A STATISTICAL APPROACH FOR PACKER IDENTIFICATION

Nguyen Minh Hai, Quan Thanh Tho*

Faculty of Computer Science and Engineering, HCM City University of Technology
268 Ly Thuong Kiet Street, Ward 14, District 10, Ho Chi Minh city, Vietnam

*Email: *qttho@cse.hcmut.edu.vn*

Received: 15 June 2016; Accepted for publication: 28 July 2016

ABSTRACT

Most of modern malware are packed by packers which automatically generate a lot of obfuscation techniques to defeat the anti-virus software. To identify packer, most of industry approaches still adopt the well-known technique of signature matching which can be easily evaded. This paper studies the new approach of applying a statistical approach to tackle this problem. We propose a new weight for extracting what obfuscation techniques might be more favourable in packers. We call it obfuscation technique frequency-inverse packer frequency ($otf°ipf$). As the term implies, $otf°ipf$ calculates values for each obfuscation techniques in a packer through an inverse proportion of the frequency of the obfuscation technique in a particular packer to the percentage of packers the obfuscation technique appears in. Obfuscation techniques with high $otf°ipf$ value show a strong relationship with the packer they appear in. Based on this weight, packer is represented by a vector of $otf°ipf$. Then the used packer is identified by measuring the similarity between vectors of packer and targeted file. For checking the accuracy of our approach, we have performed the experiments of identifying packer on 200 real-world malware for comparing between our approach with the binary signature technique adopted in CFF Explorer. The result shows that our technique produces the better detection.

Keywords: concolic testing, packer, malware analysis, tf-idf, obfuscation techniques.

1. INTRODUCTION

Modern popular malwares are packed or obfuscated by packer to generate new invariants. According to [1, 2], 80 % of malware is packed by many kinds of packers. One popular example is EMDIVI virus, a notorious advanced persistent thread (APT) targeting on many organizations in Japan e.g. government agencies, local governments, banks and universities. This malware exploited the packer UPX.

The main goal of packer is to defeat the signature based technique of anti-virus softwares. It also increases the difficulty of the reverse engineering process since it often takes a very long time for unpacking or decrypting a packed file. As a counter solution, most of anti-virus software tends to detect packer signature for verifying the malware. Since malware can obfuscate the packer signature, this technique can be easily defeated. In general, the phase of identifying and unpacking packer takes a very important step.

In general, we assume that each packer P can be represented a vector O of many obfuscation techniques $O = (O_1, O_2, \ldots, O_k)$ where O_i is frequency of obfuscation technique i^{th}. Table 1 depicts the frequency of obfuscation techniques in each packer. For example, with packer APSACK, the frequency of Checksumming, CodeChunking and Indirect Jump are 49, 1 and 12 respectively. However, when a malware F is packer by packer P, the frequency of obfuscation technique O_i is not exactly the same with the value of Table 1. The reason is that malware can accidently adopts these obfuscations itself which contaminates the frequency. Hence, to determine whether F is packed by P, the method of using the exact matching of frequency produces the low accuracy.

For solving this problem, we propose a method as follows.

- We develop a new weight $otf°ipf$ to determine the relevance of obfusaction technique O in packer P. Based on this weight, a packer is represented by a vector of $otf°ipf$ known as V_p.
- For a test file F, we construct the vector of $otf°ipf$, known as V_F. Then, by calculating the similarity between two vectors V_P and V_F, we can determine whether F is packed by P.

In this paper, we introduce a new approach of applying statistical approach to identify packers. Our key contributions are summarized as follows.

- We have extended our tool, BE-PUM [4, 5] as a generic model generator with stubs of detecting obfuscation techniques. During the on-the-fly model generation, BE-PUM calculates the frequency of obfuscation techniques in each packer.
- We propose a new weight, obfuscation technique frequency-inverse packer frequency ($otf°ipf$) for measuring the favourite obfuscation techniques in packer. Based on this weight, we construct a vector and calculate the similarity between targeted file and packer for packer identification.
- We perform the experiments on 200 real malware for checking the effectiveness of our approach.

The rest of this paper is organized as followed. Section 2 briefly describes the background knowledge of BE-PUM and packer. Section 3 describes the high-level overview of our method. Section 4 shows our experiments on 200 real-world malwares mainly collected from VirusTotal. The final section 5 discusses conclusion of our paper and some future works.

Related works: There are many binary analysis tools, e.g. JakStab [6, 7], Syman [8] and BINCOA/OSMOSE [9], CodeSurfer/x86 [10], McVeto [11] and a commercial product, IDA Pro which is claimed to be one of the most popular and powerful tools for binary code analysis. However, to the best of our knowledge, we are not aware of existing research on packer identification.

In industry, the approach of packer detection focuses on recognizing the occurrence of packer in targeted file. The main technique for this goal is to detect the packer signature often located in original entry point. This approach is used in many softwares, e.g., PEID and CFF Explorer. However, this technique can be easily evaded since malware can mutates the packer signature.

2. PRELIMINARIES

In this section, we present the basic concept of packers and BE-PUM.

2.1. Packer

Packer is a software that can mutate a binary file into another executable. The new executable preserves the original file's functionality, but has a different content on the system for preventing the process of linking between them. Packers are used on executable for mainly two reasons: to reduce the size of binary file, and to evade analysis, reverse engineering, or detection. For the first reason, packer minimizes targeted file by compressing its content and then uncompressing it on-the-fly during the execution. This first feature of a packer is very popular in the scenario where the size of programs is large and minimizing file size is a critical task in early days of computer. However, existing real-world packers are used mainly for the second reason, i.e. to protect the original file from being observed, analyzed and tampered with. For achieving this goal, packer combines many obfuscation methods which include anti-debugging, anti-cracking, anti-tracing, anti-reverse engineering, and more for preventing target file from straightforward analysis. These packers are used for protecting the licensed softwares or games from crackers. However, this feature is also exploited in malware for protecting them from detection of anti-virus software.

From [1], 80 % of malwares use packing techniques for evading the detection. A popular approach to identify a packer is by a packer signature, which is a binary pattern of a part of a file. Similar to the binary signature matching in many commercial anti-virus software, it identifies a packer by finding a statistical binary pattern matching in packer signatures, as in well-known tools, e.g. CFF Explorer. However, when the technique of self-modification mutates the code layout to obfuscate the location of the header, it fails similar to commercial anti-virus software on polymorphic virus.

2.1.1 Obfuscation techniques

Packer supports many obfuscation techniques making binary code very difficult to explore. Inspired by [12], we categorize them into 6 groups as follows.

- *Entry/code placing obfuscation (Code layout)*: overlapping functions, overlapping blocks, and code chunking.
- *Self-modification code*: Dynamic code which includes overwriting and packing/unpacking.
- *Instruction obfuscation*: Indirect jump.
- *Anti-tracing*: SEH (structural exception handling) and Special API (LoadLibrary@kernel32.dll and GetProcAddress@kernel32.dll)
- *Arithmetic operation*: Obfuscated constants and checksumming.
- *Anti-tampering*: Checksumming, timing check, anti-debugging, anti-rewriting, and hardware breakpoints. Anti-rewriting technique includes stolen bytes and checksumming

2.2. BE-PUM

We have been developing a tool BE-PUM [4, 5] (Binary Emulator for Pushdown Model generation), for generating a precise control flow graph (CFG) against obfuscation techniques of malware, e.g., indirect jump, self-modification, overlapping instructions, *structured exception handler* (SEH) and many obfuscation techniques adopted in packer.

2.2.1. The framework of BE-PUM

BE-PUM implements CFG reconstruction based on concolic testing. Figure 1 shows the architecture of BE-PUM including three components: symbolic execution, binary emulation, and CFG storage. It computes a single step disassembly by applying JakStab 0.8.3 [10, 11] as a preprocessor . An SMT Z3.4.4 is supported as a backend engine to generate a test instance for concolic testing. The symbolic execution picks up state from the frontier and extends in on-the-fly manner.

2.2.2 Running Example

We illustrate the operation of BE-PUM with a small example in Figure 2. At a first look, the execution follows the looping path *P = (start → 0 → 1 → 2 → 3 → 4 → 1)*. However, the instruction at the location 3 overwrites the opcode at *L1 + 1* which modifies the opcode at 1 from *EB 00* to *EB 0A*. The instruction "jmp *L2"* at 1 is modified to *"jmp L3"*. JakStab and IDA Pro fail to handle this obfuscation technique, whereas BE-PUM correctly generates
(0, "xor eax eax") → (1, "jmp L2") → (2, "mov eax, offset l1 + 1") → (3, "mov byte ptr [eax], 0Eh") → (4, "jmp L1") → (1, "jmp L3") → · · ·

Continue from the location *5*, there is a system call GetModuleHandleA at *9* and an indirect jump at *12*. The API GetModuleHandleA at *9* is invoked with parameter *0*. BE-PUM simulates its symbolic execution using JNA. The return value is stored in register *eax*.

The path formula of *(start → 0 → 1 → 2 → 3 → 4 → 1 → 5 → 6 → 8 → 9 → 10 → 11 → 12)* is *(ebx == 1000)*. For handling the indirect jump at *12*, BE-PUM adopts concolic testing by setting the value *(ebx = 1000)* (generated by Z3 4.3), and finds a new destination *14* (*13* is dead node).

2.2.3 BE-PUM as a generic unpacker tool

Preliminary version of BE-PUM can handle some typical obfuscation techniques of packers, e.g., indirect jump, self-modification, overlapping instructions, and structured exception handler (SEH). Inspired by [6], we have implemented many counter solutions for obfuscation techniques of packers which improves BE-PUM as a powerful general unpacker. Since most of malwares work in user mode, BEPUM just support user process level. It also supports symbolic binary emulation which makes BE-PUM a very effective de-obfuscation tool. We consider the SEH technique adopted in packer PETITE.

```
404116    PUSH 4022E3
40411B    PUSH DWORD PTR FS:[0]
404122    MOV DWORD PTR FS:[0] , ESP
……….
40421E    MOV BYTE PTR DS:[EDI] , AL
```

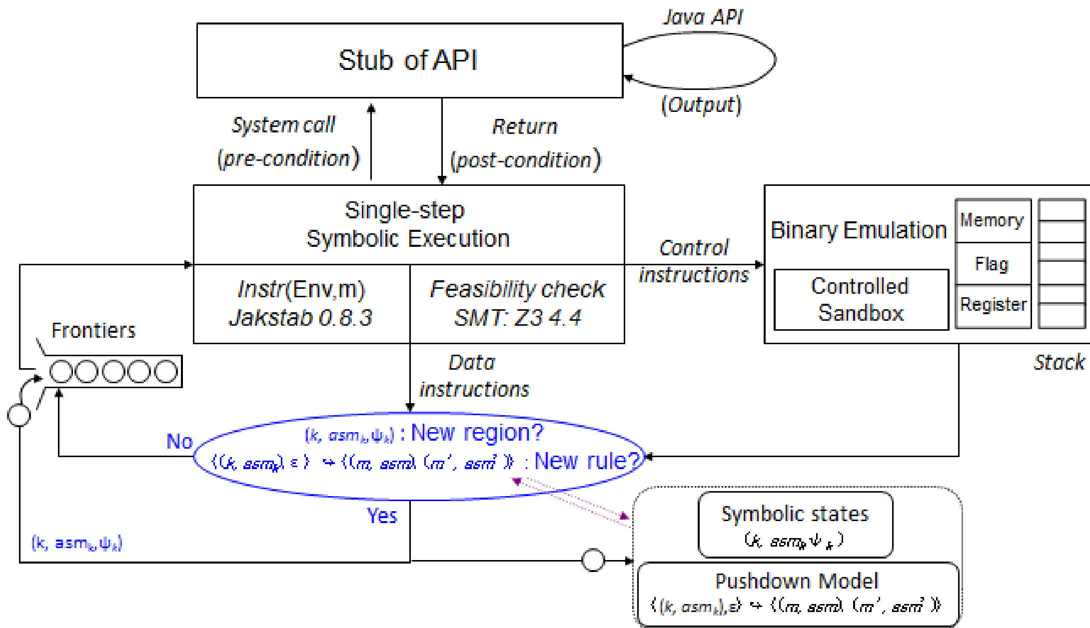

Figure 1. The framework of BE-PUM.

At *00404122*, *fs:[0x0]* is overwritten with the pointer to malicious code (the value of esp). It then creates a fault condition by "*mov*" at *0040421E*. Since the value of register *EDI* is *0*, the instruction at *0040421E* overwrites the memory address at *00000000*, which is protected by Windows operating system. This exception changes the control flow to *4022E3*. BE-PUM can precisely trace this obfuscation technique while other tools fail.

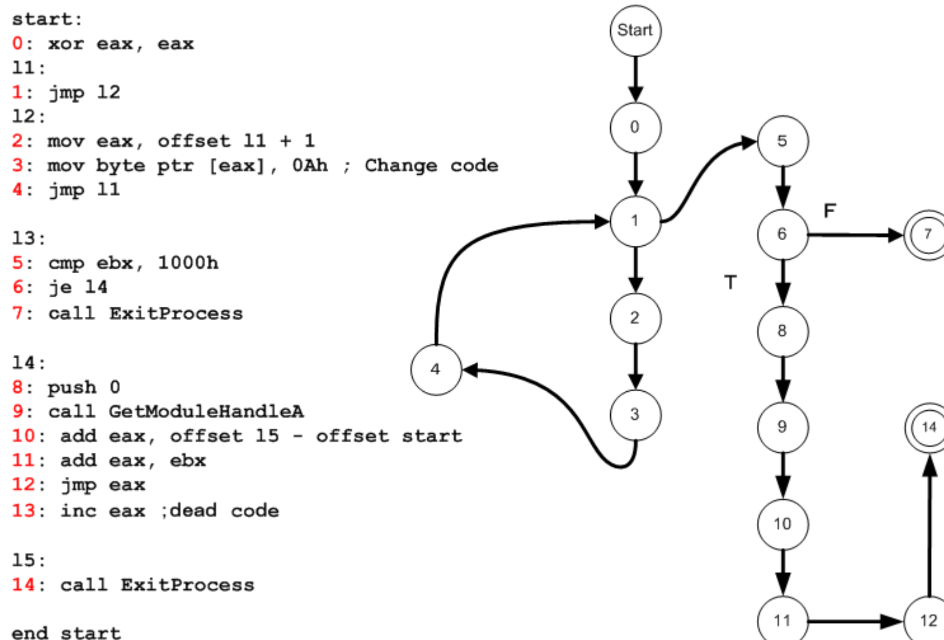

Figure 2. Running example of BE-PUM.

Current version of BE-PUM can unpack 25 different packers. The details are presented in Table 1.

Table 1. Frequency of obfuscation techniques in packer.

Packer	AntiDebugging	Checksumming	CodeChunking	IndirectJump	ObfuscatedConst	Overlapping Block	Overlapping Function	Overwriting	Packing/Unpacking	SEH	Stolen Byte	Timing Check	Special API	Hardware BPX
.BJFnt v1.3	0	3	937	0	40	208	0	57	0	0	0	0	0	0
ASPack v2.12	0	49	1	12	63	8	0	10	3	0	2	0	2	0
Exe Stealth 2.75a -> WebtoolMaster	1	1	5	15	57	1	0	23	0	3	0	0	3	0
EXEPACK	0	5	0	4	25	1	0	0	0	0	0	0	1	0
eXPressor	0	49	3	1	89	17	0	4	2	0	1	0	1	0
FSG v2.0	0	1	1	12	2	1	1	1	2	0	0	0	1	0
LamCrypt v1.0 -> LaZaRuS	0	0	0	7	7	0	0	7	0	0	0	0	0	0
MEW 11 SE v1.1	0	1	0	10	2	4	0	0	2	0	0	0	1	0
MPESS	0	14	2	2	34	3	0	12	0	0	0	0	1	0
NoodleCrypt v2.00 (Eng) -> NoodleS	0	0	76	19	47	78	10	0	4	0	0	0	9	0
nPack v1.0	0	6	0	2	24	5	16	5	3	0	2	0	3	0
PECompact 2.0x Heuristic Mode -> Je	0	1	1	16	60	8	0	11	10	1	2	0	2	0
Pencrypt	0	1	4	2	23	0	0	9	0	0	0	0	0	0
PEtite v2.1	0	57	2	8	115	16	0	15	7	2	0	0	1	0
RLPack	0	3	2	9	19	9	15	8	2	0	1	0	1	0
SafeDisc v4	0	0	1	2	5	1	0	2	0	0	0	0	0	0
Sramble	0	1	0	11	2	4	1	0	2	0	0	0	1	0
tElock 0.99 - 1.0 private -> tE!	0	18	24	22	167	105	5	0	6	20	2	0	1	0
Upack V0.37-V0.39 -> Dwing	0	10	1	19	19	1	2	3	2	0	0	0	1	0
UPX v3.0	0	5	0	4	25	1	0	0	0	0	0	0	1	0
WinUpack	0	10	1	19	19	1	2	3	2	0	0	0	1	0
WWPack32 v1.00,	0	0	1	0	13	0	0	2	2	0	0	0	0	0
Xcomp	0	36	3	7	37	18	16	6	1	0	1	0	1	0
y0da's Crypter v1.2	1	1	6	15	37	2	0	21	0	1	0	0	3	0
yoda's Crypter 1.3 -> Ashkbiz Danehl	1	1	8	15	141	5	0	24	0	3	0	0	3	0

3. OUR STATISTICAL METHOD

3.1. Power law

In statistics, a *power law* [13] is a relationship function between two quantities. When there is a relative change in one quantity, the other quantity varies proportional to the power. By determining suitable power proportion, one can generate a lot of different *power law functions*. Power law has been used widely in many fields. Among them, the most notorious application is *Pareto principle* [14], known as the 80:20 rule, which is very famous in industry. The proportional relationship between A and B of Pareto Principle is formulated as.

$$A:B = \left(\frac{1+H}{2}\right):\left(\frac{1-H}{2}\right)$$

Through changing the value of H, we can determine the appropriate proportion between A and B.

In addition, power law is also used in other fields e.g. physical, biological, and even criminal charges per convict. For computer science, the most important application of power law is *Zipf's law* [15]. Zipf's law usually refers to the “size” y of an occurrence of an event relative to its rank r. That is, the higher occurrence even y has, the higher rank y is belonged to. Zipf’s law is used widely in text processing, where “size” denotes the frequency of use of the word a text, and “rank” is the importance of this word. Zipf's law states that the size of the rth largest occurrence of the event is inversely proportional to its rank: $y \sim r^{-b}$, with b can be determined in many ways.

In this paper, we adapt Zifp's law to introduce a new index for determining the obfuscation technique relevance in packer. Using the new index, we then can generate a new vector to represent a packer. This representation vector will be adopted to determine whether a malware is packed by packer.

3.2. The $otf°ipf$ weight

Given a packer T, we denote $P = \{P_1, P_2, ..., P_n\}$ a list of malwares which are packed by T, a set of obfuscation techniques O = (O_1, O_2, …, O_k) and a set of normal file $NP = \{NP_1, NP_2, \dots, NP_m\}$, the $otf°ipf$ weight of a certain obfuscation technique O_i is defined as follows.

- Frequency of an obfuscation technique O in a binary B

$$otf(O,B) = \frac{f(O,B)}{f(O_{max},B)} \quad (1)$$

where B can be either a packer p_i or program NP_i. $f(O,B)$ is total number of times O occurs in B and O_{max} is the obfuscation technique which has largest frequency in B.

- Inverse-packer frequency of an obfuscation technique

$$ipf(O) = \frac{|NP|}{0.001*|NP|+occ(O,NP)} \quad (2)$$

where occ is the number of files in NP that O occurs.

- Then

$$otf°ipf(O,B) = otf(O,B) * ipf(O) \quad (3)$$

The intuition of the $otf°ipf$ weight is that the more times packer P uses the obfuscation technique O, the more important O is to P. But if O is also used in many normal programs, the importance of O will reduce. In the ipf formula, the factor $0.001 * |NP|$ is an adjustment to avoid division-by-zero.

Based on $otf°ipf$ weight, we propose a method of identifying packer which is described as follows.

Step 1: For a target file F, generate the vector V_F

$$V_F = \{otf°ipf(O_1,F), otf°ipf(O_2,F), \dots, otf°ipf(O_k,F)\} \quad (4)$$

Step 2: Generate the vector V_T of packer T.

$$V_T = \left\{\frac{\sum_1^n otf°ipf(O_1,P_i)}{n}, \frac{\sum_1^n otf°ipf(O_2,P_i)}{n}, \dots, \frac{\sum_1^n otf°ipf(O_k,P_i)}{n}\right\} \quad (5)$$

Step 3: If the Euclidean distance between these two vectors V_F and V_T is below the threshold ε, we identifies that F is packed by T.

$$\mathrm{d}(V_F, V_T) \le \varepsilon \quad (6)$$

with $\varepsilon = 0.001$. This value is chosen based on empirical study.

3.3. Running Example

We consider 5 examples. Among them, 3 files (Demo1, Demo2 and Demo3) are packed by UPX and the others (Demo4 and Demo5) are normal programs i.e. these files are not packed. The frequencies of 5 files are described in Table 2.

We examine the *Indirect Jump* (IJ) technique in Demo1. By applying the formulas (1) and (2) in *3.2*, we calculate the following results.

$$otf(IJ, Demo1) = \frac{f(IJ,Demo1)}{f(O_{max},Demo1)} = \frac{4}{25} = 0.16$$

$$ipf(IJ) = \frac{|NP|}{0.001*|NP|+occ(IJ,NP)} = \frac{2}{0.001*2+1} = 1.996$$

Then $otf°ipf(IJ, Demo1) = 0.16 * 1.996 = 0.31936$

Table 2. Frequency of obfuscation techniques in running examples.

Name	AntiDebugging	Checksumming	CodeChunking	IndirectJump	ObfuscatedConst	Overlapping Block	Overlapping Function	Overwriting	Packing/Unpacking	SEH	Stolen Byte	Timing Check	Special API	Hardware BPX
Demo1	0	5	0	4	25	1	0	0	0	0	0	0	1	0
Demo2	0	4	0	4	23	1	0	0	0	0	0	0	1	0
Demo3	0	4	0	4	24	1	0	0	0	0	0	0	1	0
Demo4	0	0	1	1	0	0	0	1	0	0	1	0	0	0
Demo5	0	1	0	0	1	0	0	0	1	0	0	0	0	0

The results of $otf°ipf$ in each file are depicted in Table 3.

Table 3. otf°ipf value of obfuscation techniques in running examples.

Name	Value of otf°ipf													
	AntiDebugging	Checksumming	CodeChunking	IndirectJump	ObfuscatedConst	Overlapping Block	Overlapping Function	Overwriting	Packing/Unpacking	SEH	Stolen Byte	Timing Check	Special API	Hardware BPX
Demo1	0	0.392157	0	0.313725	1.960784	0.078431	0	0	0	0	0	0	0.078431	0
Demo2	0	0.313725	0	0.313725	1.803922	0.078431	0	0	0	0	0	0	0.078431	0
Demo3	0	0.313725	0	0.313725	1.882353	0.078431	0	0	0	0	0	0	0.078431	0
Demo4	0	0	0	0.078431	0	0	0	0	0	0	0	0	0	0
Demo5	0	0.078431	0	0	0.078431	0	0	0	0	0	0	0	0	0

Then the vector of packer V_{UPX} = {0, 0.339869281, 0, 0.31372549, 1.882352941, 0.078431373, 0, 0, 0, 0, 0, 0, 0.078431373, 0}.

We consider a target file *F*, V_F = {0, 0.338859211, 0, 0.302725961, 1.872302442, 0.06343149, 0.03, 0, 0, 0, 0, 0, 0.075451373, 0}.

Applying (6), since $d(V_{UPX}, V_F) = 0.633527347 > \varepsilon$ then *F* is not packed by *UPX*.

4. EXPERIMENTAL RESULTS AND DISCUSSION

4.1. Calculating the vector of $otf°ipf$ in each packers

We performed the experiments for 7 packers, UPX v3.0, ASPACK v2, FSG v2.0, NPACK v1.0, PECOMPACT v2.0x, YODA v1.3 and TELOCK 0.99. For other packers, we could not obtain enough samples. Table 4 below shows the numbers of the training set and normal program set for each packer. Note that the training set is taken from packed samples and real-world malware which are known to be packed with the specified packer. Normal program means that this program is not packed by the specified packer.

Table 4. Number of malware for experiments.

Packer Name	Training Set (Numbers)	Set of Normal Program (Numbers)
ASPACK v2	80	22
FSG v2.0	84	20
NPACK v1.0	71	22
PECOMPACT v2.0	85	20
TELOCK v0.99	76	18
UPX v3.0	78	20
YODA v1.3	88	25

The vector of each packer is presented in Table 5.

Table 5. Vector of otf°ipf value in each packer.

Name	Value of otf°ipf													
	AntiDebugging	Checksumming	CodeChunking	IndirectJump	ObfuscatedConst	Overlapping Block	Overlapping Function	Overwriting	Packing/Unpacking	SEH	Stolen Byte	Timing Check	Special API	Hardware BPX
ASPACK v2	0	0.43478	0	1.3913	0.47826	0.65217	0	0.13043	0	0	0	0	0.6087	0
FSG v2.0	0	0.72222	0.83333	0.05556	1.44444	0.55556	0	0.33333	0	0	0.44444	0	0.16667	0
NPACK v1.0	0	0.09091	0	0.27273	0.54545	0	0.18182	1.90909	0	0	0	0	0	0
PECOMPACT v2.0	0	0.18182	0	1.72727	0.18182	0	0.81818	0.81818	0	0	0.27273	0	0.63636	0
TELOCK v0.99	0	0.33333	0.38889	0.72222	1.77778	0.83333	0.33333	0.77778	0	0	0.27778	0.22222	0.77778	0
UPX v3.0	0	0.4827312	0	0.331721	1.939892	0.081828	0	0	0	0	0	0	0.082713	0
YODA v1.3	0.09091	0.63636	0.72727	0	1.09091	0.27273	0.72727	0.72727	0	0.45455	0.18182	0.18182	0.63636	0

4.2. Experiment on real-world malware

4.2.1. Experimental setup

All experiments in this section are performed on Windows XP with AMD Athlon II X4 635, 2.9 GHz and 8GB memory. We perform experiments of the packer identification on 200 real-world malwares mainly collected from VirusTotal. According to the scan results of Virus Total Web service, 78 of these samples are downloaders, 90 are worms, and the rest are trojans.

4.2.2. Experimental results

The packer identification of our approach succeeds on 200 malware. For checking the accuracy of our approach, we also compare our results with the binary signature technique. Table 6 presents the results of packer identification in BE-PUM. Clearly, our approach produces the better results compared with the method of binary signature using CFF Explorer. Although the differences between the binary signature and our approach are not very significant, there are some special cases worth mentioning.

Table 6. Experimental results.

Packer Name	Our approach	Binary Signature
ASPACK v2	14	10
FSG v2.0	12	9
NPACK v1.0	21	11
PECOMPACT v2.0	14	12
TELOCK v0.99	33	13
UPX v3.0	92	83
YODA v1.3	14	10

Consider malware 034f9d2dc5627296141bb7d0a22032b1e8c7e47f266ada4a1da7f8dad05668b. Its binary code is "*60 BE 00 E0 95 00 8D BE 00 30 AA FF C7*" which is disassembly as follows.

```
00A31B20 >  60           PUSHAD
00A31B21   . BE 00E09500    MOV ESI,0034f9d2.0095E000
00A31B26   . 8DBE 0030AAFF  LEA EDI,DWORD PTR DS:[ESI+FFAA3000]
00A31B2C   . C787 D0566200 >MOV DWORD PTR DS:[EDI+6256D0],2A8CFF11
```

From the database of CFF Explorer, the signature of UPX 3.0 is "*60 BE ?? ?? ?? 00 8D BE ?? ?? ?? FF 57"* with ' ??' indicates a wild card. Obviously, the two binary codes differ at the final byte, *C7* vs *57* which defeats the binary signature technique. However, our approach can detect this case.

5. CONCLUSIONS

This paper proposes a new approach on packer identification using statistical approach. We first develop BE-PUM as a generic unpacker tool with the obfuscation technique detection. During on-the-fly disassembly, BE-PUM measures the frequency of obfuscation techniques and extracts the obfuscation technique relevance in packer based on the new weight $otf°ipf$. Using the vector of $°ipf$, we can calculate the distance for identifying packers. Experiment was performed on 200 malware collected from VirusTotal with 7 targeting packers to compare between our approach with the traditional approach. The accuracy of our approach outperforms the state-of-the-art tool CFF Exprolore. Although our experiment only covers 7 packers, it can be easily extended to cover other packers. It is the first future work. A main drawback is that our tool, BE-PUM is quite heavy which produces the slow processing time. Other future work is that we will improve the performance of our tool with multi-threaded implementation.

Acknowledgements. This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.01-2015.16.

REFERENCES

1. BitDefender - Anti-virus technology whitepaper, Technical report, Washington, DC. USA, 2007.

2. Morgenstern M., and Marx A. - Runtime packer testing experiences, Proceedings of the 2nd Computer Antivirus Research Organization Workshop, Hoofddorp, Netherlands, 2008, 288-305.
3. Sathyanarayan V. S., Kohli P., and Bruhadeshwar B. - Signature Generation and Detection of Malware Families, Proceedings of the 13th Australasian Conference of the ACISP, Wollongong, Australia, 2008, 336-349.
4. Nguyễn M. H., Quản T. T., and Ogawa M. - A hybrid approach for control flow graph construction from binary code, Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC), Bangkok, Thailand, 2013, 159-164.
5. Nguyễn M. H., Ogawa M., and Quản T. T. - Obfuscation code localization based on CFG generation of malware, Proceedings of the 8th International Symposium on Foundations and Practice of Security, Clermont-Ferrand, France, 2015, 229-247.
6. Kinder J., Zuleger F., and Veith H. - An abstract interpretation-based framework for control flow reconstruction from binaries, Proceedings of the 10th International Conference of the VMCAI, Savannah, GA, USA, 2009, 214–228.
7. Kinder J., and Kravchenko D. - Alternating control flow reconstruction. Proceedings of the 13th International Conference of the VMCAI, Philadelphia, PA, USA, 2012, 267–282.
8. Izumida T., Futatsugi K., and Mori A. - A generic binary analysis method for malware, Proceedings of the 5th International Workshop on Security, Kobe, Japan, 2010, 199–216.
9. Bardin S., Herrmann P., Leroux J., Ly O., Tabary R., and Vincent A. - The BINCOA framework for binary code analysis, Proceedings of the 23rd International Conference of the CAV, Snowbird, USA, 2011, 165–170.
10. Balakrishnan G., Reps T. W., Lal A., Lim J., Melski D., Gruian R., Yong S. H., Chen C. H., and Teitelbaum T. - Model checking x86 executables with codesurfer/x86 and wpds++, Proceedings of the 17th International Conference of the CAV, Edinburgh, UK , 2005, 158–163.
11. Thakur A. V., Lim J., Lal A., Burton A., Driscoll E., Elder M., Andersen T., and Reps T. W. - Directed proof generation for machine code, Proceedings of the 22nd International Conference of the CAV, Edinburgh, UK, 2010, 288–305.
12. Roundy K. A., and Miller B. P. – Binary code obfuscations in prevalent packer tools, ACM Comput. Surv. **46** (1) (2013) 1-32.
13. Manfred S. F., and Chaos K. - Power Laws: Minutes from an Infinite Paradise. W.H. Freeman and Company, New York, USA, 1991.
14. Rosen K. T., Resnick M. - The size distribution of cities, an examination of the Pareto law and primacy, Journal of Urban Economics **8** (2) (1980) 165–186.
15. Zipf G. - The Psycho-Biology of Language: An Introduction to Dynamic Philology, Boston, USA, 1935.