Binary Executable Files Homology Detection with Genetic Algorithm

Jinyue Bian[1] and Quan Qian[1,2]
(Corresponding author: Quan Qian)
School of Computer Engineering and Science, Shanghai University[1]
Shanghai 200444, China
Materials Genome Institute, Shanghai University, Shanghai, China[2]
(Email: qqian@shu.edu.cn)
(Received Mar. 16, 2019; Revised and Accepted Dec. 29, 2019; First Online May 9, 2020)

Abstract

Software homology detection is very meaningful for software copyright protection and malicious code variants detection. In this paper, we propose a genetic algorithm to justify the binary code similarity. First of all, the binary executable files are converted into control flow graph , and then use genetic algorithm to compute the similarity among control flow graphs, which is regarded as the evaluation metric for software homology detection. The experimental results show that the method is not only effective, but also the average time efficiency is 0.3 times that of the classical algorithm of graph edit distance.

Keywords: Binary Executable Files; Control Flow Graph; Genetic Algorithm; Homology Detection

1 Introduction

With the rapid development of software, code plagiarism is emerging endlessly, which seriously threatens the software intellectual property rights. In addition, in March 2018, a total of 7,235,983 viruses were found in the National Computer Virus Emergency Center,36,931 new viruses were added, and 76,606,087 computers were infected [13]. Viruses or malware's escape from detection through code variants, but the core code does not change much. Therefore, detection of code similarity is very necessary. And the existing methods can be divided into two categories, source code based or binary code based. Considering sometimes we can not get the source code. Therefore, binary code based homology detection is more promising. That is, the similarity analysis based on binary code is very important.

For binary code similarity detection, graph based method plays an important role in which Control Flow Graph (CFG) is one of the most commonly used method. Therefore, for binary code homology detection, it can be transferred to graph similarity matching problems. That is, given two graphs, graph matching involves establishing the corresponding relations between their vertexes and considering the consistency of edge sets at the same time. CFG similarity comparison methods include graph edit distance (GED), string matching, execution sequence comparison, matching program basic blocks, and so on. However, in general, the computation complexity of these methods are quite large. Therefore, in this paper, we propose a new graph matching algorithm for binary code similarity analysis. First, the binary file will be converted into CFG with relatively complete control flow information using the dynamic and static combination technique. And then use genetic algorithm (GA) to calculate the similarity between CFGs. The algorithm can be used to accurately identify the isomorphism subgraph relationship and the exact identical CFGs, which can shorten the running time greatly, so as judging the software homology effectively.

The organization of the paper is as follows: Section II describes some related work. Section III presents the framework of the proposed method. The experimental results and detailed analysis are discussed in Section IV. Section V concludes the whole paper and outlines some directions of the future work.

2 Related Work

So far, there are certain amounts of research on homology identification related area. Here, we will give some background work in two aspects, including sequenced based analysis method and graph based method that closely related to this paper.

2.1 Sequence-based Analysis Method

Aiming at the problem of source code plagiarism, Guo proposed an improved code plagiarism detection algorithm based on abstract syntax tree, which can detect plagiarism effectively [22]. Koschke demonstrated how suffix trees can be used to obtain a scalable comparison,

and presented a method to improve the accuracy through user feedback and automatic data mining [10]. Liu presented an improved abstract syntax tree, which can effectively detect code plagiarism by modifying the variable type and adding meaningless variables [14].

However, the above mentioned methods are all sequence-based ones, that can be bypassed by interference techniques, such as instruction rearrangement, equivalent instruction sequence replacement, branch inversion, *etc.* And the essence of interference is that malicious code can produce homogeneous code with different syntax but same semantics. Therefore, the other direction is dynamic based analysis, which generally relies on dynamic execution log to analysis the program behaviour. For instance, through analyzing the anomaly and similarity of process access behavior in data flow dependent networks, Mao *et al.* introduced an active learning method by minimizing risk estimation, which can improve the detection effect of malicious code apparently [23]. Although dynamic based method can extract the code running features, it relies on virtual running environment and also can be challenged by anti-virtual machine attacks [21].Yang *et al.* introduces a method of defect detection based on homology detection technology for open source software [28].

So, we can use more information, for instance, the function call diagram, to further detect malicious code. In this aspect, Chae proposed a software plagiarism detection system using an API labeled CFG (A-CFG) that abstracts the functionalities of a program [2]. Lim presented a method to compare CFGs by matching the basic block of a binary program, which can effectively identify the similar CFGs [12]. Wu gave a parallel method to extract the function call graph from the source code, and introduced a new software structure information comparison algorithm to effectively check the homology of the software [24].

2.2 Graph Matching Based Method

Graph matching is a classical problem in computer science. At the same time, GA also has some applications in graph matching, code similarity detection and other security areas [20]. Moon proposed a malware detection system using a hybrid GA, in which a malware is represented as a directed dependency graph and transforms the malware detection problem to the subgraph isomorphism problem [8]. Jaeun gave a multi-objective GA with a local search heuristic, comparing the degrees of each vertex of two graphs that are mapped, and counted the number of mismatched vertices [5]. Kim proposed a new cost function based on the program dependency graph using the GA to measure the similarity of the program, the method was proved to be feasible [7]. Xiang presented an improved GA, which mainly studied the isomorphism of subgraphs, and designed a special crossover function and a new fitness function to measure the evolution process [25].

In this paper, we combine static and dynamic method to generate CFG, and then use graph matching to evaluate the binary software homology. The main contributions of the paper are as follows:

- Design a special GA to evaluate the similarity of binary files, which can be used to accurately identify the isomorphism subgraph and the exact identical CFGs, which is normally regarded as a NP-complete problem. And the experimental result shows when the number of CFG nodes is large, our method can still get the CFG similarity and comparing with other classical methods, the time overhead of our algorithm is obviously reduced.
- For GA based CFG matching, we give a complete framework including CFG mapping to chromosome, operation design for the crossover, mutation, selection and fitness evolution.

3 The Proposed Method

The method proposed can be divided into two steps. The first step is CFG extraction from the binary executable files combining static and dynamic recovery method. And the second step is CFG encoded and read into GA to compute the similarities among different graphs. The flowchart of the method is shown in Figure 1, where the implementation of each step is described in detail below.

3.1 Binary Files' CFG Extraction

Although static method has high code coverage, it cannot obtain complete control flow. Although dynamic method can obtain the exact program execution information, the code coverage is low. So, the hybrid recovery method is the combination of dynamic and static method, which ensures not only the high code coverage but also can solve the indirect jump issues. And then use symbolic execution and reverse slicing techniques to further traverse the binary file to obtain complete control flow information.

3.1.1 Symbolic Execution

Symbolic execution is the symbol substitution for real values when running a program. The advantage is that we can traverse the paths of a program as much as possible by using the variable symbols. KLEE [9] and S2E [3] are two typical symbolic execution tools based on source code and binary code respectively. In symbolic execution, we can convert the program operation process into a mathematical expression. Whenever a judgment and a jump statement are encountered, symbolic execution gathers the path constraints of the current execution path into the constraint set of that path. Through constraint solver, the path reachability can be obtained by solving the constraint set. Combining the actual execution with the symbol execution, that first emerged in 2005, in DART [18] and CUTE [19], many problems encountered in traditional static symbol execution have been solved.

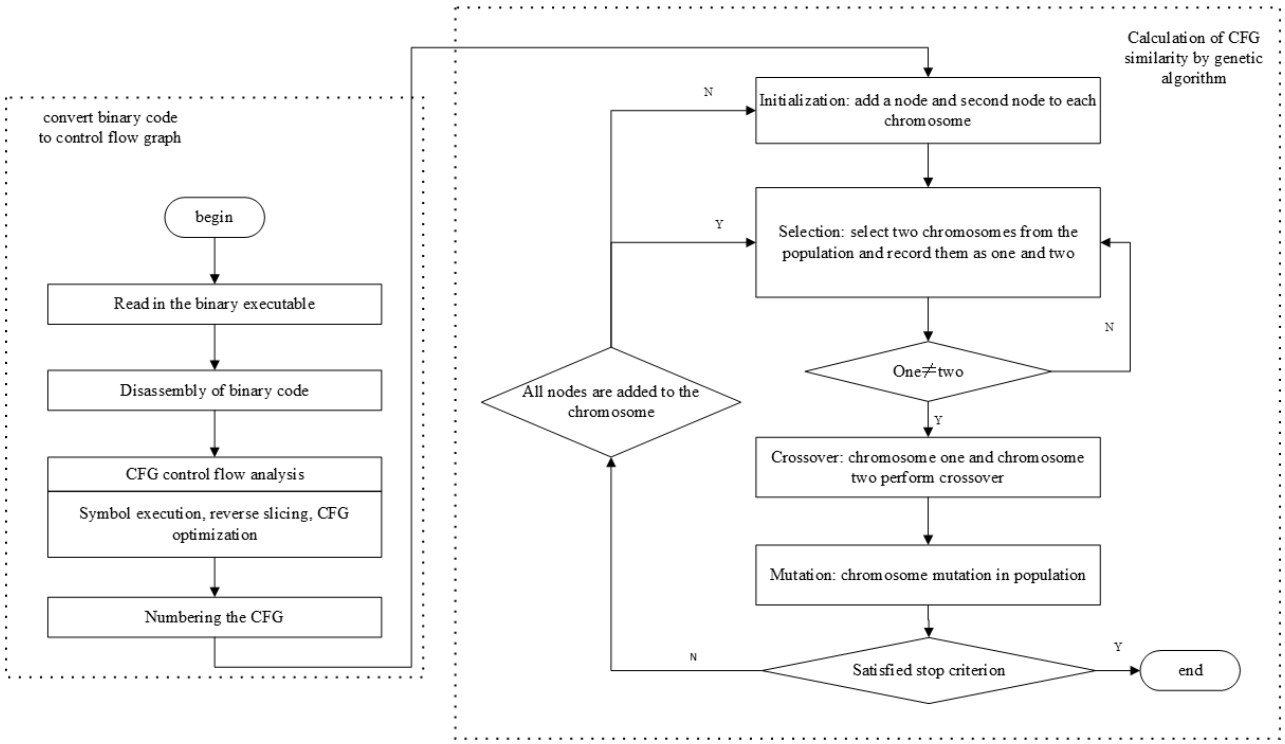

Figure 1: The flowchart of binary code homology detection

However, there are some challenges to symbolic execution, for instance, the path explosion, solver unable to solve, *etc.* For reducing the path explosion, heuristic functions, reliable program analysis and software verification techniques are some common strategies for reducing the complexity of path exploration. For constraint solving problems, there are two kinds of optimization methods. One is the elimination of uncorrelated constraints and the other is incremental solution.

3.1.2 Reverse Slicing

Program slicing is an important technique for program analysis, Negi *et al.* highlights the different test cases and comparative analysis of program slicing methods which corresponds to the applications which are usually utilized in software modification activities [15]. Given the slicing standard $< p, V >$, the forward slicing of program p contains all statements and control conditions affected by variables in V, while the backward slice of program p contains all statements and control conditions that have direct or indirect effects on variables in V.

3.1.3 CFG Generation and Optimization

Currently, the common tools for generating CFG are FXE (forced execution engine) [26], IDA Pro [17], *et al.* FXE can solve the problem of indirect branch jump by dynamically executing code, it is only suitable for the binary executable of Windows PE format under x86 architecture. IDA Pro can generate CFG and function call graph, but the repeated nodes in the graph are not merged or deleted, also for binary codes, it lacks structural information, so IDA Pro's disassembly results are far from ideal [29].

In this paper, we use a new CFG tool Angr [27], which is a binary analysis platform based on Python. It implements different symbolic execution strategies, such as veritesting [1]. The Angr system is originally designed for the DARPA Challenge. It can load different binary format files and their dependent libraries, and integrate many advanced binary analysis techniques to generate CFGs by combining dynamic and static methods. Moreover, Angr can optimize the basic block overlap in the CFG, and the overlapping parts are merged and subdivided to obtain more accurate control flow information. Meanwhile, since Angr analyses and removes the edges generated by loop structure and pseudo-return, it simplifies the CFG and alleviates the explosion problem of program state space, which improves the reconstructing ability of CFG.

3.2 GA for CFGs Similarity Calculation

Given two directed graphs, and then number the nodes in the graph [4], for example, as shown in Figure 2. We use GA to select the best individual after several generations of evolution, such as crossover and mutation, until the evolution stop criteria is satisfied.

3.2.1 Chromosome Initialization

The GA adds a node of the matching graph to each chromosome once each iteration, a corresponding node of the matched graph is generated by random function until all

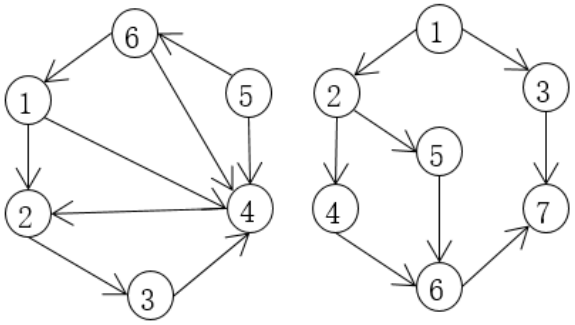

Figure 2: Two examples of numbering nodes for directed graphs

the nodes of the matching graph are added to the chromosome. According to Figure 2, the initialization process is shown in Table 1, all the left nodes in the chromosome comes from a relatively small matching graph (the left one of Figure 2) and the right nodes come from the matched graph (the right one of Figure 2). About the matching graph, adding some rules for initialization:

- When the in-degree of the left node is zero, and the out-degree of the right node is zero, they do not match. And if the out-degree of the left node is zero, and the in-degree of the right node is zero, it will not match. For example, in Figure 2, node 5 of the left graph doesn't match node 7 of the right graph, because the in-degree of node 5 is zero and the out-degree of node 7 is zero.

- The current right node number is different from the previous right node number in the chromosome, and the number does not exceed the total number of nodes in the matched graph. For example, in the first chromosome of Table 1, the node corresponding to the left node 2 must not be node 5 and the number doesn't exceed 7.

3.2.2 Chromosomes Selection

Both crossover and mutation operations in GA rely on selection function. The selection operation in this paper is similar to roulette-wheel selection. First of all, a random function produces a number between 0 and 1 denoted as a. Sum the fitness value of all chromosomes in the population as S; Next, calculate the cumulative fitness value for each chromosome, that is the sum of the fitness from the first chromosome to the current one k, marked as S_k. Set S_k as the divisor and S the dividend, the resulting quotient is recorded as b_k and compared with a, as shown in Equation (1). If $a \leqslant b_k$, returns the current chromosome number k.

For example, in Table 1, according to Equation (1), the cumulative fitness value of the first chromosome is $b_1 = \frac{f1}{f1+f2+f3+f4}$, and the cumulative fitness value of the second chromosome is $b_2 = \frac{f1+f2}{f1+f2+f3+f4}$. If $a \leqslant b_2$, returns the current chromosome number 2, and so on.

$$b_k = \frac{S_k}{S} = \frac{\sum_{i=1}^{k} f_i}{\sum_{i=1}^{L} f_i}. \tag{1}$$

Where f denotes the fitness value and $L = sizepop$, $k \leqslant sizepop$.

3.2.3 Chromosomes Crossover

Before a crossover, we generate a number between 0 and 1 through a random function, denoted as $rand$. If $rand \leqslant cross$, then do crossover operation, otherwise, keep silent.

Two chromosomes are extracted from the population by the selection function, the right nodes of the two chromosomes (noted as $Parent\_one$ and $Parent\_two$) are crossed, and the beginning position of the middle part is marked as "$begin$", the ending position of the middle part is marked as "end". $begin$ and end generated by random functions must satisfy $begin < end$, and the difference is not equal to the length of the whole chromosome. Let the middle part of the $Parent\_one$ as the middle part of the $child\_one$ after crossing, then let the nodes of the $Parent\_two$ in turn to fill in the $child\_one$, and after that do conflict detection. If there is a conflict, traversing backward in turn. After the traversal, if the nodes of the $child\_one$ chromosome is not filled up, then use the random function to generate the missing node and then perform conflict detection. According to the above crossover rules, can obtain two child chromosomes after the crossover.

For example, take the right node of the first and second chromosome in Table 1 to do a crossover and assume that $begin = 3$, $end = 5$, the specific crossover steps with the $Parent\_one$ in mind are shown in Figure 3. Thus, the chromosome produced by the crossover are $1 \to 6, 2 \to 5, 3 \to 4, 4 \to 2, 5 \to 1, 6 \to 3$.

Combine two chromosomal populations before and after the crossover and then sorted according to the fitness value in ascending order. If the population size is set to $sizepop$, then select the top $sizepop$ chromosome to form the population after crossover.

Table 1: An example of chromosome initialization

iterations #	First chromosome	Second chromosome	Third chromosome	Fourth chromosome
1	$1 \to 5$	$1 \to 6$	$1 \to 1$	$1 \to 2$
2	$1 \to 5, 2 \to 6$	$1 \to 6, 2 \to 1$	$1 \to 1, 2 \to 4$	$1 \to 2, 2 \to 3$
3	$1 \to 5, 2 \to 6$ $3 \to 4$	$1 \to 6, 2 \to 1$ $3 \to 4$	$1 \to 1, 2 \to 4$ $3 \to 2$	$1 \to 2, 2 \to 3$ $3 \to 6$
4	$1 \to 5, 2 \to 6$ $3 \to 4, 4 \to 2$	$1 \to 6, 2 \to 1$ $3 \to 4, 4 \to 2$	$1 \to 1, 2 \to 4$ $3 \to 2, 4 \to 3$	$1 \to 2, 2 \to 3$ $3 \to 6, 4 \to 4$
5	$1 \to 5, 2 \to 6$ $3 \to 4, 4 \to 2$ $5 \to 1$	$1 \to 6, 2 \to 1$ $3 \to 4, 4 \to 2$ $5 \to 5$	$1 \to 1, 2 \to 4$ $3 \to 2, 4 \to 3$ $5 \to 6$	$1 \to 2, 2 \to 3$ $3 \to 6, 4 \to 4$ $5 \to 1$
6	$1 \to 5, 2 \to 6$ $3 \to 4, 4 \to 2$ $5 \to 1, 6 \to 3$	$1 \to 6, 2 \to 1$ $3 \to 4, 4 \to 2$ $5 \to 5, 6 \to 3$	$1 \to 1, 2 \to 4$ $3 \to 2, 4 \to 3$ $5 \to 6, 6 \to 5$	$1 \to 2, 2 \to 3$ $3 \to 6, 4 \to 4$ $5 \to 1, 6 \to 5$
fitness function value	f1	f2	f3	f4

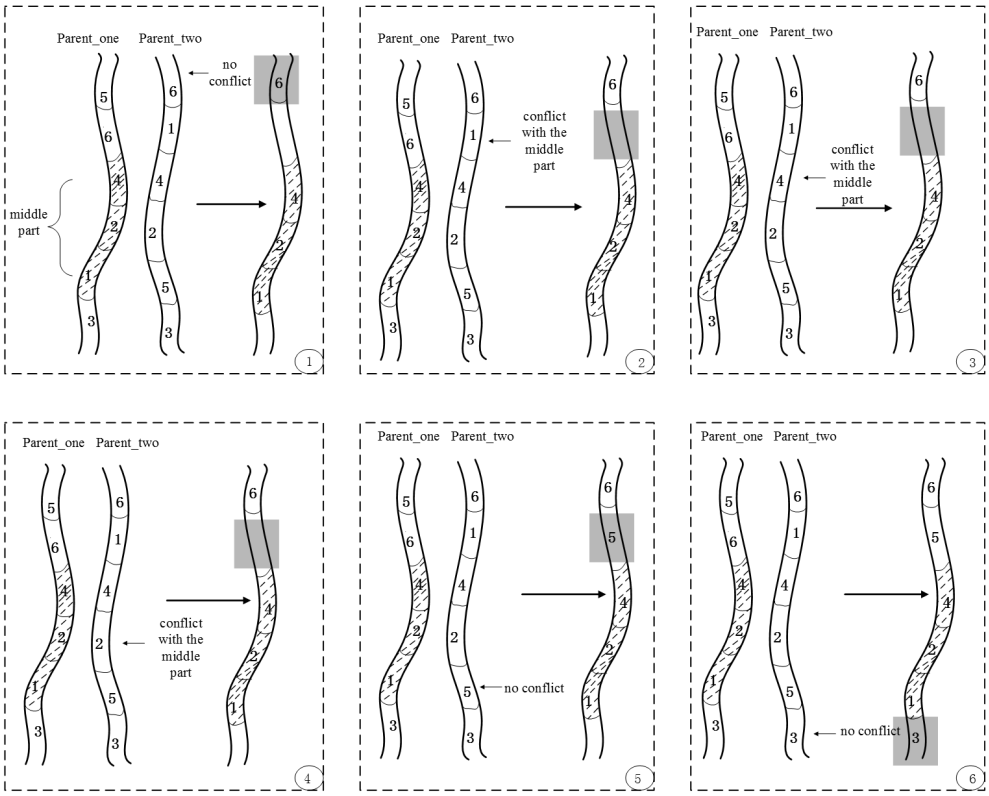

Figure 3: An example of crossover step

3.2.4 Chromosome Mutation

Each chromosome in population is selected in turn on the basis of the crossover operation. We generate a number between 0 and 1 through a random function, which is denoted as *rand*. If $rand \leqslant mutation$, the mutation operation is performed, otherwise, no mutation. When the nodes of the second graph are not all added to the chromosome, we use different random functions to generate a mutation position (*position*) and a mutation value (*num*) in the chromosome, and then do conflict detection. If there is no conflict, replace them. Otherwise, a new mutation is generated. If all the nodes of the second graph have been added to the chromosomes, use random functions to generate two different positions in the chromosome (i and j), and exchange the right nodes of the two positions as the mutated chromosomes.

For example, the third chromosome in Table 1, not all nodes of the second graph are added to the chromosome at the sixth iteration, and suppose $position = 3$, $num = 4$, it is obvious that the mutation value generated at this point conflicts with the subsequent right node, so suppose we regenerate a mutation value $num = 7$, and there is no conflict, then the chromosome mutation is shown as the left figure in Figure 4. Therefore, after the third chromosome mutation, it's $1 \to 1, 2 \to 4, 3 \to 7, 4 \to 3, 5 \to 6, 6 \to 5$. If there are 6 nodes in the second graph and all of them have been added to the chromosome, assuming $i = 2$, $j = 5$, the chromosome mutation is shown as the right figure in Figure 4. Thus, after the 3rd chromosome mutation, it's $1 \to 1, 2 \to 6, 3 \to 2, 4 \to 3, 5 \to 4, 6 \to 5$.

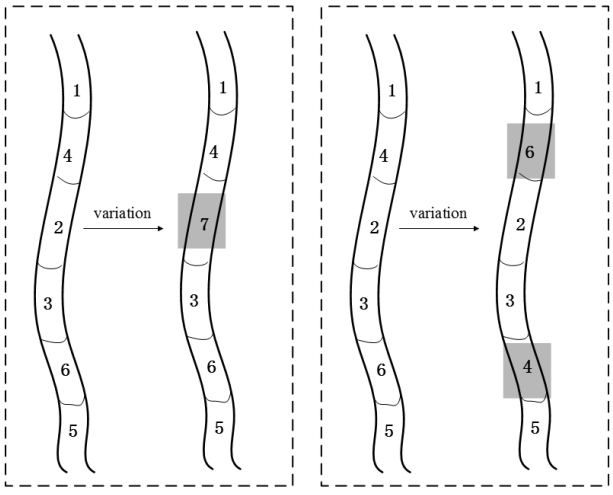

Figure 4: An example of mutation process

According to the rule of mutation, we perform the operation of chromosome variation, and we compare the fitness value of chromosomes before and after mutation. If the fitness value of the chromosome is not reduced after mutation, the mutation operation is canceled.

3.2.5 Fitness Calculation for Each Chromosome

In this paper, the fitness function F is divided into two parts $F1$, $F2$, and define $F = F1 + F2$. $F1$ is composed of the mismatched nodes number $f1$ and the mismatched edges number $f2$, that is, $F1 = f1 + f2$, where $f1$ and $f2$ are both for smaller graphs. For example, in Table 1, for the 4th chromosome, after 3 times of iteration, the results are: $1 \to 2, 2 \to 3, 3 \to 6$, at this time $f1 = 3, f2 = 2$.

However, only according to the results obtained by $F1$ is not scientific, because when two graphs are exactly the same and two graphs are isomorphic subgraphs, the results of $F1$ are all zero. Therefore, we add the $F2$ part, as shown in Equation (2), if the number of nodes in the two CFGs is equal, then $F2$ is zero, otherwise it is not zero.

$$F2 = 1 - \frac{node\_one}{node\_two}. \tag{2}$$

Where *node_one* represents the number of nodes of the smaller graph, and *node_two* the larger graph.

Therefore, the fitness function of the proposed method is shown as Equation (3).

$$F = F1 + F2 = f1 + f2 + 1 - \frac{node\_one}{node\_two}. \tag{3}$$

With this fitness function, if $F1 = 0$, $F2 = 0$, the nodes and edges of two CFGs are exactly the same. And if $F1 = 0$, $F2 \neq 0$, the two CFGs are isomorphic subgraphs.

3.2.6 Stop Criteria of Population Evolution

When any one of the followings three rules occurs, the GA will stop iteration and return the chromosome with the smallest fitness value.

- The evolution times reach the predefined number;
- During evolution, the F1 value of a chromosome equals 0;
- After several iterations, comparing with the population changes before and after each evolution, there are almost no changes in the fitness value.

4 Experiments

4.1 Experimental Data and Environment

Kernel32.dll is a very important dynamic link library file in Windows, which belongs to kernel level file. It controls the system memory management, the data input and output operation and interrupt processing. When the Windows start, the kernel32.dll resides in a specific write-protection area in memory, preventing other programs from occupying the memory area [16]. We select *loadimagefile*, *loadappinitdlls* and *basepappinitdlls* in kernel32.dll and *write.exe* as experimental data. In detail, we extract the experimental data files from Windows 7 and Windows 10, altogether 8 files. *write.exe*,

loadimagefile, *loadappinitdlls*, *basepappinitdlls* of Windows 10 system. For Windows 7 system, named them as *write_7.exe*, *loadimagefile_7*, *loadappinitdlls_7* and *basepappinitdlls_7*, and numbered them as $1, \ldots 8$, for subsequent description. All the experiments is under the Windows 10 system, and the related software include the IDA, Angr, and VS2017. In addition, for GA parameters, we set the number of population (*sizepop*) 20, the number of evolution (*maxgen*) 2000, the crossover rate (*cross*) 0.3, and the mutation rate (*mutation*) 0.8.

4.2 Comparative Experimental Methods

Among all graph matching algorithms, GED is the classical one with good fault tolerance and be suitable for various types of graphs. Although there are other graph similarity computation methods, especially the lately hot graph neural network (GNN), it needs not only massive data to train the network but also rich computational resources. Therefore, we select a series of GED algorithms as comparative ones, which can be more targeted when compared with the experimental results of the proposed method.

4.2.1 GED

The GA proposed in this paper is mainly used to calculate the similarity between two directed graphs, which is consistent with the classical GED method. Therefore, our comparative experiment selects the edit distance method and its two variant methods. GED is the sum of the minimum editing operation costs required to edit a source graph into a target graph. The cost of each step is obtained by defining the corresponding cost function [30]. In this paper, the cost of replacing, deleting and inserting the nodes and edges of the directed graph in the edit-distance method is all set to 1.

4.2.2 Edit Distance Based on Binary Linear Programming Formula

In 2015, Julien Lerouge proposed a new binary linear programming formulations for computing the exact GED between two graphs (GED_linear) [11], the advantage of which is the universality of the formula, the similarity between digraphs and undirected graphs can be calculated. But when the number of nodes is too large, the method does not work, and the time cost of this method is larger than that of other methods.

4.2.3 Edit Distance Based on Hausdorff Matching

In 2015, Andreas Fischer proposed a quadratic time approximation of GED based on Hausdorff matching (GED_distance) [6]. The advantage of the method is that the time performance is greatly improved compared with other methods. However, the disadvantage is that some accuracy is lost, that is, a small loss of precision.

4.3 Experimental Results and Analysis

4.3.1 Comparison of CFG Generated by Angr and IDA

In order to verify the integrity and simplification of the CFG generated by Angr, we compare and analyze the CFG generated by Angr and IDA. For example, the CFG of the CADET_0001 file given by the DARPA Challenge, Figure 5 is a CFG generated using Angr, and Figure 6 by IDA. The number of basic blocks in the CFG generated by Angr is more than that of IDA, and the basic block of CFG generated by Angr also contains the basic block of CFG generated by IDA. After merging overlapped nodes and eliminating cyclic and unreachable edges and nodes, Angr can obtain more accurate CFG, which can provide a more concise CFG for subsequent analysis.

Functions

Name	Address	Binary	Size	Blocks
sub_8048080	8048080	CADET_00001	273	14
sub_80481a0	80481a0	CADET_00001	435	23
sub_8048360	8048360	CADET_00001	204	12
sub_804842c	804842c	CADET_00001	11	2
sub_8048437	8048437	CADET_00001	12	1
sub_8048443	8048443	CADET_00001	2	1
sub_8048445	8048445	CADET_00001	32	2
sub_8048465	8048465	CADET_00001	32	2
sub_8048485	8048485	CADET_00001	38	2
sub_80484ab	80484ab	CADET_00001	26	2
sub_80484c5	80484c5	CADET_00001	20	2
sub_80484d9	80484d9	CADET_00001	26	2
sub_80484f3	80484f3	CADET_00001	27	1
sub_804850e	804850e	CADET_00001	34	3
_terminate	a000000	cle##kernel	0	1
transmit	a000001	cle##kernel	0	1
receive	a000002	cle##kernel	0	1
fdwait	a000003	cle##kernel	0	1
allocate	a000004	cle##kernel	0	1
deallocate	a000005	cle##kernel	0	1
random	a000006	cle##kernel	0	1

Figure 5: CFG based blocks generated by Angr

Function name	Segment	Star
sub_80	seg000	0000
sub_1A0	seg000	0000
sub_360	seg000	0000
sub_445	seg000	0000
sub_465	seg000	0000

Figure 6: CFG based block generated by IDA

4.3.2 Comparison of GA with Other Experimental Methods

First, we get the CFGs of the control group files, and the nodes and edges of the CFGs are shown in Table 2. Then take one of them and compare it with the CFGs of the control group. The experimental results are demonstrated

by similarity trend diagram, in which the solid line refers to the left Y axis and the dashed line refers to the right Y axis. In this paper, the smaller the result, the greater the similarity between the two graphs.

(2.1) Similarity between *loadimagefile* and contrast files.

CFG of the *loadimagefile* consists 8 nodes and 12 edges, and the similarity trends are shown in Figure 7. Results of GA and three comparative experimental methods show that the descending order of similarity with *loadimagefile* is $1 = 2 > 4 > 3 > 5 > 6 > 8 > 7$. However, for GED_linear and GED_distance, the edit distance between *loadimagefile* and itself is not zero, while GED and GA can get that there is no difference between *loadimagefile* and itself, and the two CFGs are exactly the same. It further shows that *loadimagefile* of kernel32.dll in Windows 10 is not modified, just the same as in Windows 7.

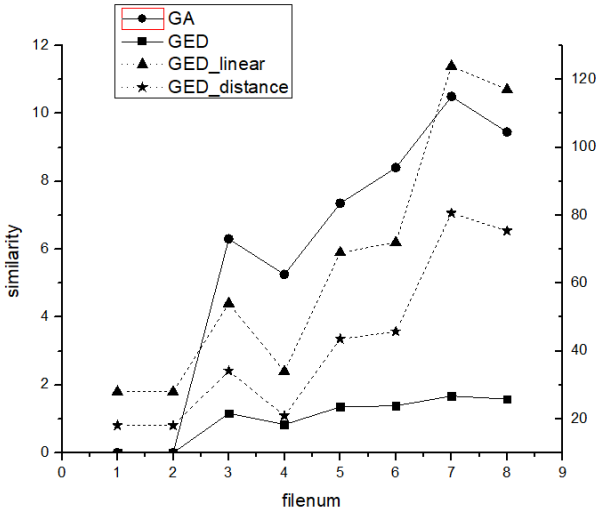

Figure 7: Similarity between *loadimagefile* and contrast files

The time performance of the four methods is shown as Figure 8. From Figure 8, it shows that the runtime of GA is relatively stable. But with the number of nodes increasing, the other three methods need more time. When the *loadimagefile* file similarity calculation with the file #8, the GA method needs the least running time, and both GED and GED_linear methods cost a long runtime. And on average, the running time of GED_linear is 9 times that of GA, GED 3 times, and GED_distance 0.6 times. But comparing with GED_distance, GA is more accurate and reasonable for identifying identical CFGs.

(2.2) Analysis of similarity between *loadappinitdlls* and contrast files.

The CFG of *loadappinitdlls* contains 17 nodes and 29 edges. As mentioned in Section 4.2, when the number of nodes is large, GED_linear cannot calculate the similarity (distance value) of two graphs. Just as in Table 3, we use NA to describe. Therefore, we mainly concern the other 3 methods and the similarity results are shown in Figure 9.

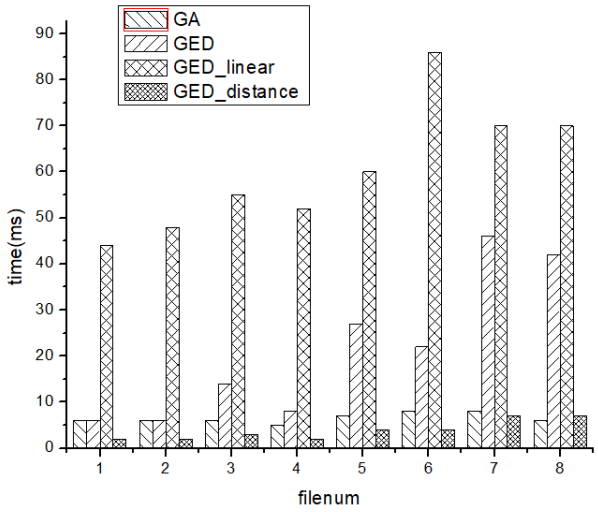

Figure 8: Time performances of four different methods for computing *loadimagefile*

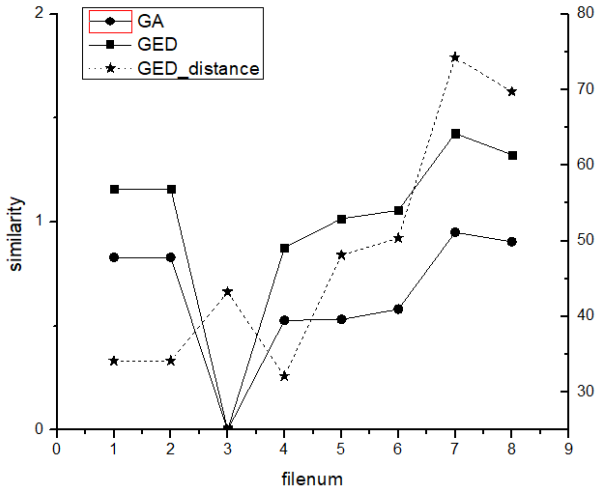

Figure 9: Similarity comparison between *loadappinitdlls* and contrast files

From Figure 9, it says that, for *loadappinitdlls*, the similarity order calculated by GED_distance is $4 > 1 = 2 > 3 > 5 > 6 > 8 > 7$, and the distance between *loadappinitdlls* and itself is not 0. When using GED, the similarity order is $3 > 4 > 5 > 6 > 1 = 2 > 8 > 7$, and *loadappinitdlls* is exactly the same as itself. For GA method, the result is consistent with the GED method.

Therefore, the results obtained, the results obtained by GED_distance and GED are inconsistent, and GA is consistent with that of GED. And when the two programs are exactly the same, the distance value should be 0, for instance, *loadappinitdlls* and file #3. Therefore, the results

Table 2: Nodes and edges of experimental files CFG

filenum(#)	1	2	3	4	5	6	7	8
Node number	8	8	17	10	24	25	48	45
Edge number	12	12	29	16	36	39	68	64

Table 3: Distance calculation by the GED_linear method

filenum (#)	filename	distance value
1	loadimagefile	46
2	Loadimagefile_7	46
3	loadappinitdlls	NA
4	Loadappinitdlls_7	50
5	basepappinitdlls	NA
6	Basepappinitdlls_7	NA
7	Write.exe	NA
8	Write_7.exe	NA

obtained by GA and GED are more reasonable. Further, the experiment shows that *loadappinitdlls* of kernel32.dll in Windows 10 is slightly modified from Windows 7.

When calculating the similarity between the *loadappinitdlls* and contrast files, the time performance of the three methods is shown in Figure 10. It can be seen that GA is less than that of both GED and GED_linear methods, and when the number of nodes is too many, the GED_linear cannot get result in a reasonable time. When calculating similarity between *loadappinitdlls* and the file #3, GA is significantly longer than that of GED_distance. And the main reason is file #3 is the *loadappinitdlls* file itself, and GA will evolve to a perfect match. When a chromosome fitness value is 0, the evolution stops. But for GED_distance, it is not 0. On average, running time for GED is 3 times that of the GA, and GED_distance 0.6 times.

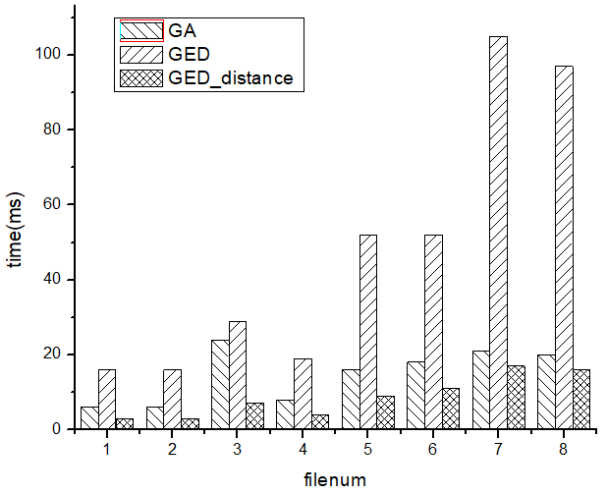

Figure 10: Time performance of 4 different methods for computing *loadappinitdlls*

(2.3) Similarity comparison between *loadappinitdlls_7* and contrast files.

The CFG of *loadappinitdlls_7* contains 10 nodes and 16 edges. The similarity diagram is as Figure 11. According to Figure 11, the results calculated by the two variations of editing distance show that the order of similarity with *loadappinitdlls_7* is $1 = 2 > 4 > 3 > 5 > 6 > 8 > 7$, and *loadappinitdlls_7* is not exactly the same as itself. For GED and GA method, the order of similarity with *loadappinitdlls_7* is $4 > 1 = 2 > 3 > 6 > 5 > 8 > 7$. So, for GA and GED, there is no difference between *loadappinitdlls_7* and itself, and two CFGs are identical.

Based on the results, the *loadappinitdlls_7* should be the most similar to itself, so the most similar file number should be 4. The results of the four methods show that the file with the least similarity to *loadappinitdlls_7* is the file #7. Therefore, it can be proved that the method proposed in this paper is reasonable to some extent.

For *loadappinitdlls_7*, the time performance of the four methods are shown in Figure 12. It can be seen that the time cost of the GED and GED_linear methods is gradually increasing with the increase of the number of nodes, but the trend of GA is almost the same as that of GED_distance. On average, the running time of GED_linear is 7 times that of GA, GED is 3 times, and GED_distance is 0.5 times.

5 Conclusions and Future work

In this paper, we proposed a genetic algorithm method to calculate the similarity between two binary files. First, the binary file is converted into CFG, and then the simi-

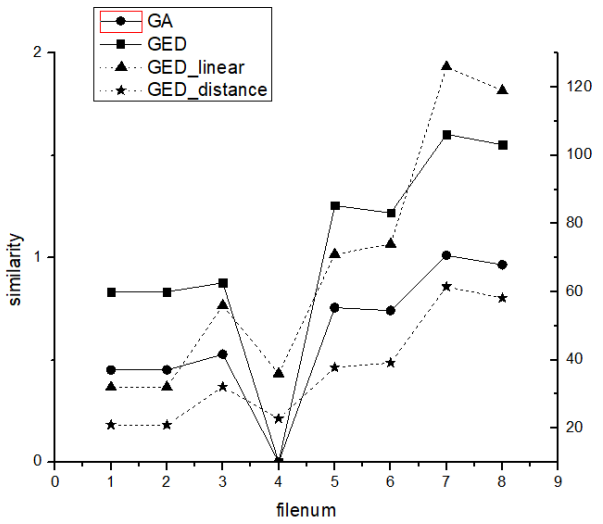

Figure 11: Similarity comparison between *loadappinitdlls_7* and contrast files

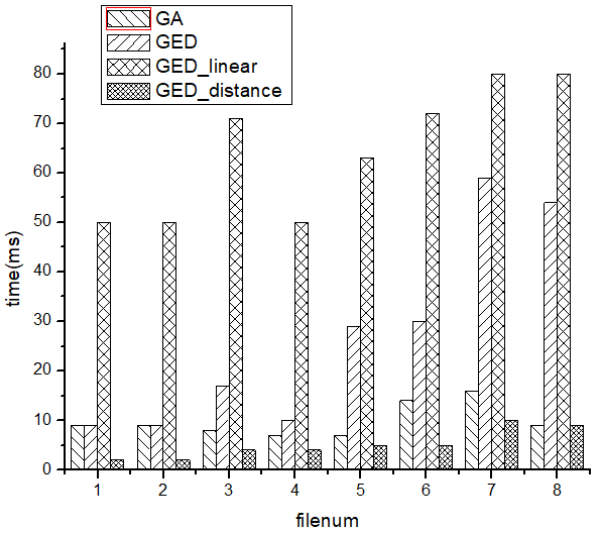

Figure 12: Time performance of 4 different methods for computing *loadappinitdlls_7*

larity between CFGs are calculated by GA. For similarity calculation, the proposed method is consistent with the GED method and can be used to accurately identify the isomorphism subgraph relationship and the exact identical CFGs, which is the basis of software homology detection.

GED is a very classical graphic similarity calculation algorithm. Compared with other methods, GED has the advantages of high accuracy and simple operation. The distance can be calculated simply by inputting 2 pairs of graphs. The similarity trend of GA in experiment (2.1) is almost the same as the three comparison algorithms, and the results of GA in experiment (2.2) and (2.3) are only the same as those of GED algorithm. Since the difference between the target file and itself should be equal to 0, we can infer that the results of GA and GED are more reasonable. Moreover, the proposed method is high efficient, especially for graphs with massive nodes. Also, according to the fitness function designed in this paper, GA can effectively identify whether the two graphs are isomorphic sub-graphs or exactly the same. About time performance, on average, GA is 0.3 times that of the GED, 0.1 times the GED_linear and 1.7 times the GED_distance.

About future work, some directions should be augmented. First of all, CFG is the analysis basis. How to enrich the CFG is a direction worth further study. After that, for GA based software homology detection, it should be reinforced in more real and wide areas applications. Meanwhile, some hyper-parameters of GA should be optimized automatically and adaptively according to different applications.

Acknowledgment

This work is partially sponsored by National Key Research and Development Program of China (2018YFB0704400, 2016YFB0700504, 2017YFB0701601), Research and Development Program in Key Areas of Guangdong Province (2018B010113001). The authors gratefully appreciate the anonymous reviewers for their valuable comments.

References

[1] T. Avgerinos, A. Rebert, K. C. Sang, and D. Brumley, "Enhancing symbolic execution with veritesting," in *International Conference on Software Engineering*, pp. 1083–1094, 2014.

[2] D. K. Chae, J. Ha, S. W. Kim, B. J. Kang, and E. G. Im, "Software plagiarism detection: A graph-based approach," in *ACM International Conference on Conference on Information Knowledge Management*, pp. 1577–1580, 2013.

[3] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," in *The Workshop on Hot Topics in System Dependability*, pp. 1286–1299, 2009.

[4] H. G. Choi, J. H. Kim, and B. R. Moon, "A hybrid incremental genetic algorithm for subgraph isomorphism problem," in *Conference on Genetic and Evolutionary Computation*, pp. 445–452, 2014.

[5] J. Choi, Y. Yoon, and B. R. Moon, "An efficient genetic algorithm for subgraph isomorphism," in *Conference on Genetic and Evolutionary Computation*, pp. 361–368, 2012.

[6] A. Fischer, C. Y. Suen, V. Frinken, K. Riesen, and H. Bunke, "Approximation of graph edit distance based on hausdorff matching," *Pattern Recognition*, vol. 48, no. 2, pp. 331–343, 2015.

[7] J. Kim, H. G. Choi, H. Yun, and B. R. Moon, "Measuring source code similarity by finding similar sub-

graph with an incremental genetic algorithm," in *Genetic and Evolutionary Computation Conference*, pp. 925–932, 2016.

[8] K. Kim and B. R. Moon, "Malware detection based on dependency graph using hybrid genetic algorithm," in *Conference on Genetic and Evolutionary Computation*, pp. 211–1218, 2010.

[9] V. Kononov and V. A. Rusakov, "On the problems of developing klee based symbolic interpreter of binary files," *Procedia Computer Science*, vol. 145, pp. 275–281, 2018.

[10] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *European Conference on Software Maintenance and Reengineering*, pp. 309–318, 2012.

[11] J. Lerouge, Z. Abu-Aisheh, R. Raveaux, P. Héroux, and S. Adam, "Graph edit distance : A new binary linear programming formulation," *Computer Science*, vol. 72, pp. 254–265, 2015.

[12] H. I. Lim, "Comparing control flow graphs of binary programs through match propagation," in *IEEE Computer Software and Applications Conference*, pp. 598–599, 2014.

[13] Mudpo, "Analysis of computer virus epidemic situation in March 2018," *Information Network Security*, no. 5, 2018.

[14] L. Nan, H. Lifang, and X. Kunfeng, "An improved software source code matching algorithm based on abstract syntax tree," *Information Network Security*, no. 1, pp. 38–42, 2014.

[15] G. Negi, E. Elias, R. Kohli, and V. Bibhu, "Reliability analysis of test cases for program slicing," in *The 1st International Conference on Innovation and Challenges in Cyber Security (ICICCS'16)*, pp. 36–40, 2016.

[16] H. Qin, *Study on Software Homology Detection Technology Based on AST Structure Optimization and CFG Comparison (in Chinese)*, Master Thesis, Beijing University of Posts and Telecommunications, 2011.

[17] Q. W. Qin, "Reverse analysis of software based on ida-Pro," *Computer Engineering*, vol. 34, no. 22, pp. 86–88, 2008.

[18] K. Sen, P. Godefroid, N. Klarlund, "DART: Directed automated random testing," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 213–223, 2005.

[19] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for C," *ACM Sigsoft Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[20] M. H. R. A. Shaikhly, H. M. El-Bakry, and A. A. Saleh, "Cloud security using markov chain and genetic algorithm," *International Journal of Electronics and Information Engineering*, vol. 8, no. 2, pp. 96–106, 2018.

[21] H. Shi, J. Mirkovic, and A. Alwabel, "Handling anti-virtual machine techniques in malicious software," *ACM Transaction on Privacy and Security*, vol. 21, no. 1, 2018.

[22] G. Tao, G. Dong, Q. Hu, and B. Cui, "Improved plagiarism detection algorithm based on abstract syntax tree," in *International Conference on Emerging Intelligent Data Web Technologies*, pp. 714–719, 2013.

[23] M. Weixuan, C. Zhongmin, and T. Li, "A malicious code detection method based on active learning," *Software Journal*, vol. 28, no. 2, pp. 384–397, 2017.

[24] P. Wu, J. Wang, and B. Tian, "Software homology detection with software motifs based on function-call graph," *IEEE Access*, no. 99, pp. 1–1, 2018.

[25] Y. Xiang, J. Han, H. Xu, and X. Guo, "An improved heuristic method for subgraph isomorphism problem," *IOP Conference Series: Materials Science and Engineering*, vol. 231, no. 1, pp. 012050, 2017.

[26] L. Xu, F. Sun, and Z. Su, "Constructing precise control flow graphs from binaries," *University of California*, 2012. (`https://pdfs.semanticscholar.org/8a80/f0d173ec7420478e4b96a8264e21e0dafac0.pdf`)

[27] S. Yan, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, and C. Kruegel, "Sok: (State of) the art of war: Offensive techniques in binary analysis," in *Security and Privacy*, pp. 138–157, 2016.

[28] J. Yang, X. Song, Y. Xiong, and Y. Meng, "An open source software defect detection technique based on homology detection and pre-identification vulnerabilitys," *Advances in Intelligent Systems and Computing*, vol. 773, pp. 932–940, 2019.

[29] Y. Zhibin, J. Xin, and S. Dawei, "A binary-oriented mixed recovery method for control flow graphs," *Computer Application Research*, no. 7, 2018.

[30] X. Zhoubo, Z. Li, Ninglihua, and A. Tianlong, "Graphic editing distance summary," *Computer Science*, vol. 45, no. 4, pp. 11–18, 2018.

Jinyue Bian is a master degree student in the school of computer science, Shanghai University. His research interests include big data analysis, machine learning, and computer and network security especially in host behaviour analysis.

Quan Qian is a full Professor in Shanghai University, China. His main research interests concerns computer network and network security, especially in cloud computing, big data analysis and wide scale distributed network environments. He received his computer science Ph.D. degree from University of Science and Technology of China (USTC) in 2003 and conducted postdoc research in USTC from 2003 to 2005. After that, he joined Shanghai University and now he is the dean of department of machine intelligence and technology, and the lab director of materials informatics and data science.