A similarity metric method of obfuscated malware using function-call graph

Article *in* Journal in Computer Virology · February 2013

DOI: 10.1007/s11416-012-0175-y

CITATIONS
35

READS
423

7 authors, including:

Ming Xu
Hangzhou Dianzi University
91 PUBLICATIONS **408** CITATIONS
SEE PROFILE

Jian Xu
Hangzhou Dianzi University
31 PUBLICATIONS **187** CITATIONS
SEE PROFILE

Ning Zheng
Hangzhou Dianzi University
49 PUBLICATIONS **311** CITATIONS
SEE PROFILE

J Comput Virol Hack Tech (2013) 9:35–47
DOI 10.1007/s11416-012-0175-y

ORIGINAL PAPER

A similarity metric method of obfuscated malware using function-call graph

Ming Xu · Lingfei Wu · Shuhui Qi · Jian Xu · Haiping Zhang · Yizhi Ren · Ning Zheng

Received: 12 January 2012 / Accepted: 16 August 2012 / Published online: 22 January 2013

Abstract Code obfuscating technique plays a significant role to produce new obfuscated malicious programs, generally called malware variants, from previously encountered malwares. However, the traditional signature-based malware detecting method is hard to recognize the up-to-the-minute obfuscated malwares. This paper proposes a method to identify the malware variants based on the function-call graph. Firstly, the function-call graphs were created from the disassembled codes of program; then the caller–callee relationships of functions and the operational code (opcode) information about functions, combining the graph coloring techniques were used to measure the similarity metric between two function-call graphs; at last, the similarity metric was utilized to identify the malware variants from known malwares. The experimental results show that the proposed method is able to identify the obfuscated malicious softwares effectively.

1 Introduction

The attack and defense in the world of malicious softwares is an eternal topic. With the popularization of computer and Internet, the number of malwares increases dramatically. According to the Message Labs Intelligence: 2010 Annual Security Report [28] of Symantec, in 2010, there were over 339,600 different malware strains identified in emails blocked, representing over a hundredfold increase compared with 2009.

M. Xu (✉) · L. Wu · S. Qi · J. Xu · H. Zhang · Y. Ren · N. Zheng
College of Computer, Hangzhou Dianzi University, Hangzhou, China
e-mail: mxu@hdu.edu.cn

N. Zheng
e-mail: nzheng@hdu.edu.cn

In order to evade the detection of antivirus products, malware writers have to improve their skills in malware writing. Code obfuscation is to obscure the information so that others could not find the true meaning. Malware writers use this technique to obfuscate malicious code so that the malware could not be detected. Code obfuscation can easily change the structures of malware and keep the semantics of obscured programs and functionality invariant. With the advent of automated malware development toolkits, creating new variants from existed malware programs to evade the detection of anti-virus (AV) software has become relatively easy even for unskilled aggressors. These toolkits, such as Mistfall, Win32/Simile, and RPME [29], not only are a catalyst of the huge surge in the number of new malware threats but also can hinder the detection of malware in recent years. Effective antivirus techniques should be proposed to detect the obfuscated malware and mitigate the damages caused by malware.

Traditional methods for malware detection are mostly based on malware signature. It treats malware programs as sequences of bytes and performs well for known malwares. But those syntactic-based detection methods can be easily bypassed by simple code obfuscation because it ignores programs functionality or ignores their high-level internal structures, such as basic blocks and function calls. To more reliably recognize the syntactical difference and semantical identicalness between two malware programs, a high-level structure or an abstraction, for example function-call graph, should be regarded as a signature during detecting.

Function-call graph was created from the disassembled code of program, in which all vertexes represent functions included in the program and edges represent calls relationship among functions. Function-call graph can be used to classify or identify the malware variants based on similarity between two function-call graphs. Function-call graph

abstracts away byte or instruction level details, so that it is fit as an abstract signature to detect byte or instruction level obfuscation.

This paper shows how to analyze similarity between malware samples by extracting programs function-call graph using static analysis. Firstly the operation instruction sequences of the malware binary are converted into a function-call graph to extract the structure and functional characteristics and make byte or instruction obfuscation layers removed. Secondly, the similar score of two function-call graphs is computed via a novel graph matching technique based upon maximum common edges, which makes use of information from the matched vertexes in two function-call graphs, associating the caller–callee relationships among vertices and graph coloring techniques based on function opcodes (operational codes) comparison. Finally, the proposed method is evaluated by experiments.

The rest of this paper is organized as follows. Sect. 2 reviews related works. In Sect. 3, we introduce the function-call graph and the method of graph-creation. Section 4 presents various matching methods between functions included in function-call graph. The computing similarity method between function-call graphs is proposed in Sect. 5. Section 6 evaluates our method. In Sect. 7 limitations and future works are pointed out. We conclude in Sect. 8.

2 Related works

In order to make obfuscation detection more reliable, a great number of methods have been proposed in the past several years. Christodorescu et al. [7] defined a dependence graph as a malware signature and proposed a mining algorithm to construct the graph by use of dynamic analysis. Li et al. [23] extracted maximal pattern sequence from system call sequence, and used this pattern as a feature to compute similarity among malwares. Those papers discussed above are all based on dynamic analysis. Borello and Me [3] examined the use of advanced code obfuscation techniques with respect to metamorphic viruses. They proved that reliable static detection of a particular category of metamorphic viruses is an NP-complete problem, and then they empirically illustrated their result by constructing a practical obfuscator which could be used by metamorphic viruses.

There are also some solutions for metamorphic malware detection using static analysis. Gheorghescu [11] generated a CFG (control flow graph) by traversing the code of a program and used this graph as its characteristic. Kapoor and Spurlock [15] argued that comparing malwares on the basis of functionality is more effective than binary code comparison. Tian and Batten et al. [31] used Chi-square test to classify malwares based on function length, and the functions were obtained by IDA pro [12].

Karnik and Goswami et al. [16] used cosine similarity to compute similarities among functions of two malware programs, consequently, the similarity of two malware programs can be obtained. But a drawback of their method is that the scheme could be subverted by the instruction substitution. Kruegel and Kirda [20] proposed using structural information of executables to detect polymorphic worm. It used K-subgraphs and graph coloring techniques to complete detection based on the control follow graph.

Also, on the field of static analysis, some researchers focus on called libraries or system functions. Zhang and Reeves [35] exactly made use of libraries or system functions as patterns for detection. This approach utilized a backwards data flow analysis for a program, and used the intermediate representation of semantic instructions to obtain malware patterns. System-call graph was obtained using the algorithm proposed in [14] and used this graph to detect or classify malicious programs. Lee [22] improved the method of [14] by classifying API calls into 128 groups, and achieve faster analysis. In order to detect variants produced by metamorphic engine in [2], Borello and Me etc. [4] used a measure of similarity between program behaviors obtained by lossless compression of execution traces in terms of system calls. However, those method ignores the information of local functions, thus it may cause higher false positive. Furthermore, it may suffer from the code obfuscation techniques, for example inserting the meaningless system calls.

Function-call graph is wildly used in malware variants or obfuscated malwares recognition or classification. A function-call graph can be used to represent characteristics of a malware, thus the issue of finding similarity between the malwares can be regarded as finding similarity between function-call graphs. Bilar [1] first proposed the generative mechanisms of the call graph and using of the call graphs for detection problem. Shang [27] proposed an algorithm to compute similarity using function-call graph. Graph Edit Distance (GED) [9] is a very fundamental problem when we deal with graph-similarity metric. Bipartite graph matching and neighbor-based Hungarian algorithm are used to compute the GED in function-call graph proposed by Hu [13]. GED is also used in [18] when it classifies the malware families. Kostakis [19] proposed using simulated annealing instead of bipartite matching to improve the call graph matching. The experiment shows that this method outperformed previous approaches both in execution time and solution quality. Existing methods mainly fall into two categories. On one hand, they are technologies based upon the traditional graph matching, including graph isomorphism, MCS (maximum common subgraphs), and GED (graph edit distances). However, they are proven to be an NP-Complete problem [10], leading to expensive time and space expenditure. On the another hand, there is a matching theme that aims to compute the graph similarity on basis of graph maximum common vertexes or

edges [8,20,22,27]. These methods make full use of function information and the caller–callee relationships among functions to achieve the common maximum. Compared to the traditional graph matching techniques, this type of theme is less time-consuming and space-consuming.

3 Overview of the function-call graph

This section introduces the terminologies and notations used in our function-call graph. Also, the constructing method of function-call graph is put forward briefly.

3.1 Defining the function-call graph

A function-call graph $G = (V, E)$ is composed of a vertex set V and a edge set E, where the vertex in the graph is corresponding to the function included in program and the edge mean the caller–callee relationship between two functions. Expressly, vertex u is said to be a direct predecessor of v if $(u, v) \in E$. In this case, v is said to be a direct successor of u.

For a binary executable program, vertex functions can be classified into two types: local function, i.e. function written elaborately by writer to perform a specific effect; and external function, i.e. a system or library function. In here, the local functions are all beginning with "sub_xxxxxx proc near", and end with "sub_xxxxxx endup". The content of the middle is the opcode sequence. "sub_xxxxxx" represents its name, having various names in different programs even though they have similar function. However, for external functions, their names are consistent throughout all programs. Contrary to local functions, external functions never invoke any local function.

Definition (Outdegree and Indegree). The outdegree $d^+(v)$ of vertex v is the number of direct successors of u. Similarly, the indegree $d^-(v)$ is the number of direct predecessors of v.

Definition (Degree). The degree $d(v)$ of vertex v equals $d^+(v) + d^-(v)$.

Definition (out-neighborhood and in-neighborhood). The out-neighborhood (direct successor set) $N^+(v)$ of vertex v consists of the vertexes $\{w|(v, w) \in E\}$, while the in-neighborhood (direct predecessor set) $N^-(v)$ is the set $\{w|(w, v) \in E\}$, and $N(v)$ represents all the neighbors.

Definition (common function pair set). For two programs S and T, $G_S.V$ and $G_T.V$ represent the vertex set of function-call graph G_S and G_T, respectively. The common function pair set can be defined as:

$$com\_vex(G_S, G_T) = \{(u, v)|\forall u \in G_S, \exists v \in G_T, u \ mathches \ to \ v\} \quad (1)$$

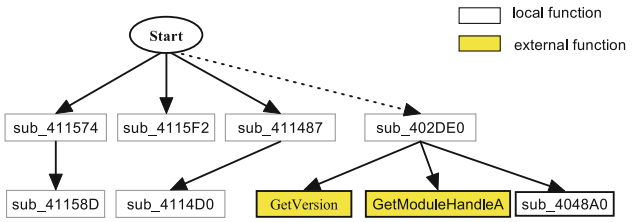

Fig. 1 Part function-call graph of Virus.Win32.Bolzano

Definition (common edge set). For two programs S and T, $G_S.E$ and $G_T.E$ represent the edge set of function-call graph G_S and G_T respectively. The common edge set is defined as follows:

$$com\_edge(G_S, G_T) = (u, v)|\forall(u, v) \in G_S, \exists(u', v') \in G_T, (u', u') \in com\_vex(G_S, G_T), and \ (v', v') \in com\_vex(G_S, G_T) \quad (2)$$

3.2 Constructing the function-call graph

To generate function-call graph from a concrete malware program, some pretreatment should be done. Firstly, PEiD [24] is used to check whether the malware file has been packed. If so, the packer envelope must be removed. Secondly, several tools, such as RL!dePacker or UPX [32], are used to unpack or decrypt the malware file. Then, utilizing the popular disassembler IDA Pro [12] disassembles the sample. After above steps, assembly code can be obtained and all function names are labeled.

The function-call graph is built up according to the assembly code. The constructed method is similar to the means proposed by Shang in [27]. It adopts a breadth-first approach to build the function-call graph, and the function-call graph is stored into an orthogonal list. It builds the caller–callee relationships with starting from the entry point functions. Due to the ignoring of the implicit function-call, some functions can not be inserted into the function-call graph. An example can be seen in Fig. 1, where dotted line represents the implicit function-call. According to [27], functions such as "sub_402DE0", "GetVersion", "GetModuleHandle" and "sub_4048A0" will be lost when constructing the function-call graph. Obviously, edges between them will be lost too. In order to avoid this phenomenon, some improvement should be done. The graph-creation algorithm should insert each function which has not been inserted into the function-call graph. This is to make sure that each function in the malware will be corresponding to a single vertex in the function-call graph. If two functions have call reference between them, an edge is added to the function-call graph. After treating, only the edge (implicit invocation) between the functions of "start" and "sub_402DE0" is lost.

For each local function, we preserve its opcode sequence from instructions. This sequence will be used in the latter matching process. At the same time, each function name is saved too.

4 Matching between two functions

If two functions are similar enough, they will be regarded as a matched function pair. Function matching is a process to determine whether a function in one function-call graph matches another function in another function-call graph. There are many algorithms used for matching functions. The methods based on precise string matching often suffer from the code obfuscated technology; moreover, some more resilient algorithms to the code obfuscation techniques, for example cosine similarity, often bring the function mismatch. To solve this problem, a new matching algorithm should be proposed.

External functions, stored in a library as part of an operating system, have uniform standard. Therefore, external function matching is just according to their symbolic names.

Local functions, the most frequently occurring functions in any program, were arbitrarily designed by the program writer according to his own purpose. So, we cannot judge their matching only based their function names, because two same local functions from the different executable programs could have the different function name. Using inner instruction-level information to compute the similarity between local functions is reasonable. Moreover, in order to bypass the detecting by antivirus products, code obfuscation techniques are wildly used in malwares. This paper concentrates on opcode sequence to match local functions and presents a graph coloring technique to deal with the code obfuscation mechanism.

4.1 Matching external functions based on their same names

External function matching is just according to their symbolic names, i.e., two external functions are regarded as a matched function pair, only if they have the same symbolic names.

4.2 Matching local functions based on the same called external functions

When two local functions call two or more of the same external functions, the two local functions will be regard as matching. This method is just like call tree signature which was proposed in [6].

4.3 Matching local functions based on their opcodes

In order to evaluate two local functions are similar enough based on opcode, a color similarity method and relaxed color similarity method for vertex are proposed based on a colored function-call graph.

Each vertex in the function-call graph will be colored in the light of the instructions used in this function. X86 Instructions will be classified as 15 classes according to their functions as shown in Table 1. A 15-bit color variable is defined to describe a color for each vertex and the initial value is 0. Each bit corresponds to a certain class of instructions. If a instruction appears in the function, the corresponding bit in color variable will be set to 1. At the same time, the number of the class instructions in the function are held into the corresponding element in a vector. Each dimension in the vector represents the number of the corresponding class instructions appearing in this function.

For example, the color variable and vector of a local function from the malware Backdoor.Win32.DarkMoon. ab (in Fig. 2) are shown in Table 2. In this table, each symbol from C_1 to C_{15} means each class of X86 instructions.

Given two vectors $\mathbf{X} = (x_1, x_2, ..., x_{15})$ and $\mathbf{Y} = (y_1, y_2, ..., y_{15})$ of two local functions respectively, the cosine similarity between them can be calculated through the following formula:

$$sim(\mathbf{X}, \mathbf{Y}) = \frac{\sum_{i=1}^{15} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{15} x_i^2} \cdot \sqrt{\sum_{i=1}^{15} y_i^2}} \tag{3}$$

If the color variable values are the same and the cosine similarity is greater than or equal to a predefined threshold

Table 1 The classified X86 instructions

Class	Name	Description
C_1	Data	Transfer data transfer such as mov
C_2	Stack	Stack operation
C_3	Port	In and out
C_4	Lea	Destination address transmit
C_5	Flag	Flag transmit
C_6	Arithmetic	Incl. shift and rotate
C_7	Logic	Incl. bitbyte operations
C_8	String	String operations
C_9	Jump	Unconditional transfer
C_{10}	Branch	Conditional transfer
C_{11}	Loop	Loop control
C_{12}	Halt	Stop instruction execution
C_{13}	Bit	Bit test and bit scan
C_{14}	Processor	Processor control
C_{15}	Float	Floating point operations

```
sub_4059DC  proc near
push   ebx
mov   ebx, eax
cmp   ds:byte_4146C1, 0
jz      short loc_405A04
push  0
call   SwapMouseButton
mov  ds:byte_4146C1, 0
mov  eax, ebx
mov  edx, offset dword_405A28
call   sub_403DEC
pop   ebx
retn
push  0FFFFFFFFh
call   SwapMouseButton
mov  ds:byte_4146C1, 1
mov  eax, ebx
mov  edx, offset dword_405A28
call   sub_403DEC
pop   ebx
retn
sub_4059DC      endp
```

Fig. 2 A function of Backdoor.Win32.DarkMoon.ab

α, the two local functions from two different executable programs are regarded as a color similar function pair.

A graph coloring algorithm $color\_similar()$ sketches the above steps shown in algorithm 1. Here two local functions f_1 and f_2 are used to compute the similarity. For each function, the color variable value is gained using "$GetColor()$"(in line 1 and 2) and vector is gained using "$GetVector()$"(in line 4 and 5). Line 3 means that, if two functions have a same color, the similarity between them will be computed. Finally, the similarity value between f_1 and f_2 is calculated in line 6.

In order to make more accurate matching, two constraints about the length and the degree of functions should also be considered. Therefore, whether two functions are a matching pair or not is ultimately determined through three factors. That is to say, when the color similarity value from graph coloring algorithms is greater than or equal to a predefined threshold α, function length similarity value (measured by $len\_sim()$, in Eq. (2)) is greater than or equal to a predefined threshold β, and degree similarity value (measured by $degree\_sim()$ in Eq. (3)) is greater than or equal to a predefined threshold γ, the two functions will be regarded as matched based on their opcodes.

Algorithm 1: The $color\_sim()$ algorithm

Input: two functions f_1 and f_2
Output: the similarity between f_1 and f_2

```
1 f1.color ← GetColor(f1.sequence);
  // get the f1 color variable
2 f2.color ← GetColor(f2.sequence);
  // get the f2 color variable
3 if (f1.color = f2.color) then
4 |   f1.vector ← GetVector(f1.sequence);
  |   // get the f1 vector
5 |   f2.vector ← GetVector(f2.sequence);
  |   // get the f2 vector
6 |   color_sim ← cosine_similarity(f1.vector, f2.vector);
7 end
8 return color_sim;
```

$$len\_sim(p,q) = \begin{cases} \frac{len(p)}{len(q)} & \text{if } len(p) \le len(q) \\ \frac{len(q)}{len(p)} & \text{otherwise} \end{cases} \tag{4}$$

$$degree\_sim(p,q) = \begin{cases} 1 & \text{if } d(p) = d(q) \\ \frac{1}{|d(p)-d(q)|} & \text{otherwise} \end{cases} \tag{5}$$

Where p and q represent the functions in G_S and G_T, and $len(p)$ returns the length of function p and $d(p)$ denotes the degree of function p. In this work, the thresholds α, β, γ are selected as 0.98, 0.83, and 0.5 respectively based on the experimental result.

4.4 Matching local functions based on their matched neighbors

As we know, for matched vertexes, it is likely that their neighbors are matched too. Based on this fact, when the direct predecessor or successor of two local functions have been matched, the two local functions could be regarded as candidate vertexes to match each other.

In Fig. 3, this is an example about how to choose the candidate successor vertexes to match. $\{a, b, c, m\}$ and $\{d, e, n\}$ are the successor vertex set of u and v respectively. The same colored vertexes mean that they have been matched. So "a, b, c" and "d, e" are regarded as the candidate vertexes to match each other.

The pseudo-code about $successorMatch()$ is given in algorithm 2. In this algorithm, a breadth-first approach is used to traverse each vertex in function-call graph, and a FIFO (first in first out) function queue ($vexQueue$ in line 1) is used

Table 2 The color and vector of function

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}
Color variable	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0
Vector	7	4	5	1	0	0	0	0	0	0	1	0	0	0	0

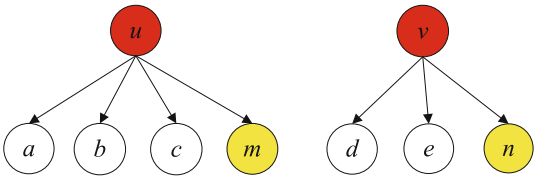

Fig. 3 Candidate vertexes for matching

to assist to realize this matching process. Line 1 initializes the queue with the common function pair set (means to insert all the common function pairs $com\_vex(G_S, G_T)$ into the $vexQueue$ in line 1). While the queue is not empty, the head element (in line 3) will be popped from $vexQueue$. From line 4 to 13, each successor candidate vertex pair will be traversed. If their color relaxed similar scores, $color\_relaxed\_sim()$, is greater than or equal to the predefined threshold α, they will be regarded as a matched function pair. Here, the color relaxed similar algorithm, $color\_relaxed\_sim()$, is almost same as to algorithm 1 in Sect. 4.3, but it does not require that $f_1.color$ is equal to $f_2.color$. The two local functions from two different executable programs are regarded as a color relaxed similar function pair, if the cosine similarity value is greater than or equal to the predefined threshold δ (In this work, the thresholds δ is selected as 0.97 based on the experimental result), but the color variable values are different.

Pseudo-code about $predecessorMatch()$ is similar to algorithm 2. In fact, it can be implemented by using all the predecessor neighborhoods to replace the successor neighborhoods in algorithm 2, i.e. $N_S^+(u)$ and $N_T^+(v)$ are replaced by $N_S^-(u)$ and $N_T^-(v)$ respectively at line 4 and 5 in algorithm 2.

Algorithm 2: The $successorMatch()$ algorithm

```
Input: the common function pairs
       com_vex(G_S, G_T) and G_S, G_T, U_S.V, U_T.V
Output: the common function pairs com_vex(G_S, G_T)
// initialize the queue using common
   function pairs
1  vexQueue ← InitVexQueue(com_vex(G_S, G_T));
// match functions using breadth traversal
2  while (vexQueue ≠ ∅) do
3      (u, v) ← vexQueue.pop();
4      foreach G_S.V_i ∈ N_S^+(u) ∩ U_S.V do
5          foreach G_T.V_j ∈ N_T^+(v) ∩ U_T.V do
6              if (color_relaxed_sim(G_S.V_i, G_T.V_j) ≥ δ) then
7                  com_vex(G_S, G_T) ←
                   com_vex(G_S, G_T) ∪ {(G_S.V_i, G_T.V_j)};
8                  U_S.V ← U_S.V − G_S.V_i;
9                  U_T.V ← U_T.V − G_T.V_j;
10                 vexQueue.push(G_S.V_i, G_T.V_j);
11             end
12         end
13     end
14 end
15 return com_vex(G_S, G_T);
```

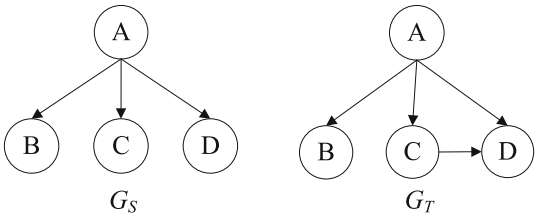

Fig. 4 An example of function-call graphs similarity

5 The similarity between function-call graphs

The basic idea is using the maximum common vertexes and edges to compute the similarity between two function-call graphs.

The similarity metric $sim_G(G_S, G_T)$ of function-call graph G_S and G_T is proposed as follows:

$$sim_G(G_S, G_T) = \frac{2|com\_edge(G_S, G_T)|}{|E(G_S)| + |E(G_T)|} \times 100\% \quad (6)$$

It means the ratio of the same edges from two function-call graphs. In this formula, $|com\_edge(G_S, G_T)|$ represents the number of common edges in function-call graph G_S and G_T. $|E(G_S)| + |E(G_T)|$ represents the total number of edges of function-call graph G_S and G_T. Obviously, for any G_S and G_T, $sim_G(G_S, G_T) \in [0, 1]$. The more $sim_G(G_S, G_T)$ is close to 1, the more G_S and G_T are similar.

For example, in Fig. 4, G_S is different from G_T. But if we only consider vertex information of two graphs, the similarity of G_S and G_T would be 100 %. To measure the similarity more precisely, edge information should be considered, because edges are able to represent characteristic of graph more than vertexes. Here, G_S has 3 edges, while G_T has 4 edges. And the number of the same edges of G_S and G_T are 3; the total number of edges is 7. Thus, the similarity of G_S and G_T is 85.71 %.

The whole function-call graph similarity algorithm is listed in the algorithm 3. At a high level, the whole function-call graph similarity algorithm is divided into five parts.

Part 1, Matching external functions (line 1 to 13 in algorithm 3). As we know, every external function shares a same name in different programs. Therefore, external functions which have the same name will be regarded as the same function. According to this fact, all the external functions which have the same name in function-call graph G_S and G_T will be regarded as the common function pairs, and these common external function pairs are preserved into the $com\_vex(G_S, G_T)$.

Part 2, Matching local functions by the same called external functions (line 14 to 24 in algorithm 3). If there is no matched external function, the algorithm bypasses this step and enters into the Part 3. In this part, algorithm will search for matched local function pair that call two or more of the

same external functions. Here, the "$InvokeExternalFun(U_S.V_i)$" returns the set of external function called by $U_S.V_i$. The matched local function pairs based on the same called external functions will be added into the $com\_vex(G_S, G_T)$.

Part 3, Matching local functions based on the opcodes (line 25 to 33 in algorithm 3). The idea of this matching process has been described in Sect. 4.3. It aims to match local functions based on their inner opcodes. If two functions meet the matching conditions, they are considered as a matched function pair based on opcodes. The matched local function pairs based on opcodes will be added into the $com\_vex(G_S, G_T)$.

Part 4, Matching local functions based on their matched neighbors (line 34 to 35 in algorithm 3). This part exploits a neighbor-biased approach to expand the matching result. If the majority of their neighbors have been matched, it is possible for two function vertexes to be matched also. As described in Sect. 4.4, both directions (successor and predecessor) should be considered. The algorithm "$successorMatch()$" (described in the algorithm 2) and "$predecessorMatch()$" implement the matching process in both directions respectively. Two algorithms find the matched local function pairs based on their matched neighbors, and add them into the $com\_vex(G_S, G_T)$.

Part 5, Calculating the similarity of two funct-ion-call graphs (line 36 to 37 in algorithm 3). In this part, the similarity between two function-call graphs will be calculated. First, the maximum common edge set $com\_edge(G_S, G_T)$ between two function-call graphs can be obtained via the "$ExtractComEdges()$" on the basis of the set of matched function pairs $com\_vex(G_S, G_T)$ (line 36). Then, the similarity between two function-call graphs is ultimately determined through the formula (6) (line 37).

The space complexity about this algorithm is $O(2{\cdot}|G.V|+|G.V|\cdot|G.E|)$, and the time complexity has various situations. In the best case, if two graphs only contain external functions, the time complexity about matching functions is $O(|G_S.V|\cdot|G_T.V|)$; and the worst time complexity is $O(m\cdot n\cdot|G_S.V|\cdot|G_T.V|)$, where m and n represent the average value about vertex neighbor number in $G_S.V$ and $G_T.V$ respectively, i.e. $m=\sum_{v\in G_S.V}|N(v)|/|G_S.V|$, $n=\sum_{v\in G_T.V}|N(v)|/|G_T.V|$. In the phase of calculating the similarity via the formula (6), it will take up $O(|com\_vex(G_S, G_T)|^2)$ time complexity, where $|com\_vex(G_S, G_T)|$ denotes the number of the matched vertex pairs.

In order to more accurately and rationally reveal the function-call graph, we proposed another similarity metric method without the constitutive property of graph. As described above, a program can be regarded as a series of functions. Generally speaking, if two programs have the sameor similar functions, they could be high similar

Algorithm 3: The function-call graph similarity

Input: function-call graph G_S and G_T
Output: the similar value $sim(G_S, G_T)$

```
// Part 1: matching external functions
1  ExternalFunSet_S ← ExtractExternalFun(G_S.V);
2  ExternalFunSet_T ← ExtractExternalFun(G_T.V);
3  U_S.V ← G_S.V;
4  U_T.V ← G_T.V;
5  foreach u_i ∈ ExternalFunSet_S do
6      foreach v_j ∈ ExternalFunSet_T do
7          if (u_i.name = v_j.name) then
8              com_vex(G_S, G_T) ←
               com_vex(G_S, G_T) ∪ {(u_i, v_j)};
9              U_S.V ← U_S.V − {u_i};
10             U_T.V ← U_T.V − {v_j};
11         end
12     end
13 end
   // Part 2: matching local functions by the
      same called external function
14 if (com_vex(G_S, G_T) ≠ ∅) then
15     foreach G_S.V_i ∈ U_S.V do
16         foreach G_T.V_j ∈ U_T.V do
17             if (|InvokeExternalFun(U_S.V_i) ∩
               InvokeExternalFun(U_T.V_j)| ≥ 2) then
18                 com_vex(G_S, G_T) ←
                   com_vex(G_S, G_T) ∪ {(G_S.V_i, G_T.V_j)};
19                 U_S.V ← U_S.V − {G_S.V_i} ;
20                 U_T.V ← U_T.V − {G_T.V_j} ;
21             end
22         end
23     end
24 end
   // Part 3: matching local functions
      according to opcodes
25 foreach G_S.V_i ∈ U_S.V do
26     foreach G_T.V_j ∈ U_T.V do
27         if (color_sim(G_S.V_i, G_T.V_j) ≥
           α ∩ len_sim(G_S.V_i, G_T.V_j) ≥
           β ∩ degree_sim(G_S.V_i, G_T.V_j) ≥ γ) then
28             com_vex(G_S, G_T) ←
               com_vex(G_S, G_T) ∪ {(G_S.V_i, G_T.V_j)};
29             U_S.V ← U_S.V − {G_S.V_i};
30             U_T.V ← U_T.V − {G_T.V_j} ;
31         end
32     end
33 end
   // Part 4: matching local functions by
      matched neighbors
34 com_vex(G_S, G_T) ←
   successorMatch(com_vex(G_S, G_T), U_S, U_T, G_S, G_T);
35 com_vex(G_S, G_T) ←
   predecessorMatch(com_vex(G_S, G_T), U_S, U_T, G_S, G_T);
   // Part5: calculating function-call graph
      similarity
36 com_edge(G_S, G_T) ←
   ExtractComEdges(com_vex(G_S, G_T), G_S, G_T);
37 sim(G_S, G_T) ← 2 × com_edge(G_S, G_T)/(|G_S.E| + |G_T.E|);
38 return sim(G_S, G_T);
```

also. From the semantic characteristic, another similarity $sim_F(P_1, P_2)$ is defined as the formula (7).

$$sim_F(P_1, P_2) = \frac{2|F_1 \cap F_2|}{|F_1| + |F_2|} \times 100\ \% \tag{7}$$

Here, F_1 and F_2 represent the function set of program P_1 and P_2 respectively; $|F_1|$ and $|F_2|$ denotes the number of functions in two programs respectively; and $|F_1 \cap F_2|$ is the number of the intersection set of F_1 and F_2. The $F_1 \cap F_2$ is mainly obtained through the matching methods shown in the Sects. 4.1 and 4.3. This method does not use the caller–callee relationship or internal structure information of graph, and only use information about function set. In contrast to the function-call graph, this method ignores the correlation between functions and is easily attacked by code obfuscation.

6 Evaluation

A series of experiments: the similarity between malware variants, distinguishing the different malware families, and distinguishing the malicious programs from benign programs, are designed for evaluating effectiveness and correctness of the proposed method. Tested malware mutant set were downloaded from VX Heavens [34]. At the same time, some benign programs were also collected for experiments.

6.1 Evaluating the similarity between malware variants

More than 400 malware variants pairs are chosen randomly to evaluate the similarity between malware variants. Almost all types of malicious software are included, such as virus, backdoor, worm, torjan horse, rootkit and so on. Using the proposed method, the similar scores among these samples are computed based on the formula (6). The statistical result is shown in Fig. 5.

According to Fig. 5, there are a great number of pairs about (80 %) belong to the similar score [0.4–1]. Here, the similar scores about 6 % pairs range from 0 to 0.2. It suggests that

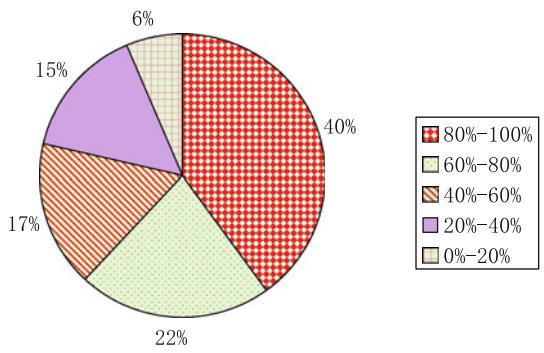

Fig. 5 Similarity statistics of variants pairs

Table 3 Similarity between Sality variants

%	Sality.a	Sality.c	Sality.d	Sality.e	Sality.f	Sality.g
Sality.a	**100**	99.67	96.01	62.79	40.86	87.04
Sality.c	99.32	**100**	95.70	62.58	40.73	86.76
Sality.d	99.31	98.87	**100**	64.51	41.98	88.40
Sality.e	66.46	65.96	68.02	**100**	58.05	68.25
Sality.f	60.61	53.75	60.48	89.70	**100**	43.80
Sality.g	81.73	87.79	83.55	66.46	64.62	**100**

Bold values indicate the variants 100 % similar to itself at diagonal

the function-call graph effectively represents the signature of the malware sample, and the similarity metric rightly depicts the common characteristics among malware variants.

The reason why there exist low-score pairs is that the sizes of malwares and variants are so largely different that the number of vertexes and edges may be observably different. For instance, the number of functions belong to Email-Worm.Win32.Mimail.m is 128, but there are only 48 functions in Email-Worm.Win32.Mimail.q. Furthermore, the number of edges are 245 and 60, respectively. There are 17 common edges (34 common functions) according to the proposed method. Then, the similar score is only 11.1 %.

Next, the similar value based on the formula (6) between the variants from several specific malware families will be listed.

Virus.Win32.Sality is a family of file infecting viruses that spread by infecting.exe and. scr files. These viruses also include an autorun worm component that allows it to spread to any removable or discoverable drive. In addition, these viruses also include a downloader trojan component that installs additional malwares via the Web. Variants of the Sality family have long been ranking the top 20 in the Monthly-Malware-Statistics [17]. In Table 3, the similarity matrix shows the similar scores between each pair of the malicious binaries. The values (100 %) in the main diagonal are similar scores that the variants compare with themselves. Our experimental results are shown below the main diagonal, and the data above the main diagonal are results from [27]. On the whole, similar scores in our proposed method are higher, thus our method performs better than method in [27].

The similar scores of Email-Worm.Win32.Klez samples comparing with [6] are shown in Table 4a, b. Email-Worm.Win32.Klez, propagating through e-mails and LAN, can modify and delete parts of the system files and user programs, and make system cannot run normally. Scores below the main diagonal are computed according our proposed method and scores above the main diagonal are belong to [6]. Symbol "N/A" in the table represents the similar score is absent because the sample is unavailable in VX heaven.

Backdoor.Win32.DarkMoon is a Trojan horse that opens a back door on compromised computer and has keylogging

Table 4 Similarity between Klez variants

%	Klez.a	Klez.b	Klez.c	Klez.d	Klez.e
(a) Similarity from Klez.a to Klez.e					
Klez.a	**100**	79.60	70.30	70.30	49.30
Klez.b	94.40	**100**	73.40	73.40	49.30
Klez.c	99.35	94.84	**100**	88.20	49.60
Klez.d	79.73	90.75	81.45	**100**	49.60
Klez.e	N/A	N/A	N/A	N/A	**100**
%	Klez.f	Klez.g	Klez.h	Klez.i	Klez.j
(b) Similarity from Klez.f to Klez.j					
Klez.f	**100**	87.00	80.70	80.70	87.00
Klez.g	98.98	**100**	80.70	80.70	87.00
Klez.h	93.04	92.92	**100**	89.60	80.70
Klez.i	93.09	92.97	99.06	**100**	80.70
Klez.j	99.10	98.98	92.81	92.87	**100**

Bold values indicate the variants 100 % similar to itself at diagonal

Table 5 Similarity between DarkMoon variants

%	∗.ab	∗.ad	∗.ae	∗.ah	∗.ai	∗.ar	∗.at	∗.aw
∗.ab	**100**	75.09	78.25	70.34	69.41	75.08	55.52	71.57
∗.ad	75.09	**100**	92.77	92.14	91.42	86.99	74.92	78.01
∗.ae	78.25	92.77	**100**	90.86	88.04	92.08	65.57	83.26
∗.ah	70.34	92.14	90.86	**100**	95.46	91.36	76.41	76.81
∗.ai	69.41	91.42	88.04	95.46	**100**	94.38	76.54	78.25
∗.ar	75.08	86.99	92.08	91.36	94.38	**100**	73.04	82.24
∗.at	55.52	74.92	65.57	76.41	76.54	73.04	**100**	85.10
∗.aw	71.57	78.01	83.26	76.81	78.25	82.24	85.10	**100**

Bold values indicate the variants 100 % similar to itself at diagonal

capability [26]. In the following matrix in Table 5, "∗" means DarkMoon. All of the scores listed in this matrix are our experimental results and the matrix is symmetric. According to this matrix, we can see that all the similar scores are greater than 55 %.

Virus and backdoor are all considered above. In order to validate the performance of our method, some rootkits were chosen for experiment. In Table 6, the mutations of the Rootkit.KernelBot family [25] were tested. Rootkit.KernelBot is a program that hides the malicious files, processes and registry entries on infected machine. Here, "∗" means KernelBot. From the Table 6, the smallest score is 45.1 % (between KenerlBot.as and KenerlBot.al). This score seems a little low, but for distinguishing the different malware families, it is high enough. The experiments in Sect. 6.2 will show it.

Table 7 show the similar scores between Virus.Win32.Champs. Here "∗" represent the Champ.Virus.Win32.Champ [33]. It is a dangerous nonmemory resident parasitic polymorphic Win32 virus. It infects the PE EXE files (Win32 executable file). We can observe that most of similar values

Table 6 Similarity between KernelBot variants

%	∗.a	∗.ah	∗.ak	∗.al	∗.as	∗.at	∗.aw	∗.az
∗.a	**100**	95.00	65.88	73.91	64.16	63.77	95.89	89.86
∗.ah	95.00	**100**	62.92	70.83	48.85	60.27	90.91	84.93
∗.ak	65.88	62.92	**100**	91.09	48.42	58.97	63.41	71.79
∗.al	73.91	70.83	91.09	**100**	45.10	54.12	69.66	65.88
∗.as	51.16	48.89	48.42	45.10	**100**	78.48	48.19	53.16
∗.at	63.77	60.27	58.97	54.12	78.48	**100**	60.61	67.74
∗.aw	95.89	90.91	63.41	69.66	48.19	60.61	**100**	87.88
∗.az	89.86	84.93	71.79	65.88	53.16	67.74	87.88	**100**

Bold values indicate the variants 100 % similar to itself at diagonal

about variants pairs are greater than 90 %, only Champ.7001 with other members has lower similarities.

6.2 Distinguishing the different malware families

The aim of this section is to evaluate the ability of distinguishing different malware families. 555 variant pairs and 541 non-variant pairs are randomly chosen as test set. These malwares include virus, worms, backdoors and Trojans.

Figure 6 shows the distribution of similarity based on the function-call graph in this experiment. Here, x-axis denotes the variants and non-variants pairs, and y-axis denotes the similarity of each pairs. We can intuitively observe that the variants have higher similar scores than non-variants. Similar scores of most non-variants are very close to 0, but some specific pairs have the scores near to 0.2. It may be that these malwares even in different families have some common codes. For variants, most of pairs have the similar scores more than 0.2, but there exist some scores that lower than 0.2, even close to 0.1. We observe these samples and find out that the number of functions in most of them are very different, and this may be caused by incomplete unpacking.

The distribution of similar value based on the function set from the formula (7) is shown in the Fig. 7. Likewise, x-axis denotes the variants and non-variants pairs, and y-axis denotes the similarity of each pairs. From the figure, the similar values of the most variants belong to [0.4–1.0] while similar scores of non-variants are from 0 to 0.4. For this method based on function set, although all of the similar values of variant pairs and non-variant pairs have been increased, the similar scores of variants are generally greater than than the non-variants. Both methods based on function-call graph and function set provide a good distinguishing way between variants and non-variants and have the ability to distinguish different malware families.

Using the same testing data set, a ROC curve (Receiver Operating Characteristic curve) based on two similarity metric methods can be obtained (in Fig. 8) to show which is better. ROC curve is a graphical plot of the TPR (True Positive

Table 7 Similarity between Champ variants

%	∗	∗.5430	∗.5447	∗.5464	∗.5477	∗.5521	∗.5536	∗.5714	∗.5722	∗.7001
∗	**100**	92.40	92.40	97.67	95.29	95.91	94.55	100	97.65	62.58
∗.5430	92.40	**100**	100	94.74	97.04	96.47	91.46	92.40	93.49	66.67
∗.5447	92.40	100	**100**	94.74	97.04	96.47	91.46	92.40	93.49	66.67
∗.5464	97.67	94.74	94.74	**100**	97.65	98.25	93.33	97.67	96.47	65.03
∗.5477	95.29	97.04	97.04	97.65	**100**	99.41	93.25	95.29	95.24	67.08
∗.5521	95.91	96.47	96.47	98.25	99.41	**100**	92.68	95.91	94.67	66.67
∗.5536	94.55	91.46	91.46	93.33	93.25	92.68	**100**	93.33	96.93	62.82
∗.5714	100	92.40	92.40	97.67	95.29	95.91	93.33	**100**	97.65	62.58
∗.5722	97.65	93.49	93.49	96.47	95.24	94.67	96.93	97.65	**100**	64.60
∗.7001	62.58	66.67	66.67	65.03	67.08	66.67	62.82	62.58	64.60	**100**

Bold values indicate the variants 100 % similar to itself at diagonal

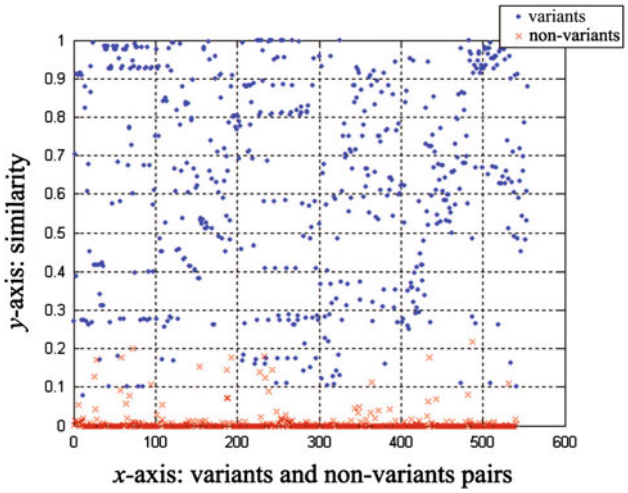

Fig. 6 The similarity of variants pairs and non-variants pairs based on function-call graph

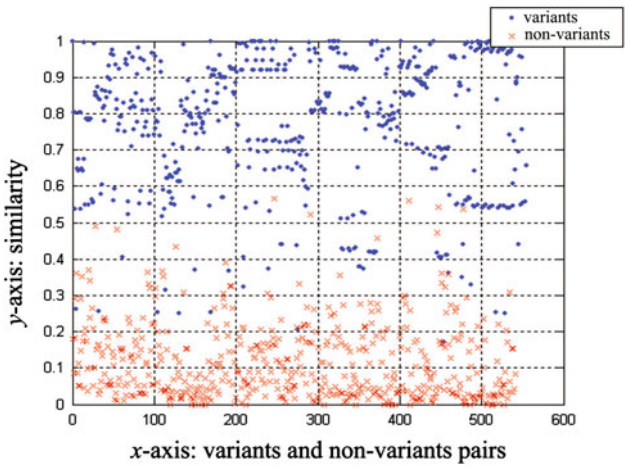

Fig. 7 The similarity of variants pairs and non-variants pairs based on function set

Rate) versus FPR (False Positive Rate), for a binary classifier system as its discrimination threshold is varied. The closer the ROC curve is to the upper left corner, the higherthe overall accuracy of classifying. In Fig. 8, abscissa 1-Specificity means FPR and the ordinate sensitivity means TPR. The solid curve indicates that the similar values are based on the function set, and the dash-dot curve denotes the similar values are based on the function-call graph. We can intuitively observe that two ROC curves are very close to the upper left corner, but the method based on function-call graph is better than the method based on function set. In contrast to the function set, the function-call graph can reflect the structural property of malware program more correctly.

Next, an instance is used to illuminate the recognising capability of our method. In a general way, if a malware has n variants, anti-virus scanner based signature would require n different signatures to recognize them. The experiment demonstrates that Email-Worm.Win32.Wozer.e, Wozer.f and Wozer.g were recognized triumphantly by only using the signature of Wozer.c, when the threshold is set as 0.15. There variants could be identified because the similar scores with Wozer.c are still greater than the threshold. The same trend occurs at other malwares, such as Mimail, DarkMoon, Bagel, MyDoom, and Klez, shown in Table 8. Here Mi.a, Sa.c, Da.ab, Ba.a, My.a, and Kl.d represent the malware Mimail.a, Sality.c, DarkMoon.ab, Bagle.a, Mydoom.a, and Klez.d respectively.

It means that the proposed method is better than the most current commercial anti-virus software at the recognition capability. To new malware variants, the possibility of recognizing them as malice is higher than the most current commercial anti-virus software only according the original malware. Moreover, our method can reduce the size of feature set.

6.3 Distinguishing the malicious programs from benign programs

In order to test the usability of our method, benign programs are necessary to be considered. 11 malicious programs and 11 legitimate binary programs are collected. For easy to

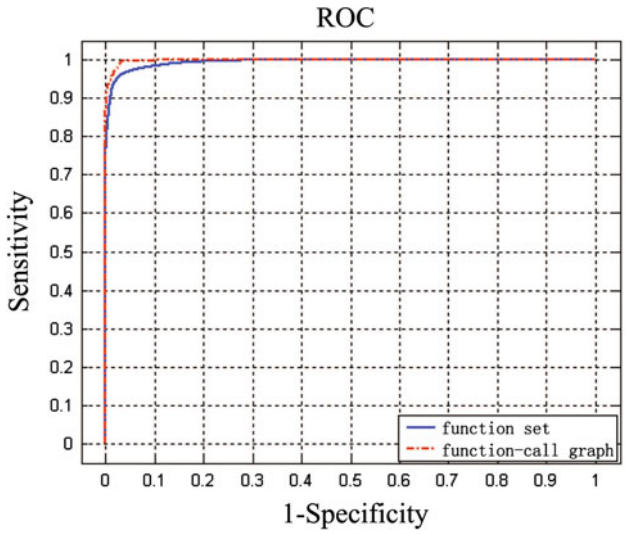

Fig. 8 A ROC curve

Table 8 Variants detecting by one signature set for each strand of malware

Class	Signatures used					
	Mi.a	Sa.c	Da.ab	Ba.a	My.a	Kl.d
Mimail.l	√	×	×	×	×	×
Mimail.m	√	×	×	×	×	×
Sality.h	×	√	×	×	×	×
Sality.i	×	√	×	×	×	×
Sality.j	×	√	×	×	×	×
DarkMoon.g	×	×	√	×	×	×
DarkMoon.j	×	×	√	×	×	×
DarkMoon.m	×	×	√	×	×	×
Bagle.b	×	×	×	√	×	×
Bagle.i	×	×	×	√	×	×
Bagle.j	×	×	×	√	×	×
MyDoom.e	×	×	×	×	√	×
MyDoom.g	×	×	×	×	√	×
MyDoom.l	×	×	×	×	√	×
Klez.h	×	×	×	×	×	√
Klez.j	×	×	×	×	×	√

compare, these samples are same as the data set used in [18]. The concrete data are listed in Table 9. The similar scores are shown in Fig. 9. Here, every coordinate (x, y) represents a binary pair (it is symmetric, i.e. (x, y) and (y, x) represent the same pair), and the z-axis denotes the similar score. For example, in the coordinate (9,9), which represents the binary pair Kido.dam.x and Kido.h, and the similar score is 1 (shown in z-axis). Similarly, the similar value is 0.6537 in coordinate (17,17) (Sality.d and Sality.e), and 0.74 in coordinate (21,21) (IE7 and IE8). Besides these 3 pairs, there are lsass and pid (similar score 0.1125), putty_0.60 and ICQ-Pager.b (similar score 0.1877).By investigating, the relative high similarities of these pairs is due to share some common functions. In the majority cases, the similarity is below 0.1 except the above mentioned pairs. Comparing with the experiment result shown in [27] (the lowest similar score of variant pairs is 0.6451 and the highest similar score between non-variant pairs is 0.2579), our method has a stronger distinguishing capacity.

Table 9 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

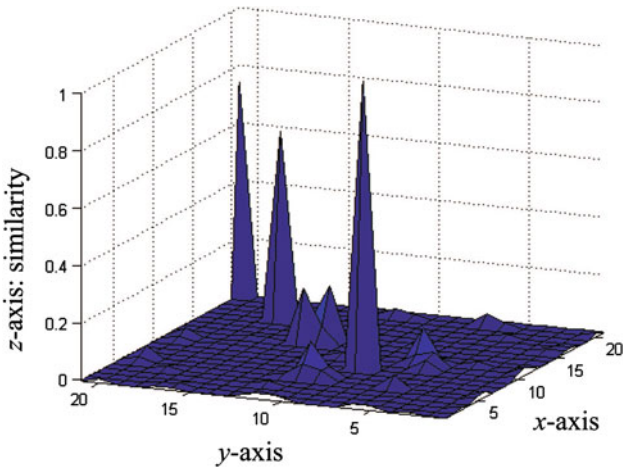

Fig. 9 Similarities among hybrid samples

7 Limitations

The encouraging experimental results mean that the proposed method in this paper is capable to recognize the malware variants or obfuscated malwares. But it also exists some defects. Similar computing method in this paper is based on static analysis. Moreover malware is commonly packed and/or encrypted. Therefore the main inconvenience is to unpack and/or decrypt malwares before constructing the function-call graph. Actually, the experiments have shown

that some malware samples are difficult to unpack and/or decryption correctly. Furthermore, the false or error assemble code will lead to the function-call graph incomplete, and make the similar score incorrect.

The proposed method depends on the function-call graph. A complete function-call graph is very important to our method. When constructing the function-call graph, implicit or obfuscated calls [21] are ignored, then these call relationships between functions can not be extracted. Implicit call for the proposed method is a potential threat. Additionally, the correctness and standardization must be considered [5]. The function-call graph can be heavily twisted by some specific obfuscation schemes, such as inserting fake conditional and unconditional jumps, or replacing call instructions with a sequence of instructions that do not alter the functionality [30]. Therefore, some transformations are essential to deal with this limitation in future work.

In future work, we also try to find the implicit invocation between functions and improve the unpack and decrypt method.

8 Conclusions

This paper presents a similar computing method between two malware variants based on function-call graph and their inner opcodes.

In order to improve the distinguishing capacity for malware variants, this paper mends the graph-creation algorithm proposed in [27] and makes the function-call graph more integrate. In the part of function matching, function opcode information and the function-call graph are combined to find out all of the common function pairs between two function-call graphs. This process is divided into four parts: matching the external functions which have the same name, matching the local functions based on the same called external function, matching the local functions via their opcodes based on graph coloring techniques, and matching the local functions based on their matched neighbors. Finishing these works, a similarity metric between two function-call graphs based upon maximum common matched edges is proposed, which makes full use of the matched vertexes obtained in the previous process. In order to illustrate the effectiveness of our method, another similarity metric method based on function set is presented also.

At last, this paper designs a series of experiments: the similarity between malware variants, distinguishing the different malware families, and distinguishing the malicious programs from benign programs, to evaluating effectiveness and correctness of the proposed method. Through experiment results, we can conclude that the proposed method can effectively recognize the malware variants or obfuscated malwares.

Acknowledgments This paper is supported by NSFC of China (No. 61070212, 61003195); Natural Science Foundation of Zhejiang Province, China (No. Y1090114, No. LY12F02006); the State Key Program of Major Science and Technology (Priority Topics) of Zhejiang Province, China (No 2010C11050). We would like to thank the anonymous reviewers for their helpful comments, suggestions, explanations, and arguments.

References

1. Bilar, D.: On callgraphs and generative mechanisms. J. Comput. Virol. **3**(4), 285–297 (2007)
2. Borello, J.M., Filiol, E., Me, L.: From the design of a generic metamorphic engine to a black-box classi cation of antivirus detection techniques. J. Comput. Virol. **6**(3), 277–287 (2010)
3. Borello, J.M., Me, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**(3), 211–220 (2008)
4. Borello, J.M., Me, L., Filiol, E.: Dynamic malware detection by similarity measures between behavioral profiles. In: Proceedings of the 2011 Conference on Network and Information Systems Security, IEEE (2011)
5. Bruschi, D., Martignoni, L., Monga, M.: Using code normalization for fighting self-mutating malware. In: Proceedings of International Symposium on Secure Software Engineering Washington, DC (2006)
6. Carrera, E., Erdelyi G.: Digital genome mapping-advanced binary malware analysis. In: Proceeding of the 2004 Virus Bulletin Conference, pp. 187–197 (2004)
7. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 5–14. ACM, New York (2007)
8. Fredrikson, M., Jha, S, Christodorescu, M., Sailer, R., Yan, X.F.: Synthesizing near-optimal malware specifications from suspicious behaviors. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 45–60. IEEE (2010)
9. Gao, X.B., Xiao, B., Tao, D.C.: A survey of graph edit distance. Pattern Anal. Appl. **13**(1), 113–129 (2010)
10. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. WH Freeman & Co, New York (1979)
11. Gheorghescu, M.: An automated virus classification system. In: Proceedings of the Virus Bulletin Conference, pp. 294–300 (2005)
12. Hex-Rays, S.A.: IDA Pro 5.5, http://www.hex-rays.com/products/ida/index.shtml (2010)
13. Hu, X.: Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 611–620. ACM, New York (2009)
14. Jeong, K., Lee, H.: Code graph for malware detection. In: Proceedings of the International Conference on Information Networking, IEEE, pp. 1–5 (2008)
15. Kapoor, A., Spurlock J.: Binary feature extraction and comparison. In: Proceedings of the AVAR 2006, Auckland (2006)
16. Karnik, A., Goswami, S., Guha, R.: Detecting obfuscated viruses using cosine similarity analysis. In: Proceedings of the First Asia International Conference on Modelling & Simulation (AMS'07), pp. 165170. IEEE Computer Society, Phuket (2007)
17. Kaspersky.: Monthly malware statistics: May 2009, http://www.kaspersky.com/news?id=207575832 (2010)
18. Kinable, J., Kostakis, O.: Malware classification based on call graph clustering. J. Comput. Virol. **7**(4), 233–245 (2011)
19. Kostakis, O.: Improved call graph comparison using simulated annealing. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 1516–1523. ACM, New York (2011)

20. Kruegel, C., Kirda, E.: Polymorphic worm detection using structural information of executable. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), pp. 207–226 (2005)
21. Lakhotia, A., Kumar, E.U., Venable, M.: A method for detecting obfuscated calls in malicious binaries. IEEE Trans. Softw. Eng. **31**(27), 955–967 (2005)
22. Lee, J., Jeong, K., Lee H.: Detecting metamorphic malwares using code graphs. In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 1970–197. ACM, New York (2010)
23. Li, J., Xu M., Zheng N., Xu. : Malware obfuscation detection via maximal patterns. In: Proceedings of the Third International Symposium on Intelligent Information Technology Application, IEEE. pp. 324–328 (2009)
24. PEiD 0.95, http://www.peid.info/ (2010)
25. Scanspyware.http://spyware.scanspyware.net/spywareremoval/rootkit.kernelbot.html (2012)
26. Securelist.http://www.securelist.com/en/descriptions/old79396 (2012)
27. Shang, S. H., Zhen, N., Xu, J., Xu, M., Zhang, H. P.: Detecting malware variants via function-call graph similarity. In: Proceedings of the 5th Malicious and Unwanted Software, IEEE, pp. 113–120 (2010)
28. Symantec.: Internet Security Threat Report, Volume 17. Technical report, Symantec Corporation. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf (2011)
29. Szor, P.: The Art of Computer: Virus Research and Defense, 1st edn. Symantec Press, NJ (2005)
30. Tabish, S.M., Shaq M.Z., Farooq M.: Limits of static analysis for malware detection. In: Proceedings of the ACSAC, IEEE Computer Society, pp. 421430 (2007)
31. Tian, R., Batten, L.M., Versteeg, S.C.: Function length as a tool for malware classification. In: Proceedings of the 3rd Malicious and Unwanted Software (MALWARE), pp. 69–76 (2008)
32. UPX 3.05, http://upx.sourceforge.net/ (2010)
33. Viruslistjp.http://www.viruslistjp.com/viruses/encyclopedia/?virusid=20425 (2002)
34. VX Heavens. http://vx.netlux.org/index.html (2010)
35. Zhang, Q., Reeves. D.S.: MetaAware: identifying metamorphic malware. In: Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07), pp. 411–420 (2007)