GSTF Journal on Computing (JOC)
DOI: 10.5176/2251-3043_4.4.354
ISSN:2251 - 3043 ; Volume 5, Issue 1; 2016 pp. 61-68
© The Author(s) 2016. This article is published with open access by the GSTF.

Toward an Approach on Probability Distribution for Polymorphic Malware Analysis

Nguyen Minh Hai
Department of Software Engineering
Ho Chi Minh City University of Technology, Vietnam
Email: hainmmt@cse.hcmut.edu.vn

Ha Minh Ngoc
Department of Software Engineering
Eastern International University, Vietnam
Email: ngoc.ha@eiu.edu.vn

Nguyen Thien Binh
Department of Software Engineering
Ho Chi Minh City University of Technology, Vietnam
Email:551105019@stu.hcmut.edu.vn

Quan Thanh Tho
Department of Software Engineering
Ho Chi Minh City University of Technology, Vietnam
Email: qttho@cse.hcmut.edu.vn

***Abstract*— Nowadays, computer security is a serious issue which attracts the interest from many nations. To identify malware, most of industry approaches still center the well-known technique of signature matching. However, modern polymorphic malwares use *packer* to obfuscate their malicious actions. A sophisticated packer can generate virtually variants of a viral code, making the signature-based technique easily defeated.**

Naturally, applying stochastic approach prompts a potential solution to handle polymorphic virus. This paper studies an approach of applying probability distribution for tackling the two important problems in analyzing polymorphic malware, which are to identify a potential malware and to detect packer which malware adopts. For the first goal, we derive a new frequency-based weight to identify most specific instructions for each malware family, known as *instruction frequency-inverse malware frequency* ($if^{\circ}imf$) . For the second one, we propose a new term, *obfuscation technique frequency-inverse packer frequency* ($otf^{\circ}ipf$) for evaluating the importance of obfuscation techniques in packers. We have performed the experiment on 4194 real malware and the result is very promising.

Keywords— power law; malware analysis; packer; concolic testing; formal method

I. INTRODUCTION

Malware [1] or malicious software, is software program which targets on damaging or disrupting a computer. Popular kinds of malware include virus, trojans, spammer, flooder, keylogger, etc. In 2014, according to a report[1] from *International Data Corporation* (IDC) and the *National University of Singapore* (NUS), more than 491 billion dollars has been spent on the war against malicious software (malware).

For detecting malware, there are three major techniques including *signature matching*, *virtual emulation* in a sandbox, and *model checking*. Malware signature [2] is a binary pattern characterizing the typical features of malware. Most of industrial anti-virus softwares focus on identifying the regular expression based signature for detecting malware. However, since modern malware tends to adopt the obfuscation techniques especially with the use of packer for generating new invariants of malware, they can easily evade signature matching. For instance, using packer, a *polymorphic virus* can generate a complex signature, which is beyond the scope of regular expressions [3]. Figure 1 presents an example of packer UPX[1] which transforms from the original file (hello.exe) to a new file (hello_upx.exe). The newly-generated file preserves the same original functions of hello.exe but has the different content on the system. Note that packer adopts packing technique and many other obfuscation techniques which can generate a complex signature for defeating signature matching.

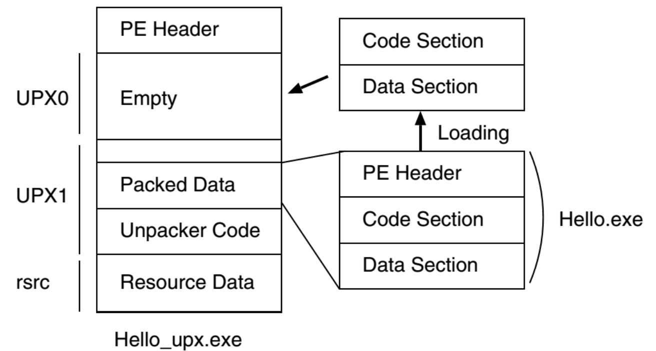

Figure 1. Example of packer UPX

Virtual emulation sets up a sandbox to explore behavior of malware. This technique requires a full emulation of system environment including *Window APIs* [11]. Not only it is a very heavy task but also it requires a suitable abstract level which is not easy. Furthermore, malware can adopt many anti-emulation techniques for detecting whether it runs on an emulator. As an alternative, recent research focuses on the approach of model checking. Model checking technique composes of two steps. The first step, model generation extracts an abstract model from binary executables. The *control flow graph* (CFG) is commonly chosen as an abstract model. When a CFG is generated, model checking technique is applied for checking the typical properties of malware [12][16][17].

[1]http://www.scmagazine.com/breaches-malware-to-cost-491-billion-in-2014-study-says/article/339167/
[2]http://upx.sourceforge.net

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

TABLE 1. Frequency of obfuscation techniques in packer

Packer	AntiDebugging	Checksumming	Code Chunking	Indirect Jump	Obfuscated Constants	Overlapping Block	Overlapping Function	Overwritting	Packing	SEH	Stolen Bytes	Timming Check	TwoAPIs	Hardware Breakpoints
ASPACK	x	x	x	x	x	x	x	x	x					
BJFNT	x	x	x	x	x	x								
EXEPACK	x	x	x	x	x	x	x							
EXESTEALTH	x	x	x	x	x	x	x	x	x	x	x			
EXPRESSOR	x	x	x	x	x	x	x	x					x	x
FSG	x	x	x	x	x	x	x	x						
LAME	x	x	x											
MEW	x	x	x	x	x	x	x	x						x
MORPHNAH	x	x	x								x			
MPRESS	x	x	x	x	x	x								
NOODLECRYPT	x	x	x	x	x	x	x							
NPACK	x	x	x	x	x	x	x	x	x					
PECOMPACT	x	x	x	x	x	x	x	x	x	x				
PEENCRYPT	x	x	x	x	x	x							x	
PETITE	x	x	x	x	x	x	x	x				x		
RLPACK	x	x	x	x	x	x	x	x	x					
SCRAMBLEv0.1	x	x	x	x	x	x	x	x						x
SCRAMBLEv0.2	x	x	x	x	x	x	x	x						
TELOCK	x	x	x	x	x	x	x	x	x	x	x			
UPACK	x	x	x	x	x	x	x	x						
UPX	x	x	x	x	x	x	x						x	
WINUPACK	x	x	x	x	x	x	x	x						
WWPACK32	x	x	x	x	x	x						x		
XCOMP	x	x	x	x	x	x	x	x	x					
YODAv1.2	x	x	x	x	x	x	x	x	x	x			x	x
YODAv1.3	x	x	x	x	x	x	x	x	x	x	x			
PELOCK	x	x	x	x	x							x		
PESPIN	x	x	x	x	x	x	x	x						x

According to [4], 80% of modern polymorphic malwares are obfuscated by packer to create new invariants. Among them, one notorious example is EMDIVI[3] virus, an *advanced persistent thread* (APT) which targets on many Japanese organizations, e.g. government agencies, local governments and universities. Malware adopts packer for defeating the signature based technique of anti-virus softwares by obfuscating its content. Moreover, packer also increases the difficulty of the reverse engineering since the process of unpacking or decrypting a packed file may take a very long time.

In this paper, we study the approach of applying probability distribution for analyzing malware. Based on this law, we measure the most important features of malware in two levels. For the instruction level, we identify the most important instructions belonged to each malware class. For example, since most of SEH-virus[4] use the technique of *Structured Exception Handler* (SEH), the instruction *mov fs:[0], esp* is very important. Although these malwares also adopts many other obfuscation techniques, this instruction is still the typical feature. In other way, when this instruction is identified in a sample, it can be infected by this class of malware. By using the probability distribution, we can measure the importance of this instruction to such kind of malware.

For technique level, we calculate the importance of obfuscation techniques in each packer. For example, Table I presents some well-known packers, each of which adopts various obfuscation techniques, e.g. *indirect jump*. Each obfuscation technique is deployed in a packer with a different frequency. However, the problem is that such obfuscation techniques can also be used not only by the packer code but also by other parts of the program. In addition, a packer itself can also have several different versions, whose frequencies of the obfuscation techniques may be slightly different. Moreover, since to disassemble a packed program for evaluating the frequencies of obfuscation technique is by no means a trivial task, resulting that the evaluated frequencies may not be perfectly correct. Thus, all of those reasons show that counting exact numbers of obfuscation techniques detected in a program may lead to misidentify the presence of packers adopted in this program. Our key contributions are presented as follows.

- We have developed a tool, BE-PUM[6][7][8] as a generic model generator with stubs of detecting obfuscation techniques. During the on-the-fly model generation, BE-PUM identifies and measures the frequency of instructions and obfuscation techniques in each malware.
- Based on BE-PUM tool, we implement a framework for tackling the two main problems in malware including identifying malware and detecting packer.
- For malware identification, we derive a new frequency-based weight, instruction frequency-inverse malware frequency $(if^{\circ}imf)$ to identify the most important instructions in each malware class.
- For packer detection, we propose a new term, obfuscation technique frequency-inverse packer frequency $(otf^{\circ}ipf)$ for calculating the favorite obfuscation techniques in packer. Based on this weight, we calculate similarity between targeted file and packer for packer identification.

https://www.symantec.com/security_response/writeup.jsp?docid=2014-101715-1341-99
[4]http://www.remove-viruskillers.com/post/What-is-Win32Injector.SEH-Remove-Win32Injector.SEH-Completely-Off-Your-PC_8_86668.html

- We perform the experiments on 4194 real malware taken from VirusTotal[5] for measuring the effectiveness of our approach.

The rest of this paper is organized as followed. Section 2 briefly describes the preliminaries. Section 3 introduces the overview of our method. Section 4 illustrates our case study on analyzing EMDIVI malware. Section 5 presents our experiments. The final section 4 discusses the conclusion and some future works.

II. PRELIMINARIES

In this section, we present the basic concept of BE-PUM, packers, the obfuscation technique in packer and the probability distribution.

A. BE-PUM

We have been developing a tool BE-PUM (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, SEH and many obfuscation techniques adopted in packer.

1) The framework of BE-PUM

BE-PUM implements CFG reconstruction based on concolic testing. Figure 3 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] 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) Running Example

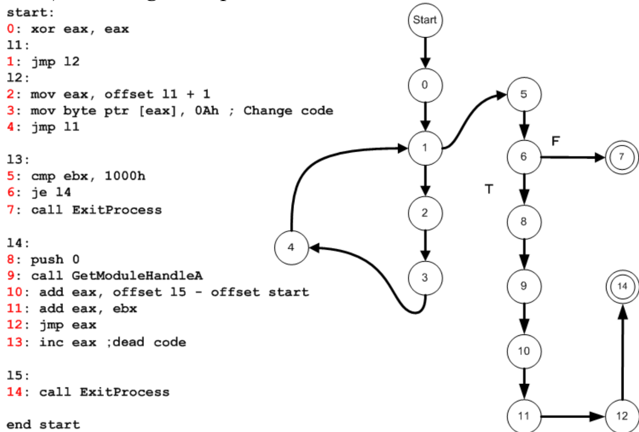

Figure 2. Running example of BE-PUM

We illustrate the operation of BE-PUM with a small example in Figure 2. At a first look, the execution follows the looping path $P = (start \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1)$. 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 *Java Native Access* (JNA). The return value is stored in register eax. The path formula of $(start \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12)$ 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).

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 [8], 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
```

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.

B. Packer

Packer targets on converting a binary file into another executable. The new one preserves the original file's functionality but with a different content on the system. Moreover, packer tries to compress targeted file for reducing the memory. However, the notable feature of packer is to protect the original file from being reversed, analyzed and tampered with. It combines many obfuscation methods including anti-debugging, anti-reverse engineering, and more for defeating the anti-virus software. This feature is mainly adopted in malware for protecting them from detection of anti-virus software.

Packer supports many obfuscation techniques. We categorize them into 6 groups as presented in details [8][9].

[5]https://www.virustotal.com

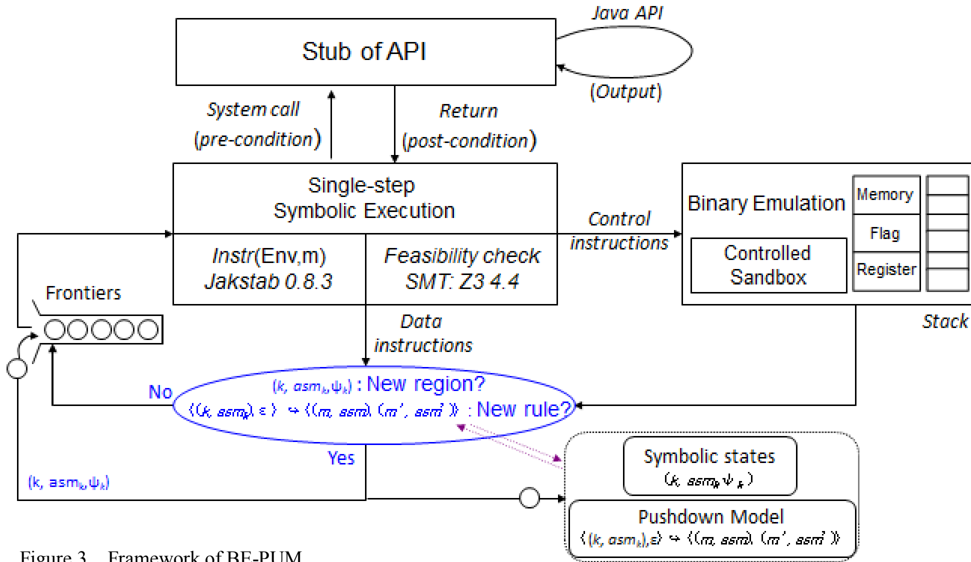

Figure 3. Framework of BE-PUM

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

We assume that each packer P can be represented a vector O of many obfuscation techniques $O=(O_1,..., O_k)$. Table 1 depicts the frequency of obfuscation techniques which we have measured using our tool BE-PUM. However, in analyzing real-world malware, the frequency of obfuscation technique is not the same with the value in Table 1. The reason is that malware can adopts these obfuscation techniques which cause noise in the expected value. Hence, the approach of exact matching produces the inaccurate results.

C. Probability Distribution

1) Power law

In analyzing malware, we have faced the problems of measuring the relationship between various quantities. We choose power law as a solution to develop measurement functions for these problems. Power law [14] is a statistical method which is widely used in many fields. In a specific context, power law is a function between two quantities which is proportional power. That is, a minor change in one quantity might cause a huge effect on the other. Each power law produces a different distribution on the quantities which can have practical applications. One of the most famous distributions is Pareto principle [13], well known as the 80:20 rules, which has a lot of practical applications in many fields, e.g. social, scientific, geophysical, and actuarial. In computer science field, Zipf's law [5] is the most famous applications of power law. In the context of text processing, Zipf's law states that frequency of a word is inversely proportional to its rank as a formula $f \sim r^{-b}$, where b depends on specific problem.

Term frequency-inverse document frequency ($tf{\circ}idf$) is a more specific application of power law and Zipf's law. $tf{\circ}idf$ measures the importance of a word to a document over a collection of text documents or corpus. For example, given a set of text documents $D = \{d_1, d_2...d_n\}$, $tf{\circ}idf$ helps to determine which document is most relevant the query. In this context, Zipf's law states that a word that occurs in more documents is less important in classifying documents over a corpus. Based on Zipf's law, various functions are proposed to calculate $tf{\circ}idf$. Among the most common uses, $tf{\circ}idf$ is defined as follow.

- Term frequency of a word t in a document d, $tf(t, d)$, is the number of time that t occurs in d.
- Inverse document frequency of a word t over a set of text documents D

$$idf(t,D) = log\frac{|D|}{|\{d \in D: t \in d\}|} \quad (1)$$

where $|D|$ is the number of documents, and $|\{d \in D: t \in d\}|$ is the number of documents that term t occurs.

- $tf{\circ}idf$ of a term t in document d over D is calculated as

$$tf{\circ}idf(t,d,D) = tf(t,d) * idf(t,D) \quad (2)$$

Figure 4 presents the plate notation [15] of $tf{\circ}idf$. Note that, the observable states (*occ* and *df*) are presented by shaded

circle and the computational states are depicted with the empty circle. The arrows illustrate the relationship between states. The W in the corner of the plate indicates that the variables inside are repeated for each word. The D means that it is for each document. *occ* state represents the number of occurrence for each word in each document. From *occ,* we can calculate the *tf* value (in *tf* state). For each word, we measure the *df* value (in *df* state) and extract the *idf* value by applying the formula (1). Combining the two value *tf* and *idf* with the formula (2), we calculate the value of $tf°idf$.

Original Zipf's law measures the relationship between two quantities in the same set of documents. In the following sections, we propose an extended model of Zipf's law to calculate relationship between various document sets in two particular applications as aforementioned.

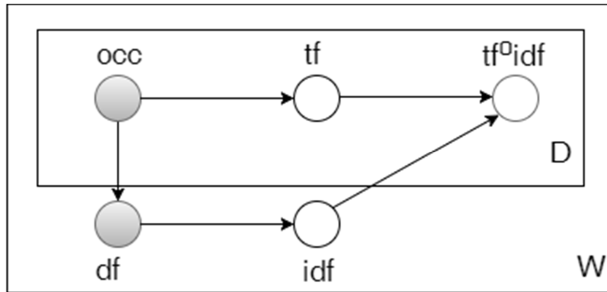

Figure 4. Plate notation of $tf°idf$ weight

Based on power law, the traditional $tf°idf$ calculates the value of *idf* on the whole set. However, in analyzing malware, we cannot calculate the same way. For tacking this problem we propose to separate the set of normal files and targeted files for strongly marking the importance of instructions or obfuscation techniques which rarely occurs in normal files.

III. THE APPROACH OF APPLYING PROBABILITY DISTRIBUTION ON EXTRACTING MALWARE FEATURES

In this section we introduce our methods for classifying malware and identifying packer based on the power law.

A. Preliminary

We separate the two sets, the set of malware and the set of normal program. Note that, based on BE-PUM, we can construct the control flow graph of program and measure the frequency of instructions and obfuscation techniques in each program. From the results of BE-PUM, we plan to extract the typical features which happen in malware and do not occur in normal file. Hence, we cannot combine the two sets as described in the calculation of $tf°idf$. We assume that the more a feature occurs in malware, the more important it is. However, the more it happens in normal program, the less important it is.

B. Method for packer detection

1) The $otf°ipf$ weight

Given a packer T which uses a set of obfuscation techniques $O = \{o_1, o_2 ..., o_n\}$. We denote $P= \{P_1, P_2....,P_n\}$ a list of malwares which are packed by T and a set of normal file $NP = \{NP_1, NP_2, ... ,NP_m\}$. For an assembly b, we calculate vector V_b, and the measure relationship between two vectors to identify if b is packed by T.

Based on *Zipf's law,* we measure a relationship between each obfuscation technique o and assembly b, $oft°ipf\ (o,b),$ which calculated as follow

- *Raw frequency* of an obfuscation technique o in an assembly b, $f(o,b)$, is the number of time o appears in b.
- *Obfuscation technique frequency* of an obfuscation technique o in an assembly b

$$otf(o,b) = \frac{f(o,b)}{f(o_{max},b)} \quad (3)$$

  Where $f(o_{max},b)$ is the maximum value of $f(o,b)$ over all obfuscation technique
- *Inverse packer frequency* of an obfuscation technique o

$$ipf(o) = \frac{|NP|}{0.001 * |NP| + occ(o,NP)} \quad (4)$$

  Where $occ(o, NP)$ is the number of program p in NP contains o. The factor $0.001*|NP|$ is an adjustment to avoid division-by-zero. Note that we calculate the value of *ipf* in NP (normal file) set for pointing out the importance of obfuscation technique. The more an obfuscation techniques occurs in normal file, the lest importance it is.
- *otf°ipf* of an obfuscation technique o in an assembly b is calculated

$$otf°ipf = otf(o,b) * ipf(o) \quad (5)$$

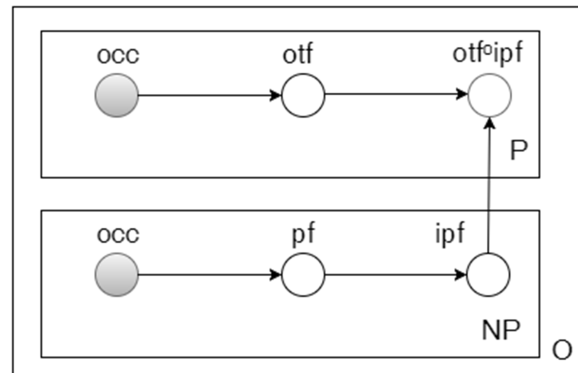

Figure 5. Plate notation of $otf°ipf$ weight

Figure 5 presents the plate notation of $otf°ipf$ weight. Note that we separate in two sets as described in the corner of plate, P for packed malwares and NP for normal files. For each obfuscation technique O, we measure the *otf* value for P set by applying the formula (3). *ipf* is calculated for NP set by

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

applying the formula (4). By applying (5), we extract the $otf°ipf$ for O.

As the formula implies, if *o* occurs more in *b* while *o* occurs less in *NP*, *o* is more important in classifying if *b* is packed by *T*. With calculated $oft°ipf$ for all obfuscation techniques, we propose a method to identify packer as follow.

- Generate vector V_T

$$V_T = \left\{\frac{\sum_1^n otf°ipf(o_1,p_i)}{n}, \frac{\sum_1^n otf°ipf(o_2,p_i)}{n} \dots, \frac{\sum_1^n otf°ipf(o_k,p_i)}{n}\right\}$$

- Generate vector F_b

$$V_b = \{otf°ipf(o_1,b), otf°ipf(o_2,b) \dots, otf°ipf(o_k,b)\}$$

- Calculate Euclidean distance between these two vectors V_F and V_b. If the distance below a threshold ε, *b* is packed by *T*. From the empirical study, we choose ε = 0.001.

2) Running example

Let see an example of 6 files and 14 obfuscation techniques, where *P = {Demo1, Demo2, Demo3}* are packed by packer *UPX*, *NP = {Demo4, Demo5}* are normal files, and unknown file F. The frequency of obfuscation techniques on these files is listed in Table 2. Consider *Indirect Jump* (IJ) technique, $oft°ipf(IJ, Demo1)$ is calculated as follow

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

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

So we have $oft°ipf(IJ, Demo1) = 0.16 * 1.996 = 0.31936$

Similarity, $oft°ipf$ of all obfuscation techniques and each files are listed in Table 3.

So we can calculate vectors V_{UPX}, and V_F as follow V_{UPX}={0,0.339869,0,0.313725,1.882352,0.078431,0,0,0,0,0,0, 0.078431,0}

V_F={0,0.338859,0,0.302725,1.872302,0.063431,0.03,0,0,0,0,0 ,0.075451,0}

Since $d(V_{UPX}, V_F) = 0.633527 > ε$, then F is not packed by UPX

C. Method for malware classification

1) The $if°imf$ weight

Based on the famous Zipf's law, we derive a new frequency based weight to identify instruction relevance in malware family, known as instruction frequency-inverse malware frequency, $if°imf$. Given a set V of infected files and a normal binaries set NV, the $if°imf$ weight of a certain instruction *I* is defined as follows.

- Frequency of an instruction *I* in a program set P

$$if(I,P) = \frac{f(I,P)}{f(I_{max},P)} \quad (6)$$

where *f(I,P)* is total number of times *I* occurs in *P* and I_{max} is the instruction which has largest frequency in P.

- Inverse-packer frequency of an instruction

$$imf(I) = \frac{|NV|}{0.001 * |V| + occ(I,V)} \quad (7)$$

where occ is the number of files in *V* that *I* occurs.

Then

$$if°imf(I) = if(I,P) * imf(I) \quad (8)$$

In the *ipf* formula, the factor $0.001 * |V|$ is an adjustment to avoid division-by-zero. Note that, for each malware family *V*, instructions *I* with high value of $if°imf$ shows a strong relationship with the *V* they appear in, It implies that if that instruction were to appear in a targeted file *F*, then *F* can belong to *V*.

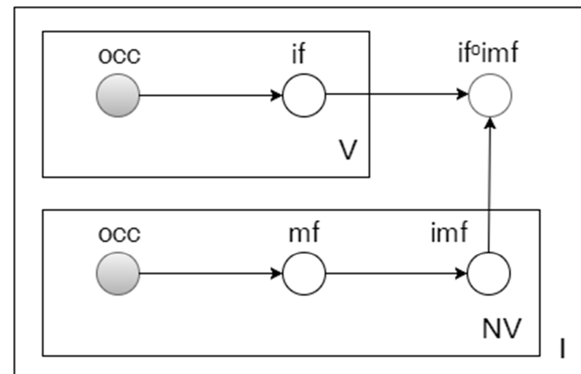

Figure 6. Plate notation of $if°imf$ weight

Figure 6 illustrates the plate notation of $if°imf$ weight. Note that we also separate in two sets as described in the corner of plate, *V* for viruses and *NV* for normal files. For each instruction *I*, we measure the *if* value for V set by applying the formula (6). ipf is calculated for *NV* set by applying the formula (7). By applying (8), we extract the $if°imf$ for O.

IV. CASE STUDY: ANALYSIS OF EMDIVI VIRUS

EMDIVI is an APT malware which targeted in many organizations in Japan e.g. banks, government agency, university. It adopts many complex obfuscation techniques to evade the signature matching detection in anti-virus software. Our tool BE-PUM can analyze this malware and extract the frequency of obfuscation techniques. The details are presented in the Table 4. Another notorious feature is that BE-PUM can also extract the address of C&C server (Command and Control) which this malware connects to transfer information. The hostname is "www.n-fit-sub.com". To the depth of our knowledge, BE-PUM is the first model generation tool which achieves this result.

TABLE 4. Frequency of obfuscation techniques in EMDIVI

Obfuscation Technique	Frequency
AntiDebugging	0
Checksumming	4
CodeChunking	0
IndirectJump	4
ObfuscatedConst	23
Overlapping Block	1
Overlapping Function	0
Overwriting	0
Packing/Unpacking	0
SEH	0
Stolen Byte	0
Timing Check	0
Special API	1
Hardware BPX	0

V. EXPERIMENTAL RESULTS AND DISCUSSION

A. Experiment Setup

We perform the experiments of packer detection on Windows XP with AMD Athlon II X4 635, 2.9 GHz and 8GB memory. The samples are 4194 real-world malwares collected from VirusTotal. Among them 1258 of these samples are downloaders, 2120 are worms, and the rest are trojans. Since the lack of time and resource, we cannot perform the experiment of malware classification.

B. Experiment on real-world malware

Among 4194 malware, our approach succeeds on 3765 malware. The other files are unknown. Figure 7 presents the results between our approach using BE-PUM with the method of binary signature using CFF Explorer. In figure 7, the vertical axis shows the number of malware identified for each methods. Clearly, our approach shows the better results. In some cases, our approach can detect the unknown packer. For example, consider malware *034f9d2dc5627296141bb7d0a11032b1e8c-7e47f266ada4a1da7f8dad05668b*. Its binary code is "*60 BE 12 E0 95 00 8D BE 00 30 AA FF C7*" which is disassembly as follows.

```
00A31B20 >  60            PUSHAD
00A31B21   . BE 12E09500    MOV ESI,0034f9d2.0095E120
00A31B26   . 8DBE 0030AAFF  LEA EDI,DWORD PTR DS:[ESI+FFAA3000]
00A31B2C   . C787 D0566200 >MOV DWORD PTR DS:[EDI+6256D0], 2A11
```

Our approach detects it as UPX while CFF Explorer fails. The reason is that the signature of UPX is "*60 BE ?? ?? ?? 00 8D BE ?? ?? ?? FF 57*". The two binary codes differ at the final byte, *C7* vs *57*.

V. CONCLUSIONS

This paper proposes a new approach of using probability distribution for tackling two goals including identifying malware and detecting packer. For the first goal, we derive a new weight $if°imf$ for extracting the most important instructions of each malware class. For the second one, we measures the frequency of obfuscation techniques and extracts the obfuscation technique relevance in packer based on the new weight $otf°ipf$. Based on the vector of $otf°ipf$, we can calculate the distance for identifying packers. Experiments and observation confirm that BE-PUM correctly handles obfuscation techniques and detect packer on 4194 real-world malware. In the future work, we will increase the number of packer for better results. Another future work is that we will perform the experiments on malware classification using the new weight $if°imf$.

REFERENCES

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

[2] E. Filiol, Malware pattern scanning schemes secure against black-box analysis," Journal in Computer Virology, vol. 2, pp. 35–50, 2006.

[3] E. Filiol, "Metamorphism, formal grammars and undecidable code mutation," Int. J. Comput. Sci. 2, , 2007, pp. 70–75.

[4] M. Morgenstern and A. Marx. "Runtime packer testing experiences". In Proceedings of the 2nd Computer Antivirus Research Organization Workshop, Hoofddorp, Netherlands, 2008, 288-305.

[5] G. K. Zipf, The Psycho-Biology of Language. "An Introduction to Dynamic Philology". Boston, USA: Houghton-Mifflin Company, 1935.

[6] 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, pp.159-164, 2013

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

[8] Nguyen Minh Hai and Quan Thanh Tho. "An Experimental Study on Identifying Obfuscation Techniques in Packer", 5th World Conference on Applied Sciences, Engineering & Technology, 02-04 June 2016, HCMUT, Vietnam, ISBN 978-81-930222-2-1.

[9] 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

[10] J.Kinder, F.Zuleger, and H.Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in VMCAI 2009, pp. 214–228, 2009. 214–228.

[11] 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.

[12] 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.

[13] 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): 165–186.

[14] Manfred Schroeder, Fractals, and Chaos, “Power Laws: Minutes from an Infinite Paradise”. W.H. Freeman and Company, New York, 1991

[15] Buntine, Wray L. "Operations for Learning with Graphical Models". Journal of Artificial Intelligence Research (2), 2008, 159–225.

[16] F. Song and T. Touili, “Pushdown model checking for malware detection,” in 18th Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 110–125, 2012. LNCS 7214.

[17] 7. F. Song and T. Touili, “LTL model-checking for malware detection,” in 19th Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 416– 431, 2013. LNCS 7795

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

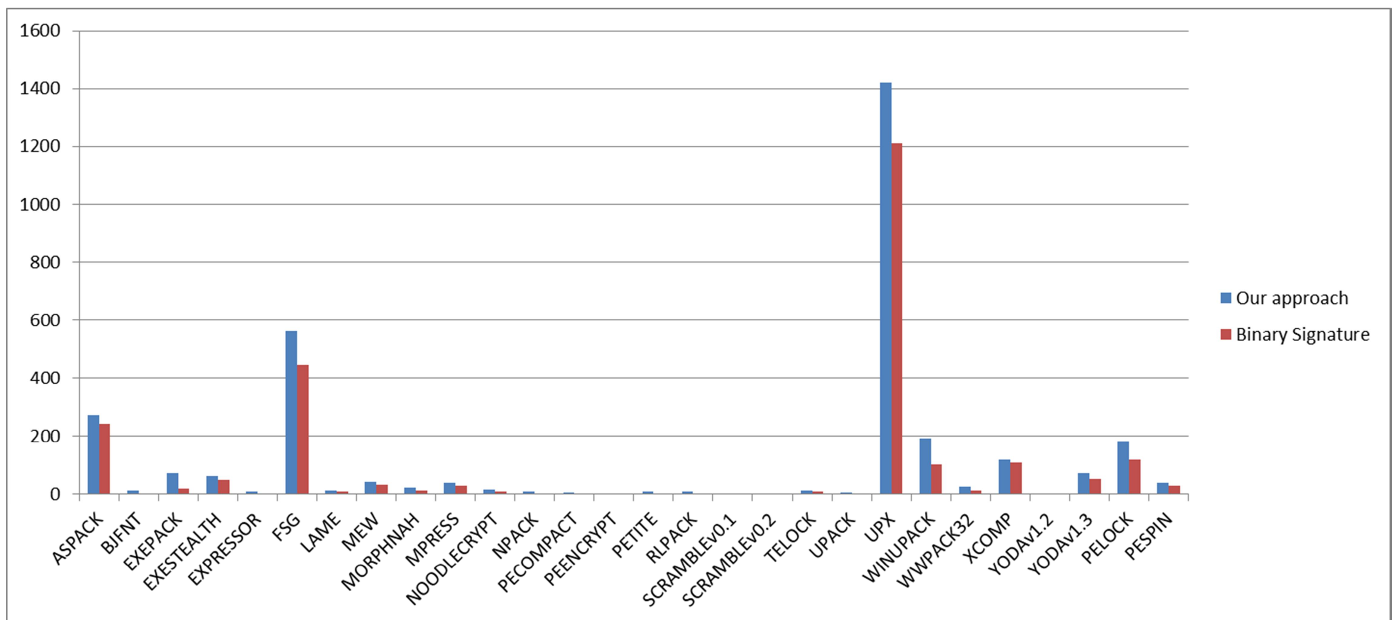

Figure 7. Experimental results