Detecting Malware Variants via Function-call Graph Similarity

Shanhu Shang Ning Zheng Jian Xu Ming Xu Haiping Zhang
Institute of Computer Science, Hangzhou Dianzi University, P.R. China
shanhu.shang@gmail.com {nzheng, jian.xu, mxu, zhanghp}@hdu.edu.cn

Abstract

Currently, signature-based malware scanning is still the dominant approach to identify malware samples in the wild due to its low false positive rate. However, this approach concentrates on programs' specific instructions, and lacks insight into high level semantics; it is enduring challenges from advanced code obfuscation techniques such as polymorphism and metamorphism. To overcome this shortcoming, this paper extracts a program's function-call graph as its signature. The paper presents a method to compute similarity between two binaries on basis of their function-call graph similarity. The proposed method relies on static analysis of a program, it first disassembles the program into assemble code, and then it uses a novel algorithm to construct the function-call graph from the assembly instructions. After that, it proposes a simple but effective graph matching method to compute similarity between two binaries. A prototype is implemented and evaluated on several well-known malware families and benign programs.

1. Introduction

Malware, short for malicious software, is software designed to infiltrate a computer system without the owner's informed consent. In the last few years, driven by profit, the spread of malware skyrockets year by year, furthermore, code reuse and online malware construction kits worsen the situation. Statistics from PandaLabs 2009 annual report [1] confirmed this fact, the total number of individual malware samples in Panda's database reached 40 million in 2009, and its research laboratory received about 55,000 daily samples. In consequence, total losses attributed to malicious programs are estimated at billions of dollars each year. Therefore, effective countermeasures must be taken to mitigate the damages caused by malware.

For a long time, antivirus products that adopt signature matching approach perform well when facing known viruses. However, there are many shortages tied to this approach. Firstly, the identification of a malware signature is both labor-consuming and time-consuming despite automatic signature extraction methods have been proposed [2]. Secondly, this syntactic-based detection method can be easily bypassed by simple code obfuscation [3] because it ignores program's functionality. As a result, these shortcomings are exploited by malware authors, and more powerful viruses like polymorphic, metamorphic viruses [4] [5] are developed to challenge traditional antivirus tools. In this recent virus-antivirus battle, code emulation combining signature scanning method is used to detect the polymorphic malware by antivirus vendors. This method is effective to most polymorphic viruses, except some cunning ones which own self-protection capabilities like anti-debug, anti-emulation. Unfortunately, a metamorphic virus is much more difficult to defeat, since it can rewrite its own code with each infection, while keeping the original functionality.

To overcome the shortages of traditional malware signature, this paper treats the function-call graph as a binary's signature. The function-call graph of a binary is a directed graph consisting of a set of vertices and a set of directed edges. Given a program, each vertex in its function-call graph corresponds to a unique subroutine in the program's source code, and each edge depicts the caller-callee relationship between two functions. Function-call graph is chose as malware signature because it represents the functionality and objective of a program semantically. No matter what obfuscating transformations are performed on the program, its functionality should be kept the same. Therefore, function-call graph is more resilient to code obfuscation techniques than signature strings.

The main contributions of this paper are as follows:

- We present a novel approach to construct a binary's function-call graph from its assembly code.

978-1-4244-9356-2/10/$26.00 ©2010 IEEE

- We put forward an algorithm to compute similarity between two binaries on basis of their function-call graphs' similarity.
- We implement a prototype system and apply it to several malware families and benign benchmark programs.

The rest of this paper is organized as follows. Section 2 briefly reviews related work. Section 3 describes the proposed method specifically. Section 4 presents our prototype, and results on applying it to several malware and benign utilities. Section 5 points out the limitations of current work and future work focus. Section 6 concludes the paper.

2. Related Works

As described in last section, code obfuscation techniques, especially polymorphism and metamorphism, posed enormous threat to antivirus detectors. To reverse the tide, many avenues have been proposed by antivirus researchers and vendors.

Christodorescu and Jha [6] firstly used the static analysis method to tackle malware code obfuscation; they utilized the program's CFG (control flow graph) to defeat simple code obfuscations, and a static analyzer for executables (SAFE) was also implemented. The same authors [7] later described a formal approach to metamorphic virus detection using a template-matching method; a template consists of a set of instruction sequences, which represents certain malicious behavior.

Another ideology to counteract the effects of self-mutation malware is to normalize the mutations to their canonical form. Walenstein et al. [8] presented an approach to construct a normalizer for a particular class of mutating malware by leveraging term rewriting theory. A similar idea had also been put forward by Bruschi et al. [9] by utilizing compiler techniques.

Programs' structural information, like CFG, function-call graph, was mined to detect malware variants too. Paper [10] proposed a method that reduces the problem of detecting malware inside an executable to a sub-graph isomorphic problem. Paper [11] presented an approach for recognizing metamorphic malware based on the pattern of library or system functions that were called. Function-call graph was used to aid the malware analysis in [12], and the method was proved to be helpful in finding similarities and differences among various malware variants and strains. Paper [13] presented an automatic technique to derive malware specification by monitoring the system calls that malware invoked. Paper [14] designed, implemented, and evaluated a malware database management system to process huge malware samples based on malware's function-call graphs, many techniques were adopted to speed up the graph similarity calculation process and indexing speed.

Our method assimilates many good ideas from papers listed above especially [12] and [14], and it makes great improvements. Previous papers that used CFG, function-call graph as malware signature often lead to expensive time and space expenditure, because these graph matching methods, such as graph edit distance, are too time-consuming (bipartite matching in time $O(n^3)$ by using of Hungarian algorithm) and space-consuming. Our method adopts a different way to compute the graph similarity on basis of graphs' common vertices. Various techniques are employed to accelerate the graph matching process.

3. The Proposed Method

In this section we will present a method to extract the function-call graph from an executable and an algorithm to compute the graph similarity. At a high level, the idea works mainly in three steps:

1) Disassemble the binary;
2) Construct a function-call graph from the assembly code;
3) Compare the similarity between two programs via calculating their function-call graphs' similarity.

Given a binary program, firstly, we use PEiD [15] to check whether it is packed and what kind of packing tool is exploited, if this is identified, several unpackers are used to unpack the file accordingly, e.g. UPX [16] and like. Secondly, we utilize the well-known interactive disassembler IDA Pro [17] to disassemble the sample, and the output of this process are assembly code of the binary and a set of identified functions. Thirdly, we traverse the assembly code to construct a function-call graph. Finally, on basis of the function-call graph, we propose a graph similarity metric to measure how similar two binaries are.

3.1. Function-Call Graph Definition

The function-call graph of a binary is a directed graph, which consists of a set of vertices and a set of directed edges, Figure 1 shows part of the function-call graph from Email-Worm.Win32.Mydoom.g. In this paper, for the convenience of computing similarity

between two graphs, we also add a labeling function μ to label the vertices and a weighting function ϖ to calculate the edges' weights and the vertices' degrees. A formal definition of a program's function-call graph is given by Definition 1.

Definition 1. A function-call graph is a 4-tuple $G = (V, E, \mu, \varpi)$, where

V is a set of finite vertices. Each vertex corresponding to a unique function in a program, so a vertex or a function represents the same thing in this paper later. A vertex contains several attributes, and Figure 2 depicts a vertex and its main attributes.

$E \subseteq V \times V$ is a set of directed edges. $\forall u, v \in V$, $\exists < u, v > \in E. < u, v >$ stands for there exists a directed edge from vertex u to vertex v, it also implies that function u contains a function call to function v, and vice versa.

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.

ϖ is a weighting function to compute the edges' weights and the vertices' indegrees and outdegrees. $\forall e \in E, \exists \varpi(e) \in N; \forall u, v \in V, < u, v > \in E, \exists \varpi(u) \in N$, $\varpi(v) \in N$, N is the set of positive integers.

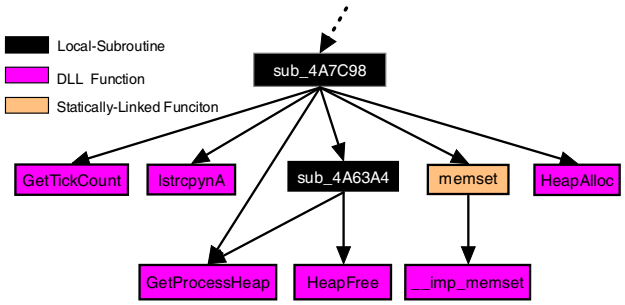

Figure 1. Part of function-call graph from Email-Worm.Win32.Mydoom.g

More specifically, μ is used to label the functions of a program into an attribute set Lv. Each label in Lv is comprised of five elements: function name, function type, a pointer points to the function's first callee, a pointer points to the function's first caller and the function's opcode sequence.

Functions are classified into three categories: Dynamically-Imported functions, Statically-Linked library functions and the Local-Subroutines. Dynamically-Imported functions are DLL functions that are linked at load time or run time of a binary, only the function names are imported into the IAT (Import Address Table) [18] of the target application, for instance, "LoadLibrary", "GetProcAddress" from Kernel32.dll; Statically-Linked library functions are resolved in a caller at compile-time, and the function bodies are copied into a target application by the linker to produce a stand-alone executable, e.g., "memset" from the C library; Local-Subroutines are functions coded elaborately by the malware authors to implement specific feature, e.g., functions that are designed to relocate the virus code or to get address of a certain API.

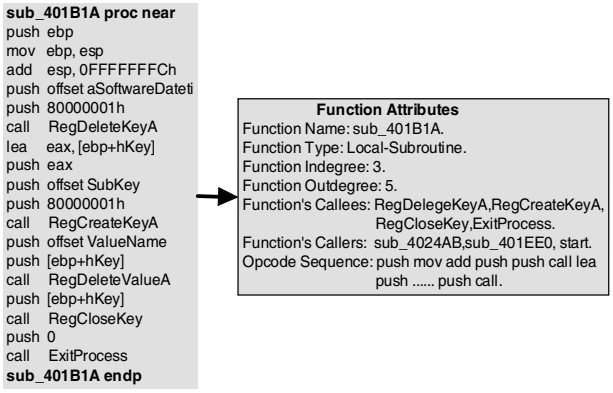

Figure 2. A function from Worm.Win32.Bagle.i

ϖ is used to count a vertex's indegree, outdegree and a directed edge's weight. The function's outdegree denotes how many functions it calls and its indegree represents how many times it is called by others. A function may call or be called by many functions, it is also possible that one function is called more than once by the same caller, so the edge's weight represents how many times a callee is invoked by the same caller in the program. If a function's indegree is greater than zero, the function maintains a list consisting of incoming edges; similarly, if its outdegree is greater than zero, it also has a list consisting of outgoing edges.

To facilitate describing the proposed algorithm more clearly, we give some relevant definitions.

Definition 2. The callee set of vertex u of graph $G = (V, E, \mu, \varpi)$ is a vertex set $Callees(u) = \{v | < u, v > \in E\}$ it represents all the functions called directly by u, and $u.outdegree = | Callees(u) |$.

Definition 3. The caller set of vertex u of graph $G = (V, E, \mu, \varpi)$ is a vertex set $Callers(u) = \{v | < v, u > \in E\}$, it represents all the functions that invoke u directly, and $u.indegree = | Callers(u) |$.

Definition 4. The neighbors of vertex u of graph $G = (V, E, \mu, \varpi)$ is a vertex set $Neibors(u) = Callees(u) \cup Callers(u)$, so $| Neibors(u) | = u.indegree + u.outdegree$.

3.2. Function-Call Graph Extraction

We now describe in detail how to build the function-call graph from a binary. The input to the algorithm is a program's assembly code and the output is a function-call graph, pseudo-code is given in Algorithm 1.

Algorithm 1
Input: An assembly file *M*.
Output: Function-call graph G_M.
Begin
/*Init the graph G_M, Funset, FunQueue.*/
1 G_M.V = ∅ ; G_M.E = ∅ ;
2 EntryFunSet = ∅ ; FunSet = ∅ ;
3 FunQueue = ∅ ; HeadVexSet = ∅ ;
/*Extract functions from assembly code.*/
4 FunSet ← *ExtractFun*(*M*);
5 EntryFunSet ← *ExtractEntryFun*(*M*);
6 FunQueue ← *InitFunQueue*(EntryFunSet);
/*Build the caller-callee relationship.*/
7 while(∅ ≠ FunQueue)
8 tailVertex = *DeQueue*(FunQueue);
InsertVertex(G_M, μ(tailVertex));
tailVertex.*enQueFlag* = true;
/*Extract tailVertex's callee set.*/
11 HeadVexSet{h_1,…,h_n}←*Callees*(tailVertex);
12 foreach i=1 to n do
if(∅ ≡ (h_i ∩ FunSet))// *hi is not in FunSet.*
continue;
endif
16 headVertex = h_i;
e = <μ(tailVertex), μ(headVertex) >;
18 if(*e* ∈ G_M.*E*)
ϖ(*e*++);
ϖ(tailVertex.*outdegree*++);
ϖ(headVertex.*indegree*++);
22 else
InsertVertex(G_M, μ(headVertex));
InsertEdge(G_M, *e*);
25 endif
26 if(false == headVertex.*enQueFlag*)
EnQueue(FunQueue, headVertex);
headVertex.*enQueFlag* = true;
29 endif
30 endwhile
31 return G_M;
End

Once a binary is disassembled successfully by IDA Pro, the assembly code is obtained, and simultaneously the functions' boundaries are identified. IDA Pro names each identified function with a symbolic name. For Dynamically-Imported functions, the function names can be gained from the IAT in the binary's PE header [18]. For Statically-linked library functions, their names can be recognized by IDA Pro's FLIRT (Fast Library Identification and Recognition Technology) [19]. Nevertheless, the Local-Subroutines' names are unavailable if the symbol table of the binary has been stripped off by the malware author. Therefore, this kind of functions is named with the same prefix "sub_", followed with the function's memory address.

The algorithm adopts a breadth-first approach to build the function-call graph by use of a FIFO function queue, and the function-call graph is stored into an orthogonal list. The algorithm builds the caller-callee relationship starting from the entry point functions. It traverses each function's instruction sequences to find all the subroutines called by the function, when all the functions are processed, the algorithm ends and the function-call graph is constructed.

First of all, the algorithm traverses the assembly code to recognize every function's boundary and save the functions into a function set called "FunSet". At the same time it extracts all the entry point functions and stores them into the "EntryFunSet". Then the function queue is initialized with the entry point functions. While the queue is not empty, the algorithm dequeues the head element ("tailVertex" in line 8) of the queue, and treats it as a function caller. Afterwards the labeled "tailVertex" is inserted into the graph G_M, and its "enQueFlag" is set to be true in case the same vertex enqueues the queue repeatedly later. After that, the algorithm traverses the instruction sequence of the "tailVertex" to extract its callee set, corresponding to line 11 in Algorithm 1. For the present, we only cope with the situation that function call instructions are identified explicitly, such as "call lstrlenA", "jmp sub_403252".

When the callees are acquired, the algorithm traverses the callee set to check whether the graph already has an edge from the "tailVertex" to the "headVertex", if so, the algorithm increase the caller's outdegree, the callee's indegree and the edge's weight by one separately (line 18-21); otherwise, it inserts the "headVertex" and the new edge into the graph. At last, the algorithm checks whether the "headVertex" has been enqueued, if not, the "enQueFlag" of the "headVertex" will be marked as true and appended to the tail of the queue. This caller-callee relationship building process maps with line 7 to line 30. When the while loop ends, the function-call graph is constructed smoothly. The time complexity of this algorithm is $O(|V|*|E|)$ and the space complexity is $O(2*|V|+|V|*|E|)$.

3.3. Function-Call Graph Similarity

The past section depicted how to construct a function-call graph from a binary, and in this section

we will present an algorithm to measure the similarity between two function-call graphs G_1 and G_2.

Semantically, the common functions shared by two binaries signify that they are likely to accomplish the same or similar tasks. Grounded on this assumption, the algorithm tries to acquire the maximum common vertices between G_1 and G_2. Once the joint vertices of the two graphs are gained, the similarity score between them can be represented by the proportion of common vertices number to the maximum vertex number. Although this is not a symmetric similarity metric, it still can reveal to what extent two graphs resemble each other.

It is worth noting that the algorithm does not simply compare the vertices between two graphs, it also takes the edges' information into consideration during the process of vertex mapping. Ahead of illustrating the algorithm specifically, a formal definition of maximum common vertex-pair set is given by Definition 5 and the graph similarity metric is given by Definition 6.

Definition 5. Let $G_1=(V_1,E_1,\mu,\varpi)$, $G_2=(V_2,E_2,\mu,\varpi)$ be two function-call graphs, and $Mcv(G_1,G_2)=\{(u,v)\mid u\in V_1\bigcap v\in V_2\bigcap M(u,v)\geq\theta\}$ be matched vertex-pair set. Actually, $Mcv(G_1,G_2)$ consists of matched vertices generated by a vertex mapping function M, and θ denotes a low similarity threshold between two vertices from G_1 and G_2. When $Mcv(G_1,G_2)$ is obtained, the similarity value is calculated by Definition 6.

Definition 6. The similarity metric between two function-call graph $G_1=(V_1,E_1,\mu,\varpi)$, $G_2=(V_2,E_2,\mu,\varpi)$ is defined as $Sim(G_1,G_2)=\dfrac{|Mcv(G_1,G_2)|}{Max(|V_1|,|V_2|)}$, where $Mcv(G_1,G_2)$ is the number of matched vertex-pairs, and $Max(|V_1|,|V_2|)$ represents the maximum vertex number of graph G_1 and G_2. For any graphs G_1 and G_2, $Sim(G_1,G_2)$ always yields a value in [0, 1], value 0 indicates a rather low degree of resemblance, and value 1 shows they share almost the same vertices.

Let Uv_1, Uv_2 denote the vertices to-be-matched in vertex set V_1 and V_2 respectively, pseudo-code for the graph similarity computation is given in Algorithm 2. The Algorithm is divided into five phases and the vertex matching process takes up the former four.

In the first step, the algorithm begins matching the vertices from the atomic functions [12]. In this paper, the atomic functions refer to Dynamically-Imported and Statically-Linked library functions that share the same names across distinct executables. If two atomic functions have the same name, they are regarded as a match and saved into $Mcv(G_1,G_2)$. When this process is completed, if $Mcv(G_1,G_2)$ is still empty, which means the two programs have no common atomic functions, the vertex mapping algorithm switches to phase 4; Otherwise, it goes to step 2 to maximize the vertex-pairs greedily.

In step 2, the algorithm searches unmatched vertices in V_1 and V_2 to lookup if there are two vertices fulfill the following condition: All the atomic functions called by the two vertices are the same; besides, the number of common atomic functions is greater than 1. If two vertices satisfy the condition, the algorithm treats them as a successful match and adds the vertex-pair into $Mcv(G_1,G_2)$. This method is a little similar to the "call-tree signatures" approach in [12], but it takes more tolerance to code reordering.

The previous step introduces a vertex matching method based on the trust from matched atomic functions. In the third stage, the algorithm takes a neighbor-biased approach to expand the matching results. In a function-call graph, the neighbors of a vertex v are functions that are called by v together with those invoke v. If a majority of two vertices' neighbors have been matched, they are mostly possible to be the same. On account of this, to two vertices not yet matched, if the proportion of their matched neighbors to their maximum neighbor number exceeds a user-defined threshold σ (the experiment results show that 0.7 is a suitable value for σ), the algorithm takes the two vertices as a match and stores them into $Mcv(G_1,G_2)$.

During step 4, to further decrease the unmatched vertices in V_1 and V_2, the algorithm exploits the vertices' instruction-level information. Firstly it extracts the effective opcodes (junk opcodes like "nop" are removed) from instruction sequences and aligns them into an opcode line (see Figure 2). The algorithm then computes the similarity between the opcode sequences of two vertices by use of the LCS (Longest Common Subsequence) algorithm [20]. On condition that the similarity score outnumbers a user-defined threshold τ (τ = 0.75 is chosen empirically), two vertices are considered as a match and kept into $Mcv(G_1,G_2)$. This process repeats until no more matches are yielded. Opcode sequence is chose to calculate the similarity between vertices because it is more resilient to code obfuscation than the whole instruction sequence.

Algorithm 2
Input: Function-call graph G_1, G_2.
Output: $Sim(G_1, G_2)$.
Begin
$Uv_1 \leftarrow G_1.V_1$, $Uv_2 \leftarrow G_2.V_2$;
Step 1 /*Vertex matching based on atomic functions.*/
AtomicFunSet1 $\leftarrow$ $ExtractAtomicFun(G_1.V_1)$;
AtomicFunSet2 $\leftarrow$ $ExtractAtomicFun(G_2.V_2)$;
$Mcv(G_1, G_2) = \{(u, v) | u \in$ AtomicFunSet1 $\cap$ $v \in$ AtomicFunSet2 $\cap$ $u.vexName == v.vexName\}$;
$Uv_1 = G_1.V_1 - \{u | u \in V_1 \cap (true == u.matchFlag)\}$;
$Uv_2 = G_2.V_2 - \{v | v \in V_2 \cap (true == v.matchFlag)\}$;
if($\varnothing \equiv Mcv(G_1, G_2)$)
goto Step 4;
endif
Step 2 /*Vertex matching based on atomic functions.*/
foreach $u_i \in Uv_1$ do
foreach $v_j \in Uv_2$ do
if(AtomicCallees(u_i) $\equiv$ AtomicCallees(v_j))
$Mcv(G_1, G_2) = Mcv(G_1, G_2) \cup (u_i, v_j)$;
$Uv_1 = Uv_1 - u_i$; $Uv_2 = Uv_2 - v_j$;
break;
endif
Step 3 /*Vertex matching based on matched neighbors.*/
foreach $u_i \in Uv_1$ do
foreach $v_j \in Uv_2$ do
NeiborPairs = $| Neibors(u_i) \cap Neibors(v_j) |$;
if((NeiborPairs / $max((|Neibors(u_i)|, |Neibors(v_j)|)) \geq \sigma$)
$Mcv(G_1, G_2) = Mcv(G_1, G_2) \cup (u_i, v_j)$;
$Uv_1 = Uv_1 - u_i$; $Uv_2 = Uv_2 - v_j$;
break;
endif
Step 4 /*Vertex matching based on opcode sequences.*/
foreach $u_i \in Uv_1$ do
foreach $v_j \in Uv_2$ do
vexSim = $LCS(u_i.opcodeSeq, v_j.opcodeSeq)$;
if(vexSim $\geq \tau$)
$Mcv(G_1, G_2) = Mcv(G_1, G_2) \cup (u_i, v_j)$;
$Uv_1 = Uv_1 - u_i$; $Uv_2 = Uv_2 - v_j$;
break;
endif
Step 5/*Graph similarity calculation.*/
$Sim(G_1, G_2) = |Mcv(G_1, G_2)| / Max(|V_1|, |V_2|)$;
return $Sim(G_1, G_2)$;
End

At the end of the vertex matching process, we obtain the matched vertex-pair set $Mcv(G_1, G_2)$, so the similarity score can be calculated by Definition 6 (step 5). The worst time complexity is $O((|V_1|*|V_2|)*(\overline{m}*\overline{n}))$, where $\overline{m}$ and $\overline{n}$ represent the average neighbor number of each vertex in V_1, V_2 separately, this complexity is resulted by some vertex pairs in the third phase of the algorithm. However, not all of the vertices need to be matched in such a way. In contrast, the best time complexity is $O(|V_1|*|V_2|)$ if the two graphs only consist of atomic functions.

4. Experimental Evaluation

For the purpose of verifying the correctness and efficacy of our ideas, we implement a prototype and apply it to a set of known malware samples and benign files. The evaluation can be divided into two parts. In the first part, we try to evaluate the algorithms' ability to identify malware variants compared to [12]; in the second part, we attempt to validate the algorithms' capability to distinguish benign binaries and malicious programs.

4.1. Variants Similarity Evaluation

The dataset used in the first experiment consists of several malware families selected from VX Heavens [21], including Email-Worm.Win32.Mimail, Email-Worm.Win32.Klez, Virus.Win32.Sality, Virus.Win32.Evol.

Email-Worm.Win32.Mimail

The Mimail family members are potent worms that spread in the form of a file attachment sent by email. The following similarity matrix show the similarity scores between each pair of the malicious binaries in percentage terms. The values 100.00 along the main diagonal are similarity scores that the variants compare with themselves. Data that lie above the main diagonal are experimental results from [12] and below are ours. Unfortunately, symbol '*' represents the similarity score is absent because the sample is unavailable in VX heaven; while sign '-' means no data is given in the contrast experiment.

	Mimail.a	Mimail.b	Mimail.c	Mimail.d	Mimail.e	Mimail.f	Mimail.g
Mimail.a	100.00	90.80	85.40	87.40	75.00	75.00	-
Mimail.b	*	100.00	84.70	88.00	74.30	74.30	-
Mimail.c	95.14	*	100.00	81.50	81.30	81.30	-
Mimail.d	*	*	*	100.00	72.30	72.30	-
Mimail.e	86.86	*	86.11	*	100.00	95.40	-
Mimail.f	86.86	*	86.11	*	100.00	100.00	-
Mimail.g	86.86	*	86.11	*	100.00	100.00	100.00

	Mimail.h	Mimail.i	Mimail.j	Mimail.k	Mimail.l	Mimail.m	Mimail.q
Mimail.h	100.00	81.70	83.20	-	90.90	88.80	41.60
Mimail.i	92.74	100.00	95.00	-	81.70	79.90	46.90
Mimail.j	92.74	100.00	100.00	-	83.20	81.30	47.80
Mimail.k	100.00	92.74	92.74	100.00	-	-	-
Mimail.l	97.58	92.74	92.74	97.58	100.00	90.30	42.40
Mimail.m	97.58	94.26	94.26	97.58	97.58	100.00	40.60
Mimail.q	27.42	28.81	28.81	27.42	26.61	27.05	100.00

The higher similarity scores demonstrate that our method outperforms [12] except the results given by Mimail.q. Compared with other variants, besides the worm itself, Mimail.q still has a dropper, that's why it seems more different from others. Becides, we researched the unpacking and vertex matching process, and found out that function-call graph extracted from Mimail.q has much less vertices than other variants, this maybe attributes to incomplete unpacking.

Email-Worm.Win32.Klez

Compared to [12], the Email-Worm.Win32.Klez family also gives better similarity results.

	Klez.a	Klez.b	Klez.c	Klez.d	Klez.e	Klez.f	Klez.g	Klez.h	Klez.i	Klez.j
Klez.a	100.00	79.60	70.30	70.30	49.30	–	–	–	–	–
Klez.b	95.98	100.00	73.40	73.40	49.30	–	–	–	–	–
Klez.c	100.00	95.98	100.00	88.20	49.60	–	–	–	–	–
Klez.d	86.94	91.04	86.94	100.00	49.60	–	–	–	–	–
Klez.e	*	*	*	*	100.00	–	–	–	–	–
Klez.f	61.47	63.91	61.47	67.28	*	100.00	87.00	80.70	80.70	87.00
Klez.g	61.47	63.91	61.47	67.28	*	100.00	100.00	80.70	80.70	87.00
Klez.h	61.56	63.75	61.56	65.63	*	94.80	94.80	100.00	89.60	80.70
Klez.i	61.56	63.75	61.56	65.63	*	94.80	94.80	100.00	100.00	80.70
Klez.j	61.47	63.91	61.47	67.28	*	100.00	100.00	94.80	94.80	100.00

Because the worms listed above in the contrast experiment were more than five years ago, in order to validate our algorithms' capability to detect up-to-date viruses, Virus.Win32.Sality family is selected as a typical case.

Virus.Win32.Sality

Virus.Win32.Sality is a family of polymorphic viruses that target Windows executable files with extensions .SCR or .EXE. From the second half of 2008 to the first half of 2009, Variants of the Sality family have long been ranking the top 20 in the Monthly-Malware-Statistics [22] published by Kaspersky Security Network. Similarity scores between mutations of the Sality family are listed below in a percent form, and the matrix is symmetric. From the matrix we can see, all the similarity scores are greater than 0.4.

	Sality.a	Sality.c	Sality.d	Sality.e	Sality.f	Sality.g
Sality.a	100.00	99.67	96.01	62.79	40.86	87.04
Sality.c	99.67	100.00	95.70	62.58	40.73	86.76
Sality.d	96.01	95.70	100.00	64.51	41.98	88.40
Sality.e	62.79	62.58	64.51	100.00	58.05	68.25
Sality.f	40.86	40.73	41.98	58.05	100.00	43.80
Sality.g	87.04	86.76	88.40	68.25	43.80	100.00

Virus.Win32.Evol

Virus.Win32.Evol family is studied to further testify the resilience of the proposed algorithm to self-mutating viruses. According to the malware description from [5], "W32.Evol is the first W32 virus using a 32-bit true metamorphic engine. It can replicate on Windows 9x as well as Windows NT and Windows 2000". The similarity values among all the three members Evol.a, Evol.b and Evol.c of the family are 100%.

4.2. Hybrid Similarity Evaluation

It is also very important to measure the similarities between malware and benign binaries, which disclose whether our method can differentiate good from evil. The data set used in this test consists of 11 legitimate binaries and 11 malicious programs. The benign binaries are collected from the system files and well-known application software; the malicious programs contain various variants from different virus families, all of them are downloaded from VX heaven, the test suite is listed in Table 1.

Table 1. The benign and malicious samples

Program Name	Description
ipv6.exe	system file from system32 folder, Win XP SP2.
lsass.exe	system file from system32 folder, Win XP SP2.
netstat.exe	system file from system32 folder, Win XP SP2.
cdm.dll	system file from system32 folder, Win XP SP2.
pid.dll	system file from system32 folder, Win XP SP2.
md5sum.exe	md5 checksum [23].
putty0.60	a free telnet/SSH Client, version 0.60.
Firefox3.6.3	Firefox browser version 3.6.3.
install_icq7.exe	a popular instant chatting program, version 7.
IE7-WinXP-x86-chs.exe	Internet Explorer 7 for XP SP2 users in China.
IE8-WinXP-x86-chs.exe	Internet Explorer 8 for XP SP2 users in China.
Trojan.Win32.AVKill.a	a hacker tool with subversive purposes.
Trojan.Win32.ICQPager.b	a password capture.
Worm.Win32.Bagle.i	a worm spreads via e-mail and file sharing.
Virus.Win32.Evol.a	a 32-bit metamorphic virus.
Worm.Win32.Kido.ih	also known as Conficker.
Worm.Win32.Kido.dam.x	a variant of the Kido family.
Worm.Win32.Mimail.c	a potent worm spreads via email attachment .
Virus.Win32.Sality.d	a polymorphic virus.
Virus.Win32.Sality.e	a variant of the Sality family.
Virus.Win9x.ZMorph.5200	an early polymorphic virus.
Virus.Win32.Zmist	a metamorphic malware written by Zombie.

Similarities between any two of these 22 binaries are calculated by the proposed method, producing 231 similarity value pairs. Experimental results are pictured in Figure 3, x-axis, y-axis indicate the 22 binaries respectively and z-axis represents the similarity percentages among the samples.

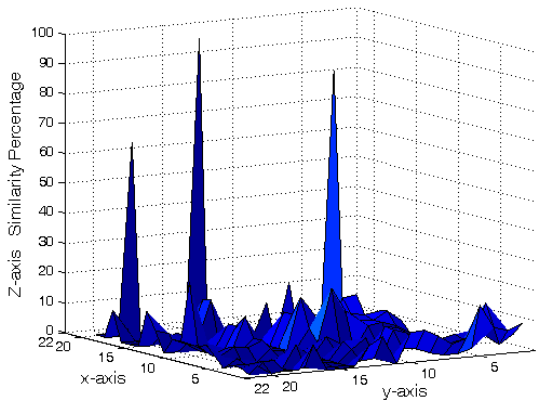

Figure 3. Similarities among hybrid binaries

As we can see from the graph, there are 3 vertical lines whose similarity scores are much higher than the others. Actually, they are generated by 3 program pairs: IE7 and IE8 (similarity value 0.9091), Sality.d and Sality.e (similarity value 0.6451), Kido.h and Kido.dam.x (similarity value 1). Besides the 3 pairs, for the great majority cases, the similarity scores are below 0.25 except the score 0.2579 generated by ICQ7 and Zmist. We investigated the matched vertices of the two binaries, and found that they shared 48 atomic functions. This was why they produced a higher similarity score than others.

5. Limitations and Future Work

Although preliminary experimental results are very inspiring, it still has many drawbacks to fully realize malware automatic analysis.

As our method relies on the static analysis results, so the main limitation comes from static analysis. Many code obfuscation techniques, like code packing, entry point obscuring, have been exploited to hinder or invalidate the disassembly process. The experiment process proved this fact as well; a few instances downloaded from VX Heavens were packed and difficult to unpack. Hence dynamic analysis is needed to help disassemble the malware inerrably.

Implicit function-call is a severe threat to current prototype too. A few malware invoke the atomic functions implicitly, which may cause the current method invalid, so that future work will focus on resolving this shortage.

6. Conclusion

This paper presented a method to compute the similarity between two binaries on basis of their function-call graphs' similarity.

To construct a binary's function-call graph, this paper introduced a novel approach by use of a FIFO function queue, and the graph was stored into a well-designed orthogonal list. Once the function-call graph was extracted, the paper computed the similarity between two graphs using an asymmetric similarity metric. In the process of vertex matching, information from multiple aspects was mined to maximize the common vertex pairs.

In the end, the proposed method was prototyped and evaluated by wild malware, system files and well-known applications. Empirical results demonstrated the efficacy and resilience of the proposed method in identifying malware variants. Besides, the method could also be used in other areas of malware analysis, such as phylogeny construction.

Acknowledgement

This paper is supported by NSFC (No. 61070212, No.61003195), Natural Science Foundation of Zhejiang Province, China under Grant No. Y1090114.

References

[1] Annual Report PandaLabs 2009, http://www.pandasecurity.com/homeusers/security-info/tools/reports/.

[2] K. Griffin, S. Schneider, X. Hu and T.-c. Chiueh. Automatic generation of string signatures for malware detection. In Proceedings of RAID 2009, pages 101-120, Brittany, France, Sep. 2009.

[3] M. Christodorescu and S. Jha. Testing malware detectors. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2004 (ISSTA'04), pages 34–44, Boston, MA, USA, ACM Press, July 2004.

[4] J. Marie, Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. Journal in Computer Virology, Volume 4, pages 211-220, 2008.

[5] P. Szor. The art of computer virus research and defense. Symantec Press, USA, first edition, 2005.

[6] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In Proceedings of the 12th conference on USENIX Security Symposium, Volume 12, Washington, DC, 2003.

[7] M. Christodorescu, S. Jha, et al. Semantics-aware malware detection. In IEEE Symposium on Security and Privacy, May 2005.

[8] Walenstein, R. Mathur, et al. Constructing malware normalizers using term rewriting. Journal in Computer Virology, Volume 4, pages 307-322, 2008.

[9] D. Bruschi, L. Martignoni, M. Monga. Code normalization for self-mutating Malware. Journal in IEEE Security & Privacy, Volume 5, pages 46-54, 2007.

[10] D. Bruschi, L. Martignoni, M. Monga. Detecting self-mutating malware using control-flow graph matching. In Detection of Intrusions and Malware & Vulnerability Assessment, Volume 4064, pages 129-143, Nov. 2006.

[11] Z. Qinghua and D. S. MetaAware: Identifying metamorphic malware. In Proceedings of the 23th ACSAC on Computer Security Applications Conference, pages 411-420, Dec. 2007.

[12] E. Carrera and G. Erdelyi. Digital genome mapping - Advanced binary malware analysis. In Proceedings of the 2004 Virus Bulletin Conference, 2004.

[13] M. Christodorescu, S. Jha and C. Kruegel. Mining specifications of malicious behavior. In Proceedings of the ACM SIGSOFT symposium on the foundations of software engineering, pages 5-14, Dubrovnik, Croatia, 2007.

[14] X. Hu, T.-c. Chiueh and K. G. Shin. Large-scale malware indexing using function-call graphs. In Proceedings of CCS 2009, pages 611-620, Chicago, Illinois, USA.

[15] PeiD 0.95, http://www.peid.info/, 2010.

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

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

[18]Microsoft Portable Executable and Common Object File Format Specification Revision 8.1. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx, 2010.

[19] Chris Eagle. The IDA Pro book, No Starch Press, first edition, Aug. 2008.

[20] Thomas H. Cormen, Charles E. Leiserson, et al. Introduction to algorithms. MIT Press, second edition, 2001.

[21] VX Heavens, http://vx.netlux.org/, 2010.

[22] Kaspersky Monthly Malware Statistics: May 2009, http://www.kaspersky.com/news?id=207575832, 2010.

[23] md5 checksum, http://www.etree.org/md5com.html, 2010.