A Novel Malware Variants Detection Method Based On Function-call Graph

Lingfei Wu
Institute of Computer Application Technology
Hangzhou Dianzi University
Hangzhou, China, 310018
50fei@163.com

Ming Xu Jian Xu Ning Zheng Haiping Zhang
Institute of Computer Application Technology
Hangzhou Dianzi University
Hangzhou, China, 310018
mxu@hdu.edu.cn

Abstract—**Code obfuscation plays a significant role in metamorphic malware. Moreover, identifying a metamorphic malware variant is a challenge task, because its obfuscation engine can easily generate various variants with different forms while maintaining the same functionality to escape detection. This paper presents a novel approach to recognize metamorphic malware based on programs' function-call graphs. Graph-coloring and cosine similarity techniques are used to measure the similarity of two programs on the basis of function-call graph. Experimental results have shown that the proposed method can accurately detect the metamorphic malware variants.**

Keywords-malware; graph-coloring; function-matching

I. INTRODUCTION

The terminology "malicious software (malware)" refers to different kinds of software which intend to carry out malicious tasks on computer system. The typical malware include virus, worms, spyware, and trojans ect. It will engender an enormous loss to person even if the society.

Nowadays the number of malware increases dramatically, especially, code obfuscation techniques aggravate this phenomenon. According to the "Message Labs Intelligence: 2010 Annual Security Report" [1] of Symantec, in 2010, there were over 339,600 different malware strains identified in emails blocked, representing over a hundredfold increase compared with 2009. This is largely due to the growth of polymorphic and metamorphic malware variants, typically generated from toolkits that allow a new version of the code to be generated quickly and easily. Effective antivirus techniques should be proposed to detect the obfuscated malware and mitigate the damages caused by malware.

In order to evade the detection of antivirus products, malware writers have to improve their skills in malware writing. Obfuscation is to obscure the information such that others cannot find the true meaning. Malware writers use this technique to obfuscate malicious code so that it is difficult to reverse-engineer, and then its malicious content cannot be detected. Obfuscation can easily change the structures of malware and keep the programs' semantics and functionality invariant. A lot of obfuscation engines were designed by malware authors in the past few years, such as Mistfall, Win32/Simile, and RPME [2]. Therefore, metamorphic malware will go into mass production easily.

However traditional methods for malware detection are mostly based on malware signature. It uses syntactic information as a signature rather than semantic information. Now, more and more researchers are paying attention to semantic signatures to detect malware. This paper is an improvement of [3], which proposed a method to detect malware variants based on function-call graph. Function-call graph can describe a program's function. It is effective to detect malware variants using this graph. In this paper we propose a new method to match functions instead of LCS (Longest Common Subsequence) using in [3]. For obfuscation, our method performs more robust.

The rest of this paper is organized as follows. Section 2 review related works. In section 3, we detail our analytical approach and steps. Section 4 evaluates our techniques and in Section 5, limitations and future works are pointed out. We conclude in Section 6.

II. RELATED WORKS

Malware detection is a hot field as the development of computer and internet. Many predecessors have done a lot of research in this field. However, as obfuscation techniques continuously evolved. It leads to some signature-based detection approaches useless.

In order to make obfuscation detection more reliable, a great number of means have been proposed in the passed several years. Mihai Christodorescu et al. [8] defined a dependence graph as a malware's signature and proposed a mining algorithm to construct the graph by use of dynamic analysis. Jianli et al. [9] extract maximal pattern sequence from system call sequence, and use this pattern as a feature to compute similarity among malwares.

The papers discussed above are all based on dynamic analysis. There are also some solutions for metamorphic malware detection using static analysis. Gheorghescu [10] generates a CFG (control flow graph) by traversing the code of a program and uses this graph as its characteristic. Kapoor and Spurlock [11] argue that comparing malware on the basis of functionality is more effective than binary code comparison. R. Tian and L.M. Batten et al. [12] use some mathematics methods such as Chi-square test to classify malware based on functions' length, the functions are obtained by IDA pro [6]. Abhishek Karnik et al. [7] use cosine similarity to compute similarities among functions of two malware, consequently, the similarity of two malwares can be obtained. But a drawback of this method is that such a scheme could be subverted by the use of methods such as the instruction substitution method or using a dictionary of similar instructions. Also, on the field of static analysis, some researchers focus on library or system functions which are called. Qinghua Zhang, Douglas S. Reeves [13] exactly make use of library or system functions as patterns for detection. This approach makes a backwards data flow analysis to a program, and use the intermediate representation of semantic instructions to obtain malware's pattern. System-call graph is obtained using the algorithm proposed by [14] and use this graph to detect or classify malicious programs. Jusuk Lee [15] improve the method of [14] by classifying API calls into 128 groups, and make the graph to be more fast simple analysis. This method ignores the information of local functions, thus, it may cause higher false positive. Also, it may suffer the code obfuscation techniques such as insertion of meaningless system calls.

III. Function-call Graph System

To realize our idea, the design of the whole system is divided into three steps.

a) Generate function-call graph using assembly code.
b) Match each vertex in the graph.
c) Compute similarity of two graphs.

Before creating the function-call graph, we should make sure that the malware problem can be disassembled accurately. Tools such as PEiD [4] and UPX [5] are used to unpack the program.

A. *Function-call Graph Definition*

Function-call graph is graph of a program's binary code. It describes the functions' relationship of a program and is consist of a set of vertices and a set of edges between vertices. Assembly code can be obtained from IDA pro [6]. A function-call graph is written $G = (V, E, \mu, \omega)$, where:
V is vertex-set. Each vertex is corresponding to a unique function in the program;
E is edge-set, $E \subseteq V \times V$;
μ is a labeling function to label the vertices, $\forall v \in V$, $\exists \mu(v) \in Lv$, where Lv denotes a set of vertex attributes;
ω is a weighting function to compute the edges' weights and the vertices' indegrees and outdegrees.
Specifically, functions (vertices) are classified into three categories: Dynamically-Imported functions, Statically-Linked library functions and the Local-Subroutines.

B. *Function-call Graph Generation*

For generating the function-call graph, the algorithm adopts a breadth-first approach to build the function-call graph, and the graph is stored into an orthogonal list. It builds the caller-callee relationship starting from the entry point functions. It traverses each function's instructions to find all the subroutines called by the function. The graph will be constructed when all the functions are processed. Details about how to build the function-call graph can be seen in [3].

C. *Function-matching*

Semantically, if two malware perform the same functionality, the vertices in both graphs are connected in the same way. Grounded on this assumption, it is necessary for us to match each vertex between two graphs. We call this matching process as function-matching. It ends when we find all the common vertices.

Function-matching is divided into two parts. First, we match vertices of two programs using the feature of function-call graph's structure and ignore the inner information of the function (this part has been discussed in [3], we will not introduce it in this paper). Second, the functions' internal information should be used. The detail of this part is described below.

There are still many vertices haven't been thought over after the first part of matching algorithm. We realize the function of the second part through 2 steps. In the first step, preprocessing the unmatched vertices (functions) using the technique of graph coloring. In the second step, similarity algorithms are used to calculate the similarity based on the vertices (functions) which have the same color through the first step. In this paper, we use cosine similarity to calculate the similarity.

Step 1, in this step, we mark a color for each vertex in the light of functionality of every instruction. We classify the X86 instructions into 15 classes according to their functions as shown in TABLE Ⅰ. 15 bit color values are defined to describe a fingerprint for each vertex and initialize these values to 0. Each bit corresponds to a certain class of instructions. We traverse every instruction of a function and map it to the class. If one or more instructions appear in the class, set the corresponding bit to 1, at the same time, the number of instructions belongs to a certain class should be added up for the calculating of the second step. When we finished traversing the instructions of a function, we get a color (fingerprint) of corresponding vertex. The pair of vertices which have the same color are selected to compute similarity in succeeding step. The graph coloring technique can cope with certain instruction substitution obfuscation such as replace instructions with other function-similarity ones, since the function-similarity instructions have been classified to the same class.

TABLE I. Color classes

class	Description	class	Description
data transfer	such as mov instruction	jump	unconditional transfer
stack	stack operation	branch	conditional transfer
port	in and out	loop	loop control
lea	destination address transmit	halt	stop instruction execution
flag	flag transmit	bit	bit test and bit scan
arithmetic	incl. shift and rotate	processor	processor control
logic	incl. bit/byte operations	float	Floating point operations
string	string operations		

Step 2, cosine similarity will be used on the basis of step 1, which have calculated the frequency of each color class.

Specifically, there are 15 classes. Vectors which have 15 dimensions are used as parameters for cosine similarity. Every dimension in the vector represents the number of instructions occur in corresponding class. Given two vectors *X* and *Y*, *X = (x1, x2, ..., xn),* and *Y = (y1, y2, ..., yn)*. *θ* means the angle of the two vectors. Then, the cosine similarity of is calculated through the following formula:

$$\cos(\theta) = \left(\sum x_i \cdot y_i\right) \Big/ \sqrt{\left(\sum x_i^2 \cdot \sum y_i^2\right)} \tag{1}$$

This is an example of how to calculate the similarity between two functions. TABLE II shows two sequences of instructions. According to these sequences, statistics the number of every certain class and corresponding vectors are obtained in TABLE III. Then, formula (1) comes on the stage. Thus, the similarity score between A and B is 0.797.

TABLE II. Two sequences of instructions as an example

Sequence A	Sequence B
push	push
mov	push
mov	push
mov	mov
xor	mov
add	xor
test	xor
jz	mov
test	cmp
jz	jz
mov	call
push	pop
call	pop
	pop

TABLE III. Vectors obtained from each sequence of Table 2

Class (vector)	data transfer	stack	logic	branch	jump	arithmetic
Num A	4	2	3	2	1	1
Num B	3	6	2	1	1	1

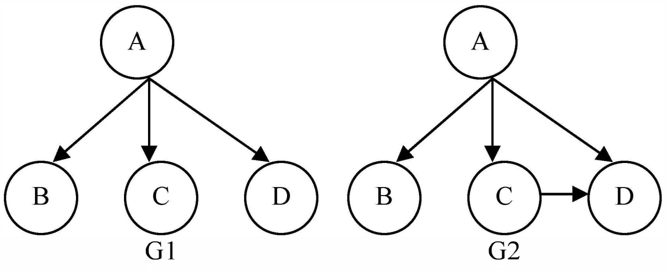
Up to now, we have introduced every single step of our algorithm. Next, we will match the exact vertices using the proposed method on the whole scale. Pseudo-code for our matching algorithm is given in Algorithm 1.

In Algorithm 1, vertices in two graph *G1* and *G2* are chose. The color of each vertex are got from 1 to 5 using *GetFingerprint()*. From 6 to 8, vectors as mentioned previously are computed using *GetVector()* for vertices which have the same color. *cosine_similarity()* in 9 is a function which calculate similarity using cosine similarity. For each unmatched vertex in *G1*, every vertex in *G2* will be traversed for matching. It is very likely that a color of a certain vertex will match more than one vertex in another graph. As we know, when one vertex in *G1* tries to find a matching vertex in *G2*, there are only two consequences, one matching vertex in *G2* or not. For finding the right matching vertex in *G2*, pseudo-code from 10 to 22 is shown to realize this function. *maxSim* in 11 is defined as the maximal similarity from the vertex pairs which have the same color. From 17, we can see that if the maximal similarity is greater than a designated threshold τ (τ = 0.85 is chosen empirically) in advance, then the corresponding vertex *j* in *G2* is the match for vertex *i* in *G1* and add this vertex pair into the *match_set*. The program ends when we finished traversing all the unmatched vertices left from step 1.

```
For each unmatched vertices (functions) in graph G1 and G2 do the following:
Input: sequences of instructions of every function u[i] and v[j]
Output: a set of matched functions {..., (x,y), ...}
1  foreach u[i] ∈ U do
2    maxSim ← 0;
3    G1.color[i] ← GetFingerprint(u[i]);
4    foreach v[j] ∈ V do
5    G2.color[j] ← GetFingerprint(v[j]);
6    if G1.color[i] is equal to G2.color[j] do
7      G1.vector[i] ← GetVector(u[i]);
8      G2.vector[j] ← GetVector(v[j]);
9      Similarity ← cosine_similarity(G1.vector[i],G2.vector[j]);
10     if Similarity > maxSim then
11       maxSim ← Similarity;
12       x ← i;
13       y ← j;
14     else goto 4
15   else goto 4
16   end for
17   if maxSim > τ then
18     match_set ← match_set ∪ (x,y);
19     U ← U-u[x];
20     V ← V-v[y];
21     return match_set;
22   else goto 1
23 end for
```

Algorithm 1

D. *Similarity Index Definition*



Figrue 1. An example for computing similarity

The function-matching has finished. Next job is to compute the similarity score between two graphs. As we seen in Figure 1, *G1* is different from *G2*, but if we only consider vertex's information of two graphs, the similarity of *G1* and *G2* would be 100%. To measure the similarity more precisely, edge's information should be considered, because edges are able to represent characteristic of graph than vertices. Then, we define the similarity *sim* of graph *G1* and *G2* as follows.

$$sim(G1, G2) = \frac{2 \times |E(G1 \cap G2)|}{(|E(G1)| + |E(G2)|)}$$

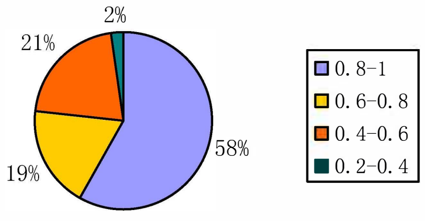
It means the ratio of the same edges of two graphs. In this formula, *|E(G1 ∩ G2)|* represents the number of the same edges of graph *G1* and *G2*. *(|E(G1)|+|E(G2)|)* represents the total number of edges of graph G1 and G2.

Obviously, for any *G1* and *G2*, *sim(G1,G2)*∈*[0,1]*. *sim(G1,G2)* is more closer to 1 means they are more similar. According to Figure 1, *G1* has 3 edges and *G2* has 4 edges. And the number of the same edges of *G1* and *G2* is 3, the total number of edges is 7. Thus, the similarity of *G1* and *G2* is 2*3/7, 0.8571.

IV. Evaluation

To inspect the effectiveness and correctness of our idea, we performed a series of experiments with the prototype system. Numerous sets of malware mutants were downloaded from VX Heavens [16]. At the same time, some benign programs were also collected for our experiments. In the first experiment, a great number of metamorphic malwares were chosen as examples to calculate similarities among them. The second experiment attempts to show the performance of classification among different malware families. In order to evaluate the capability of our algorithm to distinguish malicious programs from benign programs, the third experiment will be done.

A. *Variant Similarity Evaluation*



Figrue 2. Similarity statistics of malware variants pairs belong to the same family

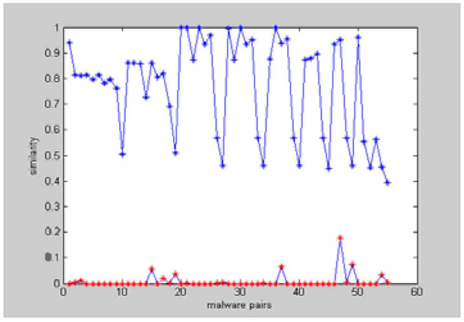
More than 200 pairs of variants which can be unpacked and disassembled correctly were collected. Size of these malware programs ranges from 8K to 1M bytes. We compute the similarity score of every malware pair that perform the similar or same task. The result, according to our statistics, is shown in Figure 2. The percentage of pairs which have the similarity score from 0.4 to 1 is about 98%. Hence, we can recognize obfuscated malware easily. Here, about 2 % pairs have a similarity score ranges from 0.2 to 0.4. This is because sizes of a malware and its variants are largely different, thus, makes the number of edges has a great difference. Obviously, similarity score will diminish using our formula in this condition. For example, size of Backdoor.Win32.DarkMoon.j and Backdoor.Win32.DarkMoon.m are 84.8KB and 58.5KB respectively, corresponding, the number of their edge are 1031 and 676 respectively. The similarity score is 0.207.

B. *Malware Classification*

In the first experiment, whether variants belong to the same malware family has been determined. In this section, we will evaluate the performance of malware classification. The malware samples listed in TABLE Ⅳ are used for this evaluation. The left part (malware belong to the same family) is corresponding to the higher scores in Figure 3. In the same way, samples of the right part are corresponding to the lower scores. In Figure 3, x-axis means the malware pairs and y-axis shows the similarity score. According to this figure, malware classification is not a confused job. Similarity scores of different families always less than 0.1 even more close to 0. The similarity value 0.1761 between Trojan-IM.Win32.Agent.j and Virus.Win32.BHO.a seems a little high, this is because a common code base may be used even they are in different families.

TABLE IV. Hybrid malicious samples

Malwares belong to the same family	**Malwares of different families**
Email-Worm.Win32.Mimail.a	Backdoor.Win32.Adbreak
Email-Worm.Win32.Mimail.c	Backdoor.Win32.DarkMoon.ai
Email-Worm.Win32.Mimail.e	Email-Worm.Win32.Klez.f
Email-Worm.Win32.Mimail.f	Email-Worm.Win32.Mimail.f
Email-Worm.Win32.Mimail.g	Email-Worm.Win32.NetSky.z
Email-Worm.Win32.Mimail.j	Rootkit.Win32.Vanti
Email-Worm.Win32.Mimail.k	Trojan-IM.Win32.Agent.j
Email-Worm.Win32.Mimail.l	Trojan-Spy.Win32.AdvancedKeyLo-gger17
Email-Worm.Win32.Mimail.m	Virus.Win32.BHO.a
Email-Worm.Win32.Mimail.o	Virus.Win32.Sality.a
Email-Worm.Win32.Mimail.p	Worm.Win32.QQPass.o



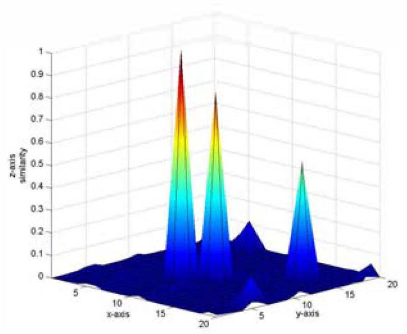
Figrue 3. Similarity scores of hybrid malicious samples

C. *Evaluation Together With Benign Programs*

Evaluation within malicious programs has been done. To appraise the availability of our idea, we implement our prototype and apply it to a set of malicious and benign programs. Samples are listed on TABLE Ⅴ. Similarity scores are shown in Figure 4. In this graph, x-axis and y-axis represent malicious samples, and the z-axis means the similarity value. Pairs which have the same function can be distinguished easily (pairs like kido.ih and kido.dam.x, IE7 and IE8, sality.d and sality.e have similarity scores 1, 0.82 and 0.51 respectively). All of the scores that benign programs compared to any one of the malicious are very low (most of them are blow 0.1 and more close to 0). Only the similarity score (more than 0.1) of lsass.exe and pid.dll seems a little abrupt. This is because both of them are consisted of system functions primarily and the same system call accounts for a significant proportion in these two programs. However, we match system functions just depend on their function name. If they have the same name, they are regarded as a match.

TABLE V. Benign and malicious samples

Benign	Malicious
ipv6.exe	Trojan.Win32.AVKill.a.
lsass.exe	Trojan.Win32.ICQPager.b
netstat.exe.	Worm.Win32.Bagle.i
cdm.dll	Virus.Win32.Evol.a
pid.dll	Worm.Win32.Kido.ih
md5sum.exe	Worm.Win32.Kido.dam.x
putty0.60	Worm.Win32.Mimail.c
Firefox3.6.3	Virus.Win32.Sality.d
install_icq7.exe	Virus.Win32.Sality.e
IE7-WinXP-x86-chs.exe	Virus.Win9x.ZMorph.5200
IE8-WinXP-x86-chs.exe	Virus.Win32.Zmist



Figrue 4. Similarities among benign and malicious samples

V. LIMITAIONS

This detection technique is based on static analysis, so the program must be disassembled before. A key point of disassemble is that the accuracy may not achieve to 100% [17]. Call instruction obfuscation, entry point obfuscation and implicit function-call can influence the detection rate since these techniques hinder the construction of function-call graph.

Our method on function-matching using graph coloring technique is invalid to some instruction substitution situations (one instruction is replaced by an instruction in another class). Such as *sub ecx, ecx* (which belong to the arithmetic class) is identified as equivalent to *mov ecx, 0* (which is a data transfer instruction).

In the future, implicit function-call should be considered and also we will try to find another method to deal with the code obfuscation techniques that the idea can not manage in this paper.

VI. CONCLUTION

This paper proposed a new method to match each functions between two programs on the basis of their function-call graphs.

The key idea of our method is to use graph-coloring technique together with statistical to complete the function-matching. Graph-coloring is a pretreatment for statistical. The statistical process is to match vertices of two programs. In addition, edges' information is used to calculate the similar value.

In the end, abundant wild malwares and benign applications were used as samples to inspect our idea. The result shows that malware variants can be well classified using the prototype in line with our method.

ACKNOWLEDGMENT

This paper is supported by NSFC (No. 61070212, No.61003195), Natural Science Foundation of Zhejiang Province, China (No. Y1090114), the State Key Program of Major Science and Technology (Priority Topics) of Zhejiang Province, China (No 2010C11050).

REFERENCES

[1] MessageLabs|Symantec Hosted Services. http://www.messagelabs.messagelabs.com/intelligence.aspx.

[2] P. Szor. The Art of Computer: Virus Research and Defense. Symantec Press, NJ, USA, first edition, 2005.

[3] Shanhu Shang, Ning Zhen, Jian Xu, Ming Xu and Haiping Zhang. Detecting malware variants via function-call graph similarity. In 5th malicious and unwanted software (malware), 2010:113-120.

[4] PEiD 0.95, http://www.peid.info/, 2010.

[5] UPX 3.05, http://upx.sourceforge.net/, 2010.

[6] IDA Pro 5.5, http://www.hex-rays.com/idapro/, 2010.

[7] A. Karnik, S. Goswami, and R. Guha, Detecting Obfuscated Viruses Using Cosine Similarity Analysis, First Asia International Conference on Modelling & Simulation (AMS'07), Phuket: IEEE Computer Society, 2007, pp. 165 - 170.

[8] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In Proceedings of the 6th ESEC/FSE, September 2007.

[9] Jian Li, Ming Xu, Ning Zheng, Jian Xu. Malware Obfuscation Detection via Maximal Patterns. Intelligent Information Technology Application (IITA). 2009:324-328.

[10] M. Gheorghescu. An Automated Virus Classification System. In Virus Bulletin Conference October 2005, pages 294–300, 2005.

[11] A. Kapoor and J. Spurlock. Binary Feature Extraction And Comparison. In Presentation at AVAR 2006, Auckland, December 3–5, 2006.

[12] R. Tian, L.M. Batten, S.C. Versteeg. Function Length as a Tool for Malware Classification. In 3rd malicious and unwanted software (malware), 2008:69-76.

[13] Q. Zhang and D. Reeves. MetaAware: Identifying Metamorphic Malware. In Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07), pages 411-420, December 2007.

[14] K. Jeong and H. Lee. Code Graph for Malware Detection. In Information Networking. ICOIN. International Conference on, Jan 2008.

[15] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. Detecting Metamorphic Malwares using Code Graphs. SAC '10 Proceedings of the 2010 ACM Symposium on Applied Computing.

[16] VX heavens. http://vx.netlux.org.

[17] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In Proceedings of the 13th USENIX Security Symposium, pages 255–270, Augeust 2004.