Hindawi
Security and Communication Networks
Volume 2019, Article ID 1043794, 12 pages
https://doi.org/10.1155/2019/1043794

Research Article

A Malware and Variant Detection Method Using Function Call Graph Isomorphism

Jinrong Bai,[1] **Qibin Shi**,[2] **and Shiguang Mu**[3]

[1]*School of Mathematics and Information Technology, Yuxi Normal University, Yuxi 653100, China*
[2]*Yuxi E-government Network Management Center, Yuxi 653100, China*
[3]*School of Physics and Electronic Engineering, Yuxi Normal University, Yuxi 653100, China*

Correspondence should be addressed to Jinrong Bai; baijr223@163.com

Received 10 May 2019; Accepted 31 August 2019; Published 22 September 2019

Academic Editor: Vincenzo Conti

The huge influx of malware variants are generated using packing and obfuscating techniques. Current antivirus software use byte signature to identify known malware, and this method is easy to be deceived and generally ineffective for identifying malware variants. Antivirus experts use hash signature to verify if captured sample is one of the malware databases, and this method cannot recognize malware variants whose hash signatures have changed completely. Function call graph is a high-level abstraction representation of a program and more stable and resilient than byte or hash signature. In this paper, function call graph is used as signature of a program, and two kinds of graph isomorphism algorithms are employed to identify known malware and its variants. Four experiments are designed to evaluate the performance of the proposed method. Experimental results indicate that the proposed method is effective and efficient for identifying known malware and a portion of their variants. The proposed method can also be used to index and locate a large-scale malware database and group malware to the corresponding family.

1. Introduction

With rise of widespread Internet access, malware have brought the severe challenge to world. Malware, short for malicious software, are any software used to disrupt computer operation, gather sensitive information, gain unauthorized access to private computer systems, and make illegal profit for malware owner [1]. Malware can be divided into many different categories based on their function, such as Virus, Trojan, Worm, Rootkit, Backdoor, DoS, and Exploit. Malware can be used to take control of compromised computers to do whatever the malware owner wants. Governments or military organizations have also designed malware to attack their enemies.

Malware can be treated as byte sequence, therefore current antivirus software use particular byte sequence as signature to identify malware. However, many malware creators modify early malware to generate new variant which needs its own signature to be correctly identified. Moreover, many toolkits are used to create hundreds of thousands of variants from the same malware. This way to evade signature-based detection has become a popular tactic. According to 2015 Internet security threat report of Symantec [2], only 13.9 percent of captured malware could be identified using signature-based detection technology in 2014. Moreover, the top 10 list of malware families accounted for 33 percent of all malware blocked in 2014 [2]. Therefore, it is urgent to research on effective malware variant detection method.

Since each individual's fingerprint is unique, fingerprint is used as the characteristic to identify individual. A program fingerprint that distinguishes from other programs has been explored by many researchers, and different fingerprint representations of a program are proposed in recent years. Particular byte, instruction, or hash signature of a program is widely used in current malware detection systems. But those methods are generally ineffective against new malware variants for the following reasons: (1) a minor modification in source codes can lead to significant changes in hash signature; (2) automated obfuscation techniques are employed to create new variants which are semantically equivalent to, but syntactically

different from, original malware; (3) polymorphic malware apply encryption and decryption techniques to change constantly the body of malware.

Particular byte, instruction, or hash signature ignore high-level internal structure of malware. Graph is prevalently used in depicting structural information, and algorithms about graph theory have been studied for about several hundreds of years. Graph representation of a program opens a door to apply mathematical methods to detect malware. Graph signature representation is high-level abstraction of a program and thus more stable and resilient than syntactic-based signature. Therefore, it is a feasible way to correctly identify known malware and their variants.

The assembly code of a program is composed of many functions. Each function performs a specific task and is a sequence of machine instructions. The function is a callable unit and can be invoked several times from other functions or itself during execution of the program. There exists the invocation relationship between functions. The definition, parameters, and invocation relationship of the function can be clearly seen from the assembly code. The invocation relationship between functions can be represented as the graph structure. Each function is denoted as a vertex of the graph, and the invocation relationship from function x to function y is denoted as a directed edge from vertex x to vertex y. Therefore, a program can be represented as function call graph (FCG).

There exists fairly small probability of the FCG of a program same with the FCG of another different program. FCG can be used as signature to identify malware. A malware detection method that represents program as FCG is proposed in this paper. Two kinds of graph isomorphism algorithms are employed to test whether the FCG of an unknown program is isomorphic with known malware or not. If yes, the unknown program is recognized as malware. The proposed method achieves 100% true-positive rate and 0% false-positive rate for known malware detection. Moreover, a large portion of packed samples and malware variants can also be correctly identified by our method. The average matching time of each benign program versus 5340 malware is 0.02 seconds, and the matching time of each malware versus 5340 malware ranges from 0.02 to 11.43 seconds. The proposed method can be used to identify known malware and its variants, index and locate a large-scale malware database, and classify malware variants.

The remainder of the paper is organized as follows. Section 2 presents a brief overview of the previous research efforts for detecting malware based on graph representation. Section 3 gives the definitions related with our approach. Section 4 describes the design and implementation of our method. Section 5 briefly introduces the existing graph isomorphism algorithms and proposes a new graph isomorphism algorithm which is special for FCG. In Section 6, we design four experiments to evaluate the performance of the proposed method and make comparisons with a related method. Section 7 discusses experimental results and presents the potential application and limitations of the proposed method. Finally, Section 8 gives our conclusions and summarizes the contributions of this paper.

2. Related Work

Researchers have made a limited amount of research studies on the graph representation of a program for malware detection/classification. Those studies represent programs as weighted opcode graph, control flow graph, data flow graph, function call graph, and system call graph. Approximate graph matching, similarity measure, data mining, and machine learning methods are used to test if an unknown program is sufficiently similar with known malware.

Hashemi et al. [3] represented disassembled executable as weighted opcode graph. In this graph, the vertex denotes the instruction of the program, and the weight of the edge represents the transition probability between the instructions. They employed power iteration method to embed this graph into eigenspace which was used to train an ensemble of classifiers for malware detection. The main challenge of this approach is the difficulty of graph extraction and the complexity of the embedding algorithm.

Nguyen et al. [4] built control flow graph (CFG) from binary code, then converted the CFG to an image-based feature representation, and finally employed convolutional neural network to perform malware detection/classification. Bonfante et al. [5] also constructed CFG from disassembled executable. They used tree automata technique to transform CFG into trees which were employed to perform efficient signature matching.

Cesare et al. [6] represented a malware as a set of CFGs which were then decompiled and structured into strings. They used either fixed size k-subgraphs or q-grams to extract features. The prefiltering algorithm was used to filter the most relevant features as feature vectors. To compare the query signature to malware signature database, they used the normalized compression distance to perform a similarity search. Their evaluation showed that q-gram features generated more accurate results than k-subgraph features.

CFG increases the level of abstraction, utilizes effectively the syntactic and semantic information of disassembled executable, and is more suitable for detecting self-mutating malware. The limitations of CFG-based approaches are as follows: (1) when control flow is obfuscated, it results in different feature representation; (2) opaque predicates are hard to determine statically, and conditional branches are always evaluated to the same path; and (3) CFG generation and feature transformation is of high complexity.

Data flow graph (DFG), which is a higher-order analog to CFG, reflects data flow relationship among program instructions. Kolbitsch et al. [7] proposed a malware variant detection method based on DFG of system calls. The malicious samples were executed in a controlled environment to capture their system call trace. Program behavior was represented as a DFG where the vertex is system call and the edge is data flow between system calls. When the output of system call x is used as the argument of system call y, a directed edge is introduced from the vertex x to the vertex y. The signature set is composed of DFGs of the malicious samples. The system call trace of an unknown program was monitored and matched with the signature set. If this

program matches with one of the signature sets, it is considered malicious.

Wuchner et al. [8] also represented binary executable as quantitative DFG. In this DFG, nodes are system entities like processes, files, registry, or network sockets, and edges represent data flow between system entities as a result of system call executions. They employed compression-based mining to derive detection models and identify unknown malware.

Mpanti et al. [9] constructed system call dependency graph (SCDG) through taint analysis over the execution of malware samples, then grouped system calls into system call groups with respect to their functionality and finally extracted abstract coverage graph based on domination relations. They used cover similarity between coverage graphs to distinguish malware and benign software.

Ding et al. [10] also generated SCDG graph by tracing the propagation of the taint data, but they extracted a common behavior graph for each malware family and used a maximum weight subgraph matching algorithm to detect malware.

As with most dynamic malware analysis approaches, DFG- or SCDG-based methods can conquer packed or obfuscated malware effectively, but they suffer from the issue of execution path coverage or in general environment-sensitive.

Carrera and Erdélyi [11] represented a program as function call graph (FCG). The approximate distance or similarity between FCGs was computed, and the hierarchy clustering algorithm was applied to obtain a taxonomy of malware. This approach is helpful in comparing similarities and differences between malware variants.

Hu et al. [12] used same FCG representation with [11], and graph edit distance (GED) between pairs of graphs was computed to locate the nearest neighbor of sample file in the malware database. This method can use graph similarity to test if an unknown program is sufficiently similar to previously seen malware. The computation of GED can be formulated to find a minimal objective function of the total matching cost. Kostakis et al. [13] applied the simulated annealing algorithm to find the optimal solution of the cost function. This improvement performs faster and yields better results.

Xu et al. [14] used similarity metric of FCG to identify obfuscated malware. The sample file was represented as FCG, and graph-coloring technique was used to map the vertex of testing FCG to the vertex of FCG signature. The maximum common edges were used to compute the similar score of two FCGs. Experimental results showed that malware in the same family had high similarity score.

Kinable and Kostakis [15] represented the malware sample as FCG. Since exact solutions for graph edit distance were computationally expensive, they computed pairwise graph similarity scores via approximate graph edit distance. To facilitate the discovery of similar malware samples, they employed several clustering algorithms, including k-medoids and DBSCAN, to group samples to their family. Experimental results showed that it was indeed possible to detect malware families via call-graph clustering.

Hassen and Chan [16] extracted FCG representations from disassembled malware binaries. They converted FCG representation into a vector representation and then applied machine learning algorithms on these vectors to group malware samples into malware families. This approach had significant performance gains by adopting a linear time FCG vector representation, but its vector representation failed to preserve every detail of a graph structure.

Above methods employ different graph representation to explore malware detection problem or indexing of a malware database. Those methods have made many significant contributions to information security area, but also suffer from the following limitations: (1) approximation match is used in most of these methods, and it leads that there exist false negatives and false positives in those approaches; (2) since the similarity measurement algorithm is not efficient, polynomial complexity algorithms need to be explored; (3) a few fine-grain graph representations increase the scale and complexity of the graph; (4) packed or obfuscated malware detection is still the challenging problem for most of these methods. In sum, the results of above methods are encouraging, and new automated malware identification and classification methods based on graph representation should be explored further.

3. Definitions

In this section, we give the basic definitions related with the proposed method.

Definition 1. A graph is an ordered pair $G = <V, E>$, where V is a set of vertices and E is a set of edges that connect some pairs of vertices. $E \subseteq V \times V$, $(x, y) \in E$, and $x, y \in V$. If the edge (x, y) is not an ordered pair and is identical to the edge (y, x), this graph is an undirected graph in which edges have no direction. If the edge (x, y) is the ordered pair of vertices and has a direction associated with vertices, this graph is a directed graph (or digraph).

Definition 2. Suppose $G_1 = <V_1, E_1>$ and $G_2 = <V_2, E_2>$ are two graphs. Graphs $G_1 = <V_1, E_1>$ and $G_2 = <V_2, E_2>$ are said to be isomorphic if there exists a bijection $\varphi: V_1 \longrightarrow V_2$ such that $\forall x, y \in V_1$, $(x, y) \in E_1 \Longleftrightarrow \varphi(x), \varphi(y) \in V_2, (\varphi(x), \varphi(y)) \in E_2$.

Definition 3. Suppose $G = <V, E>$ is a directed graph, where each vertex corresponds to a function, and the edge (x, y) implies that function x invokes function y, but not vice versa. This graph $G = <V, E>$ is denoted as function call graph which represents invocation relationships between functions in a program.

Definition 4. Suppose P is a set of computer programs, M is a set of malware, and B is a set of benign software. $M \subset P$, $B \subset P$, $M \cap B = \Phi$, and $M \cup B = P$.

Definition 5. Suppose S is a set of malware signatures. $m \in M$ and $s \in S$. If the s is the signature of malware m, signature s uniquely identifies malware m.

Definition 6. Suppose D is a malware detector. $D: P \times S \longrightarrow \{0,1\}$. If there exists a signature s in signatures set matching with program p such that program p is identified as malware, it is denoted as $\exists s \in S : D(p,s) = 1 \Longrightarrow p \in M$. If there does not exist a signature s in signatures set matching with program p such that program p is identified as benign software, it is denoted as $\forall s \in S : D(p,s) = 0 \Longrightarrow p \in B$.

Definition 7. Suppose T is a function call graph generator which generates an FCG as signature of program p. $T(p) = s_g$, where s_g is the FCG signature of program p.

Definition 8. Suppose D_g is a malware detector. $D_g: P \times S \longrightarrow \{0,1\}$. If there exists an FCG signature s_g in signatures set isomorphic with the FCG signature of program p such that program p is identified as malware, we write $\exists s_g \in S : D_g(T(p), s_g) = 1 \Longrightarrow p \in M$. If there does not exist an FCG signature s_g in signatures set isomorphism with the FCG signature of program p such that program p is identified as benign software, we write $\forall s \in S : D_g(T(p), s) = 0 \Longrightarrow p \in B$.

Definition 9. $\mathrm{TP} = |\{p \mid p \in M \wedge \exists s \in S : D(p,s) = 1\}|$, where TP (true positive) is the number of correctly identified malware. $\mathrm{TN} = |\{p \mid p \in B \wedge \forall s \in S : D(p,s) = 0\}|$, where TN (true negative) is the number of correctly identified benign programs. $\mathrm{FP} = |\{p \mid p \in B \wedge \exists s \in S : D(p,s) = 1\}|$, where FP (false positive) is the number of falsely identified benign programs. $\mathrm{FN} = |\{p \mid p \in M \wedge \forall s \in S : D(p,s) = 0\}|$, where FN (false negative) is the number of falsely identified malware.

Definition 10. $\mathrm{TPR} = \mathrm{TP}/(\mathrm{TP} + \mathrm{FN})$, where TPR is true positive rate which is the percentage of programs labeled as malicious that can receive correct label by the malware detector. $\mathrm{FPR} = \mathrm{FP}/(\mathrm{TN} + \mathrm{FP})$, where FPR is false positive rate which is the percentage of benign software falsely identified by the malware detector. $\mathrm{accuracy} = (\mathrm{TP} + \mathrm{TN})/(\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN})$, where the accuracy is the number of absolutely correctly identified instances, either malware or benign software, divided by the entire number of instances.

4. Description of the Proposed Method

4.1. System Overview. The proposed detection method is divided into two main modules: FCG signature database generation module and malware detection module. An overview of the system is shown in Figure 1. The generation procedure of the FCG signature database is as follows: (1) if sample is packed, an automatic or manual method is used to unpack the packed sample; (2) IDA Pro is used to disassemble samples into intermediate assembly code; (3) FCG is generated based on the invocation relationship of functions in assembly code; (4) FCG signatures of known samples are added to the FCG signature database.

The malware detection procedure is as follows: (1) the same procedure with FCG signature generation module is performed to generate the FCG of testing sample; and (2) the graph isomorphism algorithm is employed to test if the FCG of testing sample is isomorphic with one of the FCG signature databases. If yes, testing sample is identified as malware; elsewise, it is regarded as a benign program.

4.2. System Details. In this subsection, we describe our method in more details. In particular, we first discuss how we disassemble a program binary to assembly code. Then, we introduce our way to automatically extract FCG from assembly code. Finally, we present our approach to match the FCG of testing sample with the FCG signature database.

First, we use an automatic or manual method to unpack packed samples. The PEID is used to detect whether the sample files are packed with packer. If yes, we employ Ether, a dynamic universal unpacking tool, for automatically unpacking sample files. The majority of packed samples can be automatically unpacked by Ether and then successfully disassembled. However, a few samples cannot be successfully unpacked. We manually unpack hard samples using specialized unpacker for specific packing tool or dynamic analysis tool (OllyDBG).

Next, we use IDA Pro to disassemble the sample files into assembly code. IDA Pro is a powerful disassembler which performs automatic code analysis and generates assembly code from executable binary. In the process of disassembly, IDA Pro automatically deals with the structure of each function according to the control flow of instructions. When IDA Pro believes that a sequence of program instructions have a complete function structure, it will treat these instructions as a function. For each function, when IDA Pro can identify the original name of the function, it will label this function with its original function name. When IDA Pro cannot identify the original name of the function, the starting address of function is denoted as the name of function. At the same time, IDA Pro also labels each function with incremental positive integer according to disassembly order.

After disassembling the sample files, the invocation relationship between functions in assembly code can be represented as FCG. Each vertex of the FCG has been labeled with the function name or unique consecutive integer. Figure 2 is the FCG of "Backdoor.Win32.Sepro" file, and each vertex of the FCG is labeled with the function name. Figure 3 is also the FCG of "Backdoor.Win32.Sepro" file, and a unique positive integer is used to label each vertex of the FCG. When we use IDA Pro to generate the FCG of a sample file more than once, the structure of FCG and vertex label is identical. However, when the sample file is obfuscated or packed, a part of function names is changed. The reason is that a few functions are labeled with the starting address of function, and the starting address of function undergoes change in obfuscated or packed samples. Due to same disassembly order, when the vertex is labeled with unique positive integer, most of the obfuscated samples retain the same vertex label with original samples. Moreover, when the packed samples can be correctly unpacked, the positive

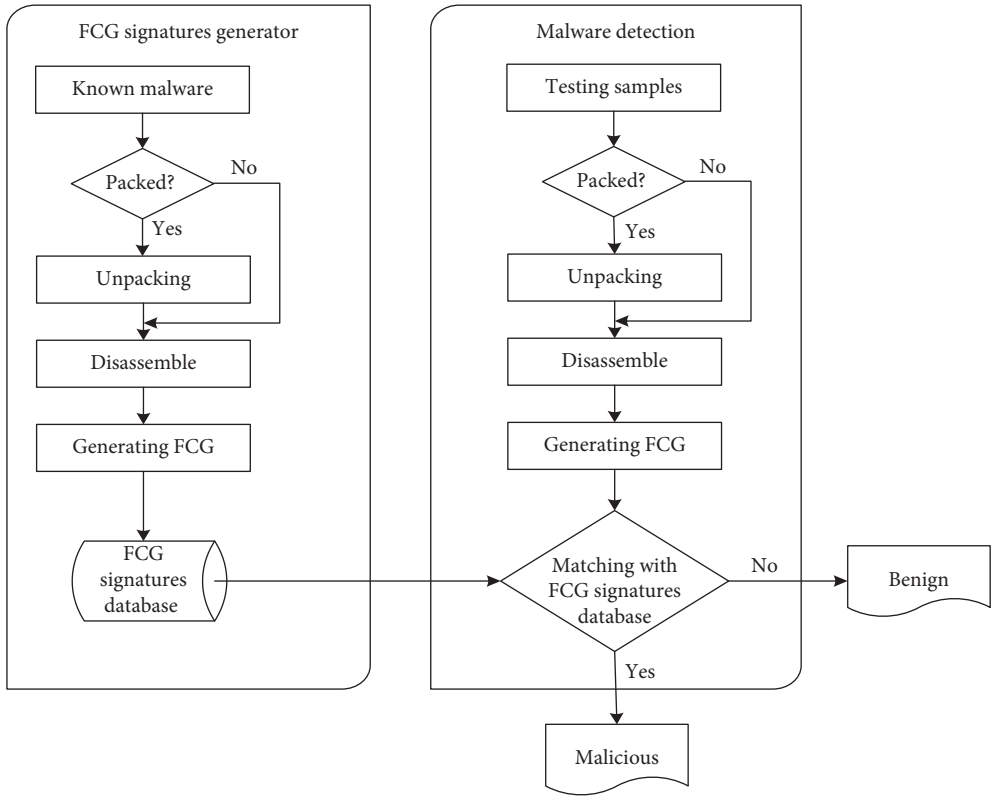

Figure 1: System overview.

integer label of each vertex in most of the unpacked samples are also the same with that of original samples. In our experiment, a unique positive integer is used to label the vertex of FCG.

"IDC scripts" is the built-in scripting language of IDA Pro and can offer assistance for repeated tasks. In order to generate automatically FCG of samples in batch, a Python program and an IDC script are written to implement this task. The IDC script call GenCallGdl function to generate FCG. The Python program lists all samples and makes IDA Pro to execute IDC script to generate automatically FCG in batch mode. In this way, we automatically extract FCG of all samples.

Eventually, all known malware are used to generate FCGs as a signature database. In detection phrase, the same process is employed to generate FCG of testing sample which is fed, along with each FCG of the signature database, to graph isomorphism algorithms for malware detection. A testing sample is identified as malware when its FCG is isomorphic with one of the FCG signature databases. Otherwise, it is regarded as benign software.

5. Graph Isomorphism Algorithms

The graph isomorphism problem can be simply expressed as the computational problem of determining whether two graphs which look differently are in fact structurally equivalent. In computational complexity theory, the graph isomorphism problem belongs to NP, but it is not known to

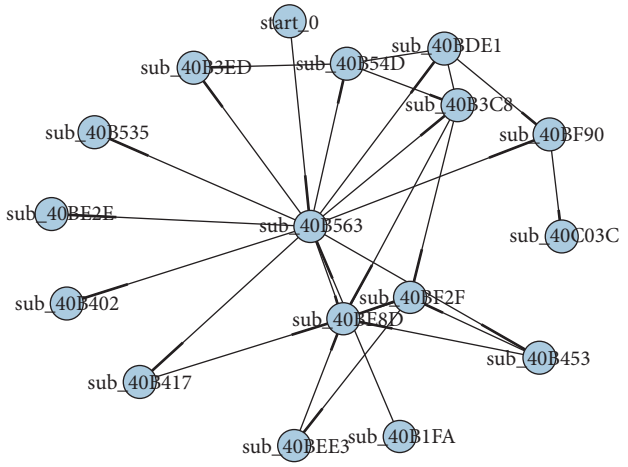

Figure 2: The FCG of Backdoor.Win32.Sepro (each vertex is labeled with the function name).

belong to either of P- or NP-complete. The theoretical investigation research studies on the graph isomorphism problem show that the provably polynomial time algorithms for all types of graph still have not been found. Currently, heuristic methods are the main way to solve this problem. The researchers have proposed different heuristic methods such as Ullman algorithm [17], VF algorithm [18], and its improved version—VF2 algorithm [19]. The inexact graph matching algorithms based on the genetic algorithm [20] and neural network [21] are also proposed to solve this problem.

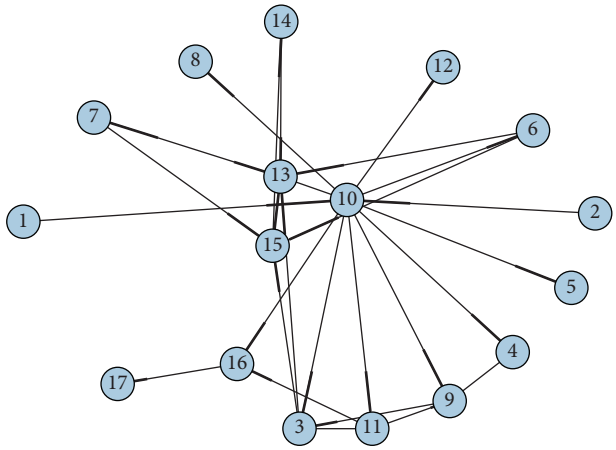

Figure 3: The FCG of Backdoor.Win32.Sepro (each vertex is labeled with the unique positive integer).

5.1. VF2 Algorithm. The VF2 algorithm is most commonly used for both graph isomorphism and subgraph isomorphism and is one of the fastest graph isomorphism algorithms. The VF2 algorithm is based on a depth-first search strategy to find a mapping M between the two graphs $G_1 = <V_1, E_1>$ and $G_2 = <V_2, E_2>$. The mapping M can be described by the set of ordered pair $M = \{(m, n) \in V_1 \times V_2\}$, which represents the mapping of a vertex m of G_1 to a vertex n of G_2. A partial mapping $M(s)$ is a subset of M and associated with the state s of the matching process. $M(s)$ denotes the subgraph $G_1(s)$ isomorphic with the subgraph $G_2(s)$, and the subgraph $G_1(s)$ and $G_2(s)$ include the vertices of $M(s)$. If a candidate pair of vertices (u, v) is feasible, it is added to the partial mapping $M(s)$. The mapping search is repeated until the matching solution is obtained or the search space has been fully explored. The brute force approach can be used to compute all the possible partial solutions and find the matching solution, but it is not feasible for large-graph matching. In order to improve the performance of this algorithm, a set of syntactic and semantic feasibility rules [19] are used to prune the search space and reduce the computational cost of the matching process. The space complexity of the VF2 algorithm is $O(n)$, where n is the number of vertices. The time complexity of the VF2 algorithm is $O(n! \cdot n)$ in the worst case and $O(n^2)$ in the best case.

5.2. Our Proposed FCGiso Algorithm. Suppose $G_1 = <V_1, E_1>$ and $G_2 = <V_2, E_2>$ are two graphs. According to the definition of graph isomorphism, three necessary conditions for graph isomorphism can be described as follows:

(1) The number of vertices of G_1 is equal to the number of vertices of G_2, denoted as $|V_1| = |V_2|$

(2) The number of edges of G_1 is equal to the number of edges of G_2, denoted as $|E_1| = |E_2|$

(3) If the vertex v of G_1 is mapped to the vertex v' of G_2 under any isomorphism, then the adjacent vertices set of v is equal to the adjacent vertices set of v'.

If each vertex of G_1 can be mapped to the corresponding vertex of G_2 using unique label of vertex, the above three conditions are the sufficient and necessary conditions for graph isomorphism. According to the above three conditions, the FCGiso algorithm which is special for FCG isomorphism problem is proposed in this paper and is presented in Algorithm 1. As shown in Algorithm 1, the FCGiso algorithm is carried out with four major steps:

Step 1: if $((2\text{abs}(|V_1| - |V_2|))/(|V_1| + |V_2|)) < 0.05$, go to step 2; otherwise, G_1 is not isomorphic with G_2; where $|V_1|$ is the number of vertices of G_1 and $|V_2|$ is the number of vertices of G_2.

Step 2: if $((2\text{abs}(|E_1| - |E_2|))/(|E_1| + |E_2|)) < 0.05$, go to step 3; otherwise, G_1 is not isomorphic with G_2; where $|E_1|$ is the number of edges of G_1 and $|E_2|$ is the number of edges of G_2.

Step 3: for each vertex v in G_1:

(1) Obtain the direct successor vertices set (outnodes_1) of vertex v in G_1.
(2) Find the vertex v' of G_2 having same label with vertex v.
(3) Obtain the direct successor vertices set (outnodes_2) of vertex v' in G_2.
(4) If $(|\text{outnodes}_1 \cap \text{outnodes}_2|/|\text{outnodes}_1 \cup \text{outnodes}_2|) < 0.95$, G_1 is not isomorphic with G_2.

Step 4: graph G_1 is isomorphic with G_2.

(Remarks: Since FCGs are labeled directed graphs, investigating the adjacent vertices is identical either when testing the outgoing or the incoming edges. Therefore, we only utilize the outnodes to perform graph matching.)

Suppose $G = <V, E>$ is a directed graph, n is the number of vertices, m is the number of edges, and $d(v_i)$ is the outdegree of vertex v_i. For the FCGiso algorithm, the number of elementary operations which are performed to match vertex v_i is at most $3d(v_i)$. The number of all elementary operations performed by the FCGiso algorithm is at most $n + \sum_{i=1}^{n} 3d(v_i) = n + 3m$, where $m = \sum_{i=1}^{n} d(v_i)$. $n + 3m \le n + 3n^2$, where $m \le n^2$. Therefore, the time complexity of the FCGiso algorithm is $O(n^2)$ in the worst case.

6. Experiments

In this section, we construct four datasets and design four experiments to evaluate comprehensively the proposed detection method. Experiment I aims to verify the detection ability of known malware. Experiment II is conducted to evaluate the detection ability of packed samples. Experiment III is designed to identify variants of known malware. Experiment IV is conducted to evaluate the performance of variant classification. Finally, we compare the GED method with our method. These experiments are presented in the following subsections. The experiments are run on the machine with Intel Xeon CPU E2—2650 (4 Cores) @ 2.00 GHz with 8 GB RAM and the Windows Server 2008 operating system.

Input: $G_1 = <V_1, E_1>$ and $G_2 = <V_2, E_2>$ are labeled directed graphs.
Output: If Graph G_1 is isopmorphic with G_2, then return"true"; otherwise "false".
if $((2\text{abs}(|V_1| - |V_2|))/(|V_1| + |V_2|)) \geq 0.05$ **then**
  return false;
else if $((2\text{abs}(|E_1| - |E_2|))/(|E_1| + |E_2|)) \geq 0.05$ **then**
  return false;
else
  for each $v \in V_1$
    outnodes$_1 \longleftarrow \{m \mid m \in V_1, (v, m) \in E_1\}$; // director successor vertices set of vertex v
    $v' \longleftarrow$ the vertex of G_2 having same label with v;
    outnodes$_2 \longleftarrow \{n \mid n \in V_2, (v', n) \in E_2\}$; // director successor vertices set of vertex v'
    if $(|\text{outnodes}_1 \cap \text{outnodes}_2|/|\text{outnodes}_1 \cup \text{outnodes}_2|) < 0.95$ **then**
      return false;
    end if
  end for each
end if
return true;

Algorithm 1: FCGiso(G_1, G_2).

6.1. Experiment I: Detection of Known Malware. The dataset D1 is made up of 4985 instances of benign software and 5340 instances of malware, and all samples are in Windows PE format. We collect malware from VX Heavens Collection [22]. The benign samples are gathered from a fresh installed Microsoft Windows 10 and the website SourceForge. The benign samples consist of different categories of programs such as system software, application software, utility software, and media software. Norton AntiVirus software is used to verify that each benign sample is indeed benign. 5340 instances of malware are used to construct an FCG signature database, and whole dataset D1 is used as testing samples.

We use two algorithms (VF2 algorithm and FCGiso algorithm) for graph isomorphism test. The VF2 algorithm performs graph matching based on the structure of graph and does not utilize the vertex label. The vertex label is exploited effectively in the FCGiso algorithm which maps each vertex of query graph to the corresponding vertex of destination graph based on vertex label. The experimental results are presented in Table 1. The matching time of two algorithms is shown in Figure 4.

As can be seen from Table 1, two algorithms achieve high detection rate with low false-positive rate. It is not surprising that the TPR of both algorithms is 100% because signature-based detection methods can easily reach this result. The FCGiso algorithm correctly identifies all samples, but the VF2 algorithm falsely identifies a benign program as malware. Falsely identified sample and corresponding signature are disassembled into assembly codes for comparison and analysis. It shows that the FCG structure of two files is identical, but the assembly codes of two files are different. There is high possibility that two distinct files have identical FCG structure when huge amount of samples are used. But it is highly unlikely that distinct files have identical FCG structure and same label for each vertex of FCG. Therefore, the FCGiso algorithm achieves 0% false alarm rate. It should be noted that both algorithms identify a benign sample "netsetup.exe" as malware "Trojan.DOS.Ubuster." By comparing the assembly codes of two samples, it is found that the assembly codes of two samples are almost identical and "netsetup.exe" is variant of "Trojan.DOS.Ubuster." Norton AntiVirus software was used to scan two files. "netsetup.exe" is regarded as a benign program, and "Trojan.DOS.Ubuster" is recognized as malware. The reason is that AntiVirus software is hard to identify variants of known malware.

Table 1: Detection result of known malware.

Graph isomorphism algorithm	TPR (%)	FPR (%)	Accuracy
VF2	100	0.02	99.9
FCGiso	100	0	100

It can be seen from Figure 4 that both algorithms can decide whether the testing sample is isomorphic with one of the FCG signature databases in a short period of time. With the VF2 algorithm, the average matching time of each benign program versus 5340 instances of malware is 0.0209 seconds, and the matching time of each malware versus 5340 instances of malware ranges from 0.05 seconds to 25.25 seconds. For the FCGiso algorithm, the average matching time of each benign program versus 5340 instances of malware is 0.02 seconds, and the matching time of each malware versus 5340 instances of malware ranges from 0.02 seconds to 11.43 seconds. With the increase of the number of vertices, the matching time of the VF2 algorithm increases more rapidly than that of the FCGiso algorithm when each malware is matched with the FCG signature database. Compared with the VF2 algorithm, the FCGiso algorithm has better performance in either benign program or malware versus FCG signature database. The matching time of each benign program versus 5340 instances of malware has a notable peak when the number of vertices reaches 3750. The reason is that FCG of "netsetup.exe" file is isomorphic with FCG of "Trojan.DOS.Ubuster."

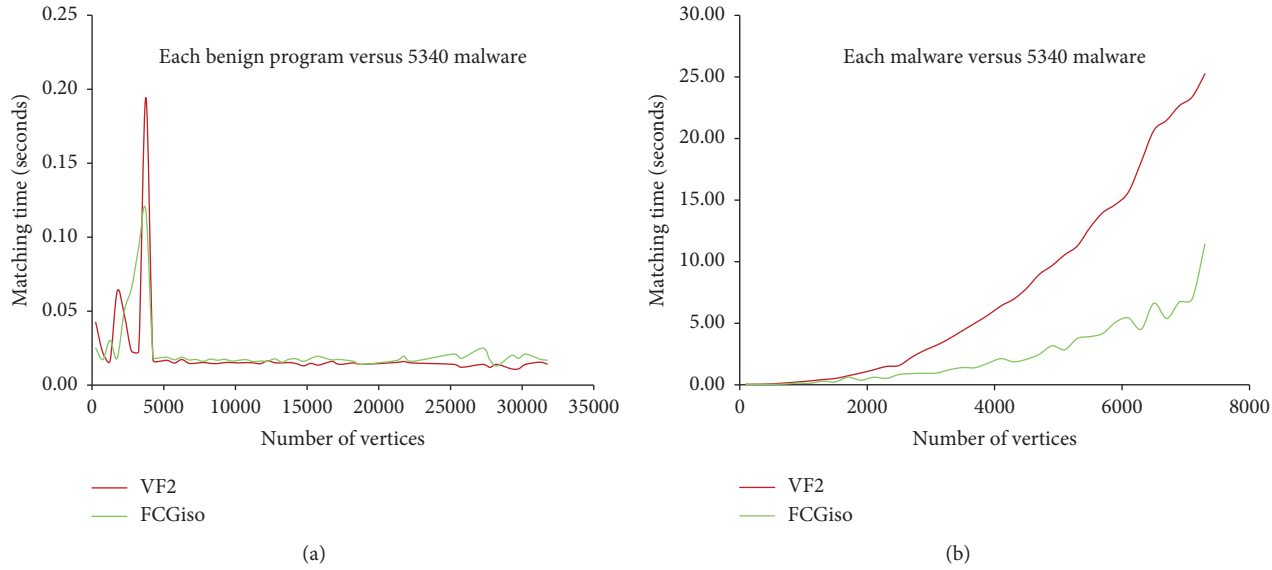

Figure 4: Matching time of the VF2 algorithm and FCGiso algorithm.

6.2. Experiment II: Detection of Packed Samples. To test the performance of identifying packed samples, we construct the dataset D2. In this dataset, "notepad.exe" and "calc.exe" are used as signatures. We use 9 kinds of packers shown in Table 2 to pack signature samples as testing dataset. The VF2 algorithm and FCGiso algorithm are used to evaluate the performance of identifying packed samples. Two algorithms achieve identical experimental results which are listed in Table 2.

It can be seen from Table 2 that the proposed method can identify most of packed samples and achieve 83.3% detection rate. Our method cannot identify the packed samples which are packed by ASProtect packer. ASProtect packer employs compression, encryption, antidebuggers, and anti-disassembler techniques to protect the samples. The universal unpacking tools are hard to correctly unpack these packed samples. "calc.exe" packed by MEW packer cannot be identified by our method, but "notepad.exe" packed by MEW packer can be identified correctly. It is obvious that the universal unpacking tools cannot unpack correctly all samples packed by MEW packer. It is noteworthy that original file, packed file, and unpacked file have different file size and hash-based signature. We can conclude that the hash-based signature method is easy to be deceived and are generally ineffective for identifying packed malware. But our method can identify most of packed samples when the packed samples can be unpacked correctly.

6.3. Experiment III: Detection of Malware Variants. Experiment III is conducted to evaluate whether the proposed method could identify variants of known malware. In this experiment, we use dataset D3 which consists of 239 instances of malicious samples from four malware families. These samples are collected from VX Heavens Collection [22], and all samples have unique MD5 hash value. One of each malware family is used as signature, and the remaining of each malware family constitute the testing samples. We use the VF2 algorithm and FCGiso algorithm to conduct the experiment and reach identical experimental results which are reported in Table 3.

It can be seen from Table 3 that the detection rate of each malware family ranges from 52.2% to 99.3%. By analyzing those samples, we found that packing and obfuscating techniques are used to generate a large portion of malware variants, and a few malware variants are generated by modifying the payload of original malware. Our method can identify most of packed and obfuscated malware but cannot detect malware variants whose FCG has changed.

Malware writers frequently use obfuscation techniques to generate new malware variants which have equivalent semantics but different syntactical representation with original malware. Commonly used obfuscating techniques include junk code insertion, equivalent instructions substitution, register reassignment, instructions reordering, and branch inversion. The first four obfuscating techniques do not change the FCG of original malware so that our method can correctly identify their variants. There is a certain degree of probability that the last obfuscating technique modify FCG of original malware to generate new variants which cannot be identified by our method. The detection rate of our method can be increased when several distinct FCG signatures of the same malware family are added to the FCG signature database. It can be concluded that our method has the ability to detect a portion of malware variants.

6.4. Experiment IV: Malware Variants Classification. In order to evaluate the performance of malware variants classification, we construct dataset D4 which is composed of 96,337 instances of malware collected from VX Heavens Collection [22]. The experimental procedure was as follows:

Table 2: Detection result of packed samples.

Packer	notepad.exe				calc.exe			
	Original size (k)	Packed size (k)	Unpacked size (k)	Detection result	Original size (k)	Packed size (k)	Unpacked size (k)	Detection result
UPX	65	49	124	Yes	112	55	163	Yes
ASPack		56	99	Yes		70	145	Yes
FSG		46	132	Yes		60	191	Yes
MEW		44	128	Yes		56	243	No
PE-PACK		37	96	Yes		45	143	Yes
WinUPack		44	160	Yes		52	215	Yes
ASProtect		369	69	No		373	114	No
PackMan		53	124	Yes		69	171	Yes
PECompact		49	112	Yes		56	159	Yes

Table 3: Detection result of malware variants.

Malware family	Signature	Number of malware variants	Number of identified variants	Detection rate (%)
Virus.Win32.Mooder	Virus.Win32.Mooder.c	10	8	80
Trojan.Win32.BHOLamp	Trojan.Win32.BHOLamp.mx	25	18	72
Backdoor.Win32.Small	Backdoor.Win32.Small.acp	158	157	99.3
Trojan-Downloader.Win32.Zlob	Trojan-Downloader.Win32.Zlob.qlr	46	24	52.2

Step 1. Initialize an empty FCG signature database.

Step 2. For each malicious sample, the FCGiso algorithm is used to determine whether there exists an FCG signature in the signature database which is isomorphic with this sample. If yes, this sample is classified into the corresponding malware family. Otherwise, this sample was added to the FCG signature database.

Step 3. Repeat step 2 until all malicious samples are grouped into the corresponding malware family.

The top 10 list of malware families are shown in Table 4. As can be seen from Table 4, the top 10 list of malware families contain 8354 samples which are about 8.6 percent of total samples. "Trojan-Banker.Win32.Banbra" is the largest family which contains 4671 instances of variants. The experimental results confirm that our approach could group malicious samples to the corresponding family. The drawback of our method is that a portion of malware variants whose FCG are different from original malware are classified into new family. With this problem, evolutionary relationship can be established between families with the help of artificial analysis. In sum, our method is accurate and efficient but needs to be improved to be more resilient.

6.5. Comparisons with the GED Method. Graph edit distance (GED) is defined as the least-cost edit operations (such as node insertion, edge insertion, node deletion, edge deletion, node substitution, and edge substitution) required in graph G_1 to make it isomorphic with graph G_2. Suppose $G_1 = <V_1, E_1>$ and $G_2 = <V_2, E_2>$ are two FCGs, n is the number of least-cost edit operations which are performed on graph G_1 to make it isomorphic with graph G_2, and the normalized graph edit distance (NGED) is defined as $2n/(|V_1| + |V_2|)$.

Table 4: The experimental results of variants classification.

Malware family	Number of samples
Trojan-Banker.Win32.Banbra.abc	4671
Backdoor.Win32.Small.aaa	1367
Trojan.Win32.Pakes.age	381
Trojan-Banker.Win32.Banbra.cuo	323
Trojan-Banker.Win32.Agent.y	323
Trojan-Banker.Win32.Banbra.dml	309
Trojan.Win32.Pakes.bdq	257
Trojan-Banker.Win32.Agent.f	255
Trojan-Banker.Win32.Bancos.adn	236
Backdoor.Win32.Poison.abo	232

Since NGED computation for FCGs with large number of vertices cannot be performed within acceptable time, we discard large graph samples from dataset D1 and construct dataset D5 whose number of vertices are between 40 and 190. This dataset consists of 532 instances of benign software and 1075 instances of malware. 1075 instances of malware are used to construct an FCG signature database, and the whole dataset D5 is used as testing samples. The vertex distribution of dataset D5 is shown in Figure 5.

Several studies used graph edit distance to compute the similarity of distinct sample files. We test GED-based methods using our datasets (D2, D3, and D5). The vertex difference ratio (VDR) of two graphs is defined as $(2\text{abs}(|V_1| + |V_2|))/(|V_1| + |V_2|)$. When benign programs are matched with the FCG signature database, we only perform matching between testing sample and signature whose VDR is below 0.1. When malicious samples are matched with the FCG signature database, we only perform matching between testing sample and signature having same number of vertices. Experimental results are listed in Table 5 and the matching time of the GED method is shown in Figure 6.

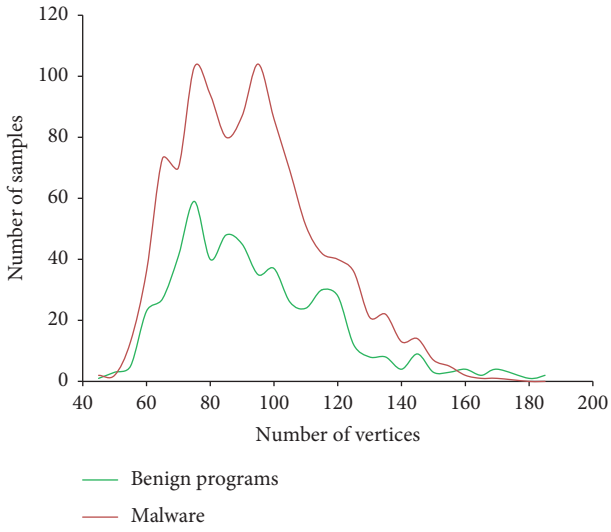

Figure 5: The distribution of dataset D5.

It can be seen from Figure 6 that the matching time is not positively correlated with the number of vertices, but it is related to the distribution of the FCG signature database. The reason is that if many FCG signatures have approximately equal number of vertices with the testing sample, NGED computation will consume a great deal of time. Moreover, the matching time between an FCG with 115 vertices and a FCG signature database is 5736.7 seconds, which is unacceptable performance. We can see from Table 5 that this method reaches similar results with our method when the threshold of NGED is 0.05. When the threshold of NGED is 0.2, this method achieves slightly better result than our method in detecting packed samples and malware variants, but the false alarm rate of detecting known malware is 37.1%. By analyzing Table 5 and Figure 6, we found that the GED method suffers below significant limitations: (1) this method needs to be improved due to its high processing overheads; (2) this method is more resilient than our method, but it also results in high false alarm rate; (3) this method is not feasible for the graph with large number of vertices. In sum, graph edit distance computation is more complex than the graph isomorphism problem, and our method is more effective and efficient when the testing sample is matched with a large FCG signature database.

7. Discussions

The FCG is a high-level abstract representation of a program, and several studies [11–14] use the FCG representation to identify malware variants. But those studies use the similarity metric to compare distinct samples, and they regard that exact graph match belonged to NP problem, and graph isomorphism algorithms could not be used to match the FCGs of distinct samples. However, the FCG of a program is a sparse-directed graph which makes the graph isomorphism algorithm's time complexity more lower. Therefore, graph isomorphism test can be performed in a short period of time. In our experiment, graph auto isomorphism test for an FCG with 64,934 vertices costs 102.3 seconds when the VF2 algorithm is used. The matching time of this FCG is 18.9 seconds when the FCGiso algorithm is used. It can be concluded that the graph isomorphism algorithm can be used to malware and variant detection based on FCG representation.

It could be intuitively thought that detection methods [11–14] might be more efficient than our method. Since many FCG signatures have approximately equal number of vertices and edges with the FCG of testing sample, those methods must compute the similarity metric between testing sample and each FCG signature. It makes those methods' time complexity more higher. However, only comparing the number of vertices and edges between two FCGs, our method can ascertain if there exists the possibility of isomorphism between testing sample and each FCG signature. Only when two FCGs have approximately equal number of vertices and edges, the graph isomorphism algorithm is employed to test if two FCGs are isomorphic. The studies [11, 13] exploited opcode sequence of function to assist in computing the approximate graph edit distance; hence, the algorithm's efficiency is significantly improved. But the opcode sequence of malware variants has undergone a certain degree of changes, there exists the possibility of false negatives in [11, 13].

Malware detection should at least consider the following properties: accuracy, efficiency, and resilience. The approaches in [11–14] may be more resilient than our method, but our method is more accurate and efficient than those approaches. Moreover, there exists the possibility of false alarms in those approaches due to their inexact characteristic. In antivirus software, the false alarm of detection method is almost intolerable. There exists fairly small possibility of false alarms in the byte-level signature-based detection method; therefore, it is widely used in current antivirus software. There exists fairly small probability that distinct files have identical FCG structure and same label for each vertex of their FCG unless those files are closely related, such as malware and its variants. Therefore, our method has very low false alarm rate.

Antivirus software commonly use hash signature or byte signature of malware to uniquely label malicious sample due to their low false alarm rate. Both syntactic-based signatures have poor resilient to minor modifications of malware and are generally unable to recognize packed or obfuscated variants. Furthermore, antivirus experts need to manually analyze and extract the particular byte signature of malware, but our method can automatically extract the FCG signature of malware. Four experiments are performed to analyze the validity and efficiency of the proposed approach in this study. Experimental results show that our method is not easy to be deceived and are effective and efficient for identifying known malware and a portion of their variants. FCG representation overcomes the weaknesses of syntactic-based signatures.

Antivirus companies received the huge influx of malicious samples, and they needed to quickly judge whether an unknown sample is one of the huge malware databases or a variant of known malware. Syntactic-based signature

Table 5: Experimental results comparisons.

Dataset	NGED ≤ 0.05		NGED ≤ 0.1		NGED ≤ 0.2		Our method	
	TPR (%)	FPR (%)	TPR (%)	FPR (%)	TPR (%)	FPR (%)	TPR (%)	FPR (%)
D2	83.3	0	83.3	0	88.8	0	83.3	0
D3	86.6	0	89.1	0	90.1	0	86.6	0
D5	100	0.4	100	8.3	100	37.1	100	0

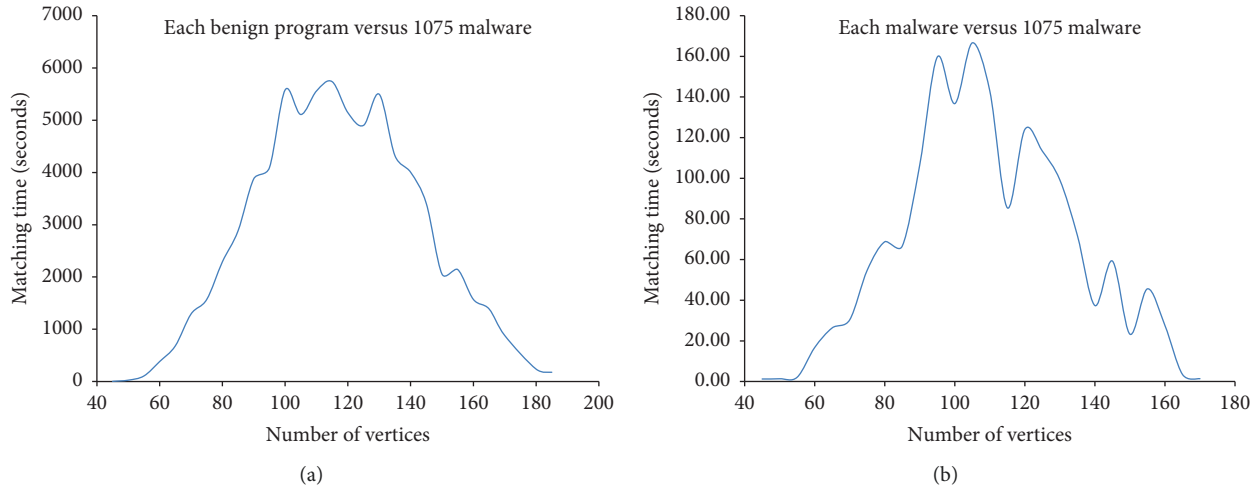

Figure 6: Matching time of GED method.

methods are generally ineffective for packed or obfuscated variants. Our method can be used to index and locate a large-scale malware database due to its validity and efficiency. Moreover, our method can be used to group variants of malware.

Our method also suffers from the following limitations: (1) our method can detect only known malware and a portion of their variants, and it cannot identify fully unknown malware; (2) compared with syntactic-based signature methods, our method needs more space to store the FCG signature database; and (3) malware writers can employ antidisassembling techniques to prevent a precise decoding of malware, so that our method cannot process this type of malware. Possible improvements to remove or alleviate above limitations are suggested as follows: (1) inexact graph isomorphism algorithms can be used to identify malware variants whose FCG has undergone minor changes, and (2) distributed computing platform can be used to implement our method. It not only solves the storage problem of huge FCG signature database, but also decreases significantly the matching time of graph isomorphism algorithms.

Machine learning-based detection methods can classify unknown samples as malicious or benign with high accuracy, but those methods may generate false alarms and are hard to pinpoint unknown sample to a specific malware family. Machine learning-based detection methods and signature-based detection methods have their own inherent advantages and disadvantages. Malware detection is a very complex problem; hence, a detection method alone cannot conquer this problem. Incorporating signature-based methods and machine learning-based methods may overcome the drawbacks of these methods and construct a more effective and efficient detection system.

8. Conclusions

Researchers have proposed several malware detection methods based on a variety of graph representations of a program. Those methods achieve good experimental results, but also suffer from limitations mentioned in Section 2. This study demonstrates how graph isomorphism algorithms are used to identify known malware and their variants based on the FCG representation of a program. Four experiments are conducted to evaluate the feasibility of our method. Experimental results show that our method is effective and efficient for identifying known malware and a portion of variants. Our method can also be used to index and locate large-scale malware database and group malware variants.

The main contributions of this paper include the following: (1) we propose a new malware detection method which represents a program as FCG and employs graph isomorphism algorithms to identify known malware and their variants; (2) we propose a simple and efficient graph isomorphism algorithm—FCGiso which effectively utilizes the vertex label of FCG to assist graph isomorphism test and outperforms the VF2 algorithm in our experiment; and (3) most of graph signature-based methods only compute the similarity metric of samples in the same family and do not present detection rate, false alarm rate, and matching time, but we perform a comprehensive performance investigation to verify the efficiency and validity of the proposed method.

Existing studies represent a program as nature form graph, such as control flow graph, data flow graph, function call graph, and system call graph. For nature form graph, it requires a complex algorithm to compute the similarity metric between testing sample and graph signature. We plan to explore hand-crafted graph representation which has enough semantic information to perform graph matching or similarity computing in our future research work.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Research Foundation of China, grant no. 61841206, by Local Undergraduate Universities (part) Joint Fund for Fundamental Research Project of Yunnan Province, China, under grant no. 2017FH001-055 and no. 2017FH001-101, and by Construction Plan of Key Laboratory of Institutions of Higher Education in Yunnan Province, China.

References

[1] S. Kramer and J. C. Bradfield, "A general definition of malware," *Journal in Computer Virology*, vol. 6, no. 2, pp. 105–114, 2010.

[2] Symantec Corporation, *Symantec Internet Security Threat Report*, Internet Security Threat Report, Symantec Corporation, Mountain View, CA, USA, 2015.

[3] H. Hashemi, A. Azmoodeh, A. Hamzeh, and S. Hashemi, "Graph embedding as a new approach for unknown malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, pp. 153–166, 2017.

[4] M. H. Nguyen, D. L. Nguyen, X. M. Nguyen, and T. T. Quan, "Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning," *Computers & Security*, vol. 76, pp. 128–155, 2018.

[5] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "Architecture of a morphological malware detector," *Journal in Computer Virology*, vol. 5, no. 3, pp. 263–270, 2009.

[6] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware VariantDetection," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 4, pp. 307–317, 2014.

[7] C. Kolbitsch, P. M. Comparetti, C. Kruegel et al., "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, pp. 351–366, Montreal, Canada, August 2009.

[8] T. Wuchner, A. Cislak, M. Ochoa, and A. Pretschner, "Leveraging compression-based graph mining for behavior-based malware detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 1, pp. 99–112, 2019.

[9] A. Mpanti, S. D. Nikolopoulos, and I. Polenakis, "A graph-based model for malicious software detection exploiting domination relations between system-call groups," in *Proceedings of the 19th International Conference on Computer Systems and Technologies*, pp. 20–26, Ruse, Bulgaria, 2018.

[10] Y. Ding, X. Xia, S. Chen, and Y. Li, "A malware detection method based on family behavior graph," *Computers & Security*, vol. 73, pp. 73–86, 2018.

[11] E. Carrera and G. Erdélyi, "Digital genome mapping–advanced binary malware analysis," in *Proceeding of the of Virus Bulletin Conference*, pp. 187–197, Chicago, IL, USA, 2004.

[12] X. Hu, T. C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pp. 611–620, Chicago IL, USA, November 2009.

[13] O. Kostakis, J. Kinable, H. Mahmoudi et al., "Improved call graph comparison using simulated annealing," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1516–1523, ACM, Pisa, Italy, 2011.

[14] M. Xu, L. Wu, S. Qi et al., "A similarity metric method of obfuscated malware using function-call graph," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 1, pp. 35–47, 2013.

[15] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, pp. 233–245, 2011.

[16] M. Hassen and P. K. Chan, "Scalable function call graph-based malware classification," in *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*, pp. 239–248, Scottsdale, AZ, USA, March 2017.

[17] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[18] J.-M. Jolion and W. G. Kropatsch, "Graph based representations in pattern recognition," in *Computing Supplement*, Lyon, France, April 1997.

[19] L. P. Cordella, P. Foggia, C. Sansone et al., "An improved algorithm for matching large graphs," in *Proceeding of 3rd IAPR-TC15 Workshop on Graph-Based Representations in Pattern Recognition*, pp. 149–159, Ischia, Italy, May 2001.

[20] A. D. J. Cross, R. C. Wilson, and E. R. Hancock, "Inexact graph matching using genetic search," *Pattern Recognition*, vol. 30, no. 6, pp. 953–970, 1997.

[21] B. J. Jain and F. Wysotzki, "Solving inexact graph isomorphism problems using neural networks," *Neurocomputing*, vol. 63, pp. 45–67, 2005.

[22] Virus Collection, 2011, http://vx.netlux.org.