quite heavy. We are improving BE-PUM in two views, the algorithms and the implementation. Current BE-PUM simply unholds loops in binary code during symbolic execution. Although most of loops in malware are bounded their iterations by constants, sometimes they are quite huge. We are trying to introduce an automatic generation of loop invariant to BE-PUM. The data structure for the memory model in BE-PUM is also under consideration. We also tried a multi-threaded implementation [24], which requires further investigation.

We are also interested in the ratio of the obfuscation techniques to the program size. Currently, the detection of *unpacked code* depends only on the presence of packing/unpacking code. Adding to the non-existence of the packing/unpacking code, if the ratio is enough small, we can convince more that it is *not packed*.

There are two directions for future work.

- By manual observation, we found that BE-PUM detects near the OEP of the payload. By analyzing the payloads, malware would be characterized in two steps, by the used packer and the payload characteristics.
- The packer identification by BE-PUM is precise, but quite slow. It will be useful to apply on automatic training set generation for statistical methods, like [34].

Acknowledgments

This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.01-2015.16. This work is supported by JSPS KAKENHI Grant-in-Aid for Scientific Research(B) (23300005,15H02684).

8. REFERENCES

[1] John, F. G., Alejandro, F., Richard, L., Victor, L., Biplab, S., Sravana, K., Sristi, L., Logesh, M. and Mangalam, N. The Link between Pirated Software and Cybersecurity Breaches. Retrieved from http://news.microsoft.com/download/presskits/dcu/docs/idc_031814.pdf.

[2] Al-Anezi, M.M.K. Generic packing detection using several complexity analysis for accurate malware detection. In *International Journal Advanced Computer Science*, 5(1), 2014.

[3] Osaghae, E.O. Classifying Packed Programs as Malicious Software Detected. In *International Journal of Information Technology and Electrical Engineering*, Vol. 5, pp 22-25, 2016.

[4] Santos, I., Ugarte-Pedrero, X., Sanz, B., Laorden, C. and Bringas, P.G. Collective classification for packed executable identification. In *Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference*, pp. 23-30, Perth, Australia, 2011.

[5] McAfee The Good, the Bad, and the Unknown available online: http://www.techdata.com/mcafee/files/MCAFEE_wp_appcontrol-good-bad-unknown.pdf (accessed on 21th May 2017).

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

[7] T. Ban, R. Isawa, S. Guo, D. Inoue, K. Nakao. Efficient malware packer identification using support vector machines with spectrum kernel. In *AsiaJCIS*, pp.69-76, 2013.

[8] Yan, W., Zhang, Z., and Ansari, N. Revealing Packed Malware. In *IEEE, Security and Privacy*, Vol. 6, Issue: 5, pp. 65 âĂŞ 69.

[9] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos and Pablo G. Bringas SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Journal of 2015 IEEE Symposium on Security and Privacy*, pp.659 - 673

[10] T. Ban, R. Isawa, S. Guo, D. Inoue, K. Nakao. Application of string kernel based support vector machine for malware packer identification. In *IJCNN*, 2013

[11] G. Bonfante, J. Fernez, J.-Y. Marion, B. Rouxel, F. Sabatier, A. Thierry. Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *ACM SIGSAC CCS*, pp.46-53, 2015.

[12] F. Guo, P. Ferrie, T. Chiueh. A Study of the Packer Problem and Its Solutions. in *RAID*, pp.98-115, 2008 LNCS 5230.

[13] R. Isawa, M. Kamizono, D. Inoue. Generic Unpacking Method Based on Detecting Original Entry Point. In *NIP*, pp.593-600, 2013. LNCS 8226.

[14] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, H. Lee. Generic Unpacking using Entropy Analysis. In *Malware*, pp.114-121, 2010.

[15] K. Kancherla, J. Donahue, S. Mukkamala. Packer identification using Byte plot and Markov plot. Journal of Computer Virology and Hacking Techniques, 12(2), pp.101-111, 2016.

[16] M. G. Kang, P. Poosankam, H. Yin. Renovo: a hidden code extractor for packed executables. In *ACM WORM*, pp.46-53, 2007.

[17] J. Kinder D. Kravchenko. Alternating control flow reconstruction. In *VMCAI*, pp.267-282, 2012. LNCS 7148.

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

[19] J. Kinder. *Static Analysis of x86 Executables.* PhD thesis, Technische Universitat Darmstadt, 2010.

[20] L. Martignoni, M. Christodorescu, S. Jha. OmniUnpack: Fast, Generic, Safe Unpacking of Malware. In *ACSAC*, pp.431-441, 2007.

[21] M. Morgenstern, A. Marx. Runtime packer testing experiences. In *CARO*, pp.288-305, 2008. LNCS 6174.

[22] M. H. Nguyen, T. B. Nguyen, T. T. Quan, M. Ogawa. A hybrid approach for control flow graph construction from binary code. In *IEEE APSEC*, pp.159-164, 2013.

[23] M. H. Nguyen, M. Ogawa, T. T. Quan. Obfuscation code localization based on CFG generation of malware. In *FPS*, pp.229-247, 2015. LNCS 9482.

[24] M. H. Nguyen, T. T. Quan, D. A. Le. Multi-threaded On-the-Fly Model Generation of Malware with Hash Compaction. In *ICFEM*, pp.159-174, 2016.

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

[26] P. Royal, M. Halpin, D. Dagon, R. Edmonds, W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *ACSAC*, pp.289-300, 2006.

[27] V. S. Sathyanarayan, P. Kohli, B. Bruhadadeshwar. Signature Generation Detection of Malware Families. In *ACISP*, pp.336-349, 2008. LNCS 5107.

[28] Mutz, Darren and Valeur, Fredrik and Vigna, Giovanni and Kruegel, Christopher Anomalous System Call Detection. In *ACM Trans. Inf. Syst. Secur.*, Vol. 9, no. 1, pp. 61-93, February. 2006.

[29] F. Maggi, M. Matteucci and S. Zaneroin. Detecting Intrusions through System Call Sequence and Argument Analysis. In *IEEE Transactions on Dependable and Secure Computing*, Vol. 7, no. 4, pp. 381-395, Dec. 2010.

[30] M. Shafiq, S. Tabish, M. Farooq. PE-Probe: leveraging packer detection and structural information to detect malicious portable executables. In *VB*, pp.29-33, 2009.

[31] D. Song, et al. Bitblaze: A new approach to computer security via binary analysis. In *ICISS*, pp.1-25, 2008. LNCS 5352.

[32] F. Song, T. Touili. Pushdown model checking for malware detection. In *TACAS*, pp.110–125, 2012. LNCS 7214.

[33] F. Song, T. Touili. LTL model-checking for malware detection. In *TACAS*, pp.416–431, 2013. LNCS 7795.

[34] L. Sun, S. Versteeg, S. Boztas, T. Yann. Pattern recognition techniques for the classification of malware packers. In *ACISP*, Berlin, pp.370-390, 2010. LNCS 6168.

[35] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, T. W. Reps. Directed proof generation for machine code. In *CAV*, pp.288–305, 2010. LNCS 6174.

[36] R. David, S. Bardin,T.D. Ta, J. Feist, L. Mounier, M.-L. Potet, J.-Y. Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis. In *SANER*, pp.653–656, 2016.

[37] M. Morgenstern and H. Pilz Useful and useless statistics about viruses and anti-virus programs. In *Proceedings of the CARO Workshop 2010*, pp.653–656, 2010.

[38] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. IEEE Symp. Security and Privacy (S&P)*, 2015.

[39] Deep Instinct Research Team Certificate Bypass: Hiding and Executing Malware from a Digitally Signed Executable. In *Proceedings of Black Hat USA 2016*, August 2016.