Hindawi
Security and Communication Networks
Volume 2018, Article ID 5716878, 10 pages
https://doi.org/10.1155/2018/5716878

Research Article

Security Feature Measurement for Frequent Dynamic Execution Paths in Software System

Qian Wang[1,2] **Jiadong Ren**,[1,2] **Xiaoli Yang**,[1,2] **Yongqiang Cheng**[3] **Darryl N. Davis**,[3] **and Changzhen Hu**[4]

[1]*College of Information Science and Engineering, Yanshan University, Qinhuangdao, Hebei 066000, China*
[2]*Computer Virtual Technology and System Integration Laboratory of Hebei Province, Hebei 066000, China*
[3]*Computer Science, University of Hull, Hull HU6 7RX, UK*
[4]*Beijing Key Laboratory of Software Security Engineering Technique, Beijing Institute of Technology, 5 South Zhongguancun Street, Haidian District, Beijing 100081, China*

Correspondence should be addressed to Qian Wang; wangqianysu@163.com

Received 12 October 2017; Accepted 19 February 2018; Published 22 March 2018

Academic Editor: Zheng Yan

The scale and complexity of software systems are constantly increasing, imposing new challenges for software fault location and daily maintenance. In this paper, the Security Feature measurement algorithm of Frequent dynamic execution Paths in Software, SFFPS, is proposed to provide a basis for improving the security and reliability of software. First, the dynamic execution of a complex software system is mapped onto a complex network model and sequence model. This, combined with the invocation and dependency relationships between function nodes, fault cumulative effect, and spread effect, can be analyzed. The function node security features of the software complex network are defined and measured according to the degree distribution and global step attenuation factor. Finally, frequent software execution paths are mined and weighted, and security metrics of the frequent paths are obtained and sorted. The experimental results show that SFFPS has good time performance and scalability, and the security features of the important paths in the software can be effectively measured. This study provides a guide for the research of defect propagation, software reliability, and software integration testing.

1. Introduction

The increase in complexity of software requirements makes software developers unsure of the development quality of software system; in effect the "software crisis" still has not been completely solved. How to effectively excavate the inherent characteristics of the software system structure, to recognize, measure, manage, and control the complexity of software structure, becomes a key problem for solving the development bottleneck in the software industry.

Research on the complexity of software network structure can combine the methods of complex system science and statistical physics. Depending on the granularity, software systems can be composed of different types of software entities, such as functions, classes, subroutines, packages, and artifacts. With these entities interacting with each other, software systems can achieve specific functional requirements. If the software entities are viewed as nodes and the relationship between the nodes is abstracted as edges, the software execution process presents a nonlinear network structure according to the relationship of the entities [1] and also a linear sequence structure according to the sequential characteristics of the execution order. Then, the software system can be expressed as an abstracted complex network model and a sequence model, which provides a new train of thought [2] for the description of the software system.

The root cause of the security danger hidden in software lies in the vulnerability of the entity itself. The vulnerability is the measurement of the potential danger of a software entity to be used as an attack and can be discussed from the perspective of computer network [3, 4] or software static code analysis, but the integrity (whole structure) and the dynamic execution (behavior characteristic) of software system are ignored. In addition, the degree to which software system

security is threatened depends not only on the severity of the fault, but also on the fault propagation capacity of the entity. If one or more functions fail, the fault may be propagated to other functions by invocation relationships and further lead to a part of or the whole software system crashing, known as "cascading failure" [5]. Therefore, the software security feature measurement should take into account the vulnerability and propagation of software entities.

How to quantitatively measure the security features of nodes from the software complex network is the premise and basis for further analysis of the software behavior trajectory path. At present, there are lots of methods for discovering the important nodes in complex networks. The classic methods based on centricity contain degree centrality [6], closeness centrality [7], betweenness centrality [8], eigenvector centrality [9], subgraph centricity [10], and so on. The classic methods based on random walk model include PageRank [11], LeaderRank [12], and their improved algorithm NodeRank [13]. Wang and Lü [14] by means of the influence node mining method prove that the defect propagation capacity of a node is stronger if the in-degree and out-degree of the node are bigger. Huang et al. [15] based on the invocation and dependency relationships between functions with the fault probability of nodes calculate the fault accumulation degree of upper nodes by the iteration from the leaf nodes. These methods attempt to describe the relevance of software node importance to fault generation and propagation, but fail to form a measurement of software security.

Sequence or path is the most basic and important way for the description of dynamic software execution process. The full execution path of the whole software can reflect the occurrence order and frequency of the software internal entities. However, the method of path extraction and mining is restricted by the nested, circulatory, iteration and the continuous invocation relationships of entities. Most software path mining algorithms are extracted on the basis of complex networks. For example, Tang et al. [16] propose an algorithm for shortest path mining between any two vertices in complex network. Zhang et al. [17] minimize the length of the extracted path and reduce the unnecessary time overhead by further processing the repetitive structure. The GP method proposed by Nguyen et al. [18] can automatically detect and fix software vulnerabilities according to the software execution path. Murtaza et al. [19] predict future software possible defects by analyzing the historical vulnerability sequence data with characteristics of Markov to provide adequate response time. Zou et al. [20] analyze the reliability of Digital Instrumentation and Control software system based on the flow network model by finding sensitive paths in the complexity software. These algorithms are based on the network to extract path, which can lead to the phenomenon of repeated reading and approximate connection; also, these software security analyses cannot work without existing vulnerability information or real faults as their training data.

In this paper, the Security Feature measurement algorithm of Frequent dynamic execution Paths in Software, SFFPS, is proposed. A complex network model and a sequence model are formed based on software dynamic execution behavior. It is for early security feature measurement, before there are real vulnerabilities or faults generated, which can provide the premise for the software quality and reliability evaluation. The main contributions are as follows.

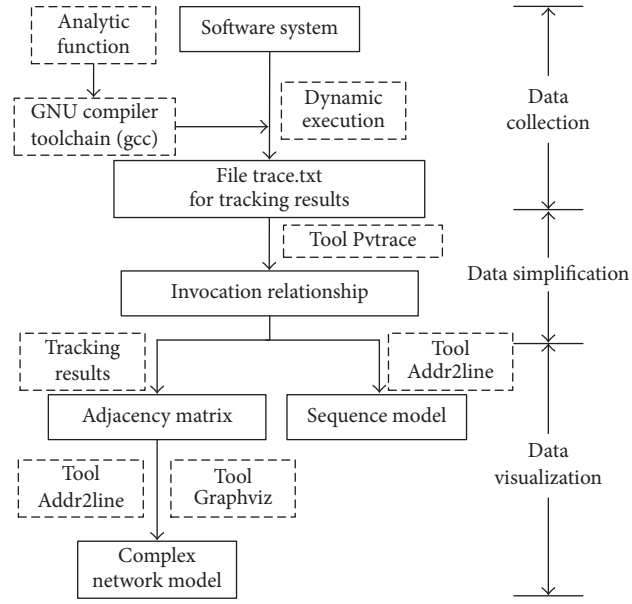


Figure 1: Theory process of model construction.

(1) The software system is mapped to a complex network model and sequence model, from the nonlinear perspective to effectively express the characterization of complex correlation between software entities and from the linear perspective to capture sequential characteristics of the dynamic execution.

(2) The behavior nature of fault accumulation and propagation is analyzed based on the system structure of software dynamic execution and standard measurement of security features (vulnerability and propagation) being defined.

(3) Frequent paths in software dynamic execution are mined and weighted by the node security features. The key paths which are worthy of attention are ensured by both their frequency and security features.

The remainder of the paper is organized as follows. Section 2 gives the model construction. Sections 3 and 4 develop the definition of the security features and the SFFPS algorithm. Section 5 provides some examples. Section 6 presents the performance study of SFFPS and shows the rank of the important paths. Section 7 contains the concluding remarks.

2. Constructions of Complex Network Model and Sequence Model

The dynamic execution trace of software systems contains three phases, which are data collection, tracking data simplification, and data visualization as shown in Figure 1. The modeling process of simple functions is shown in Figure 2.

Phase 1. Match the entry and exit configuration functions of the GNU compiler toolchain (gcc), and insert the analysis function into the entry and exit of the application functions

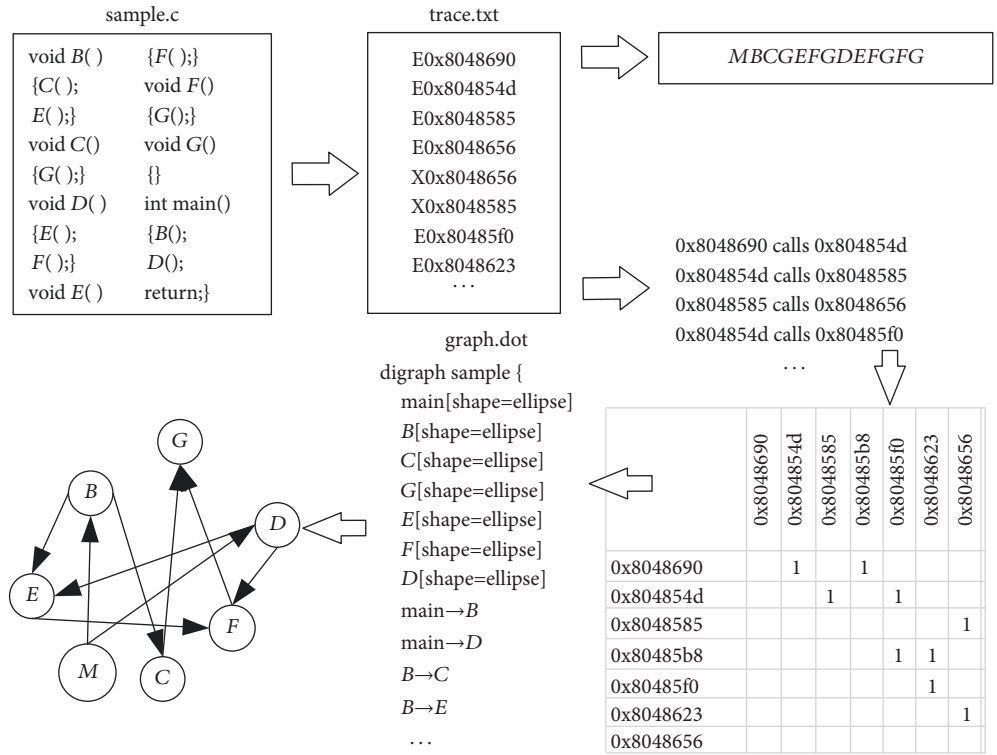


Figure 2: Modeling process of invocation relationship between simple functions.

to trace the function execution process. The tracking results are recorded in the file trace.txt.

Phase 2. The letters "E" and "X" before the tracking addresses represent the entry and exit of a function, respectively. A simplification tool Pvtrace is used to analyze the function invocation according to the letters "E" and "X." An address transformation tool Addr2line is used and the address is transformed to function name.

Phase 3. Map the function invocation order to sequence model and a visualization tool Graphviz is used to form the complex network, which defines the global relationship between all the functions.

According to Figure 2, the corresponding relationships of function address and function name are as follows:

0x8048690 → M(main); 0x804854d → B; 0x8048585 → C; 0x80485b8 → D.

0x80485f0 → E; 0x8048623 → F, 0x8048656 → G.

Only the addresses with the letter "E" are used for sequence model construction.

3. The Security Feature Definition and Measurement of Function Nodes

The security feature measurement of a function node is based on the software structure; the analysis of vulnerability and propagation is according to cumulative effect and the spread effect caused by the mechanism of fault production and propagation. The global accessibility and fault tolerance with step attenuation effect are fully considered, so the node security features are calculated according to the degree distribution and step attenuation factor.

Definition 4 (software complex network). In a software complex network, functions are defined as the nodes; the invocation relationships between functions are defined as edges.

Definition 5 (vulnerability). Vulnerability of a function node is the characteristic that a function node may break down because of the effect of its invocated fault node through invocation relationship.

Typically, if a node invocates more other nodes, it is more functional and vulnerable. That is to say, it is more likely to be affected and be faulted. The calculation of V (vulnerability) is as follows:

$$V(u) = \text{outDegree}(u) + \sum_{w \in \text{OS}(u)} \theta * V(w), \quad (1)$$

where u, w represent function nodes, $V(u)$ represents the vulnerability of node u, OutDegree (u) represents the out-degree of node u, θ represents the step attenuation factor, which satisfies $\theta \in (0, 1)$, and OS(u) represents the direct out-neighbor set of node u.

Definition 6 (propagation). Propagation of a function node is the characteristic that a function node may propagate its

```
Input: Complex network CN, step attenuation factor θ
Output: Node list with security features NFlist
for each node u in CN
  {  V(u) = calculation_V(u);
     P(u) = calculation_P(u);
     NFlist.add (u, V(u), P(u)); }
Procedure calculation_V(u)
{    V(u) = outDegree(u);
     For each node w ∈ OS(u)
          V(u) += calculation_V(w);
     return V(u); }
Procedure calculation_P(u)
{    P(u) = inDegree(u);
     for each node w ∈ IS(u)
          P(u) += calculation_P(w);
     return P(u); }
```

Algorithm 1: Calculation of node security features.

fault to the nodes by which it is invocated. The calculation of P (propagation) is as follows:

$$P(u) = \text{inDegree}(u) + \sum_{w \in \text{IS}(u)} \theta * P(w), \tag{2}$$

where $P(u)$ represents the propagation capacity of node u, inDegree (u) represents the in-degree of node u, and IS(u) represents the direct in-neighbor set of node u.

Algorithm 1 describes the calculation process of vulnerability and propagation.

4. Mining Frequent Paths from Dynamic Execution with Security Feature Measurement

The importance of a software dynamic execution path takes into account two aspects: one is the occurrence frequency of the path and the other one is the security feature coming from the nonrepetitive nodes contained in the path. These two aspects are complementary. For example, if there are lots of loop bodies in the software execution, loop body and its subset are always frequent. But because most of its contained nodes are the same, the fault influence range is small. Similarly, if a path contains many different nodes with a lower occurrence frequency, its impact range is large, but its occurrence possibility is small. That is to say, if the frequency of a path is very high and the path contains more nonrepetitive nodes, the path is worthy of more attention.

4.1. Relative Definitions of Frequent Path. Let $F = \{f_1, f_2, f_3, \ldots, f_n\}$ be a set of function symbols. S is a software execution path, and it is composed of function symbols with time-ordered occurrence. Minimal support count (mincount) can be calculated by mincount = minsup $*$ $|S|$, where minsup is a given threshold and $|S|$ is the number of function symbols in S. If there are k symbols in S, S is a k-path.

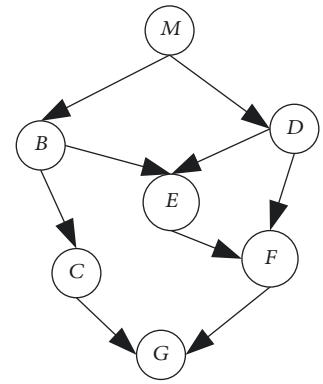


Figure 3: Complex network model of simple function invocation relationship.

Definition 7 (subpath and superpath). A path $S_1 = \langle a_1, a_2, \ldots, a_m \rangle$ is a subpath of another path $S_2 = \langle b_1, b_2, \ldots, b_n \rangle$, denoted as $S_1 \subseteq S_2$, if there are numbers $i_1, i_2, \ldots, i_m$, such that $1 \le i_1 < i_2 < \cdots < i_m \le n$ and $a_1 \subseteq b_{i1}, a_2 \subseteq b_{i2}, \ldots, a_m \subseteq b_{im}$. It can also be said that S_2 is a superpath of path S_1.

Definition 8 (support number). S is a path; the support number of S, denoted as sup(S), is defined as its occurrence number in the software execution.

Property 9 (frequent path). A path S is frequent if its support number sup(S) is equal to or more than mincount.

Property 10 (antimonotone). If path A is not a frequent path, any path B containing A, which is a superpath of A, cannot be a frequent path.

4.2. Weighting the Frequent Path Based on the Security Features of Function Node. SFFPS algorithm is for mining the security features of frequent paths based on the dynamic execution sequence model and the node security features in the complex network model. It contains two phases: one is frequent path mining and the other one is security feature weighting. First, the function nodes in the sequence model are read to form the function position set. Then, the position index is used for pattern growth; this self-growth strategy can avoid candidate generation and ensure the continuity of function execution. Finally, path frequency is validated by minimum support count mincount, and path is weighted according to the security feature of the nonrepetitive nodes contained in it. The security features of the frequent paths are measured. Algorithm 2 describes the mining and weighting process.

5. An Illustrative Example

The complex network in Figure 2 is a variant of the tree-like structure in Figure 3, which is redrawn for easier understanding.

Without losing generality, the coordination factor is set to 0.5. Security features of each node are calculated as follows.

Input: Function execution path S, minimal support threshold minsup
Output: Path list with security features Slist
mincount = minsup $*$ $|S|$;
for each node S_i in S
{ Pos(S_i).add(S_i.pos); }
for each Pos(S_i)
{ sup(S_i) = |Pos(S_i)|;
if(sup(S_i) < mincount)
Delete Pos(S_i);
else
$L_1 = L_1$.add(S_i, sup(S_i)); }
for (k = 2; L_{k-1}! = ∅; k++)
{ gen_mine(L_{k-1});
for each $l_m \in L$
{ for each different function symbol u in l_m
{$V(l_m)$+=$V(u)$;
$P(l_m)$+=$P(u)$; } } }
Sort each $l_m \in L$ by $V(l_m)$, $P(l_m)$ and form Slist;
Procedure gen_mine (L_{k-1})
{ for each $l_i \in L_{k-1}$
{ for each position pos in Pos(l_i)
{ $S_j = S[pos + 1]$;
for each position pos in Pos(l_i)
{ if (pos+1 exists in Pos(S_j))
{ Pos(l_iS_j).add(pos + 1); } } }
sup(l_iS_j) = |Pos(l_iS_j)|;
if (sup(l_iS_j) < mincount)
delete Pos(l_iS_j);
else
$L_k = L_k$.add(l_iS_j); } }

Algorithm 2: Security feature measurement of frequent paths in software.

As the "main" function is special (vulnerability is always large and propagation is 0), it is excluded for measurement.

Vulnerability

$$V(G) = \text{outDegree}(G) = 0.$$

$$V(C) = \text{outDegree}(C) + \theta * V(G) = 1 + 0.5 * 0 = 1.$$

$$V(F) = \text{outDegree}(F) + \theta * V(G) = 1 + 0.5 * 0 = 1.$$

$$V(E) = \text{outDegree}(E) + \theta * V(F) = 1 + 0.5 * 1 = 1.5.$$

$$V(D) = \text{outDegree}(D) + \theta * \{V(E) + V(F)\} = 2 + 0.5 * (1.5 + 1) = 3.25.$$

$$V(B) = \text{outDegree}(B) + \theta * \{V(C) + V(E)\} = 2 + 0.5 * (1 + 1.5) = 3.25.$$

Propagation

$$P(B) = \text{inDegree}(B) = 1;\ P(D) = \text{inDegree}(D) = 1.$$

$$P(C) = \text{inDegree}(C) + \theta * P(B) = 1 + 0.5 * 1 = 1.5.$$

$$P(E) = \text{inDegree}(E) + \theta * \{P(B) + P(D)\} = 2 + 0.5 * (1 + 1) = 3.$$

$$P(F) = \text{inDegree}(F) + \theta * \{P(E) + P(D)\} = 2 + 0.5 * (3 + 1) = 4.$$

$$P(G) = \text{inDegree}(G) + \theta * \{P(C) + P(F)\} = 2 + 0.5 * (1.5 + 4) = 4.75.$$

According to the sequence model of the example, $S = (M)BCGEFGDEFGFG$, if the minsup is set to 0.15, mincount = $0.15 * 12 \approx 2$.

Pos(B) = {1}; Pos(C) = {2}; Pos(D) = {7}; Pos(E) = {4, 8}.

Pos(F) = {5, 9, 11}; Pos(G) = {3, 6, 10, 12}.

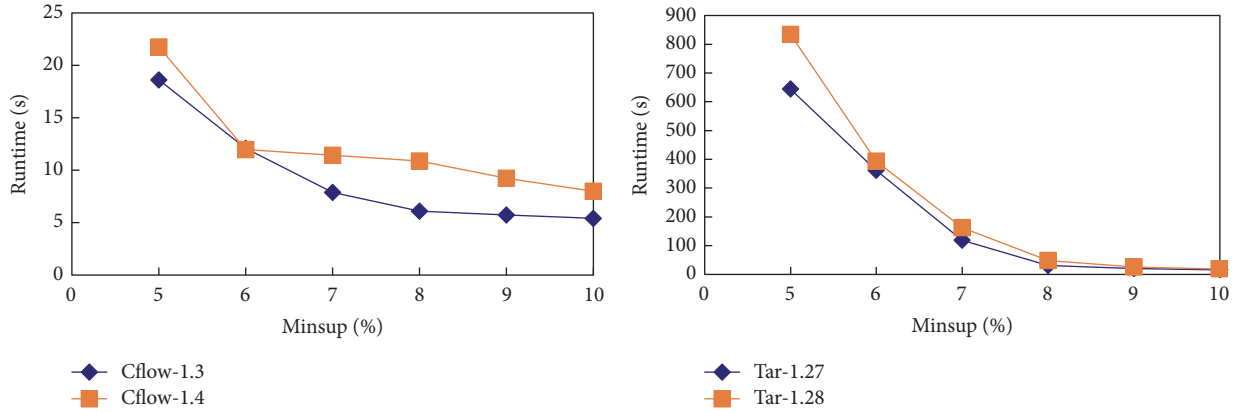
Frequent 1-Path

E, sup(E) = 2, Pos(E) = {4, 8}.

F, sup(F) = 3, Pos(F) = {5, 9, 11}.

G, sup(G) = 4, Pos(G) = {3, 6, 10, 12}.

The mining method of frequent 2-path is based on the position set of the frequent 1-path by using the adjacent position value as index to find the extended paths. For example, the position set of node E is {4, 8}, and its extended position set is {5, 9}. The function nodes in positions 5 and 9 both correspond to node F. So, Pos (EF) = {5, 9} is obtained, sup(EF) = 2, and path EF is a frequent 2-path.



(a) Runtime test of SFFPS in Cflow

(b) Runtime test of SFFPS in Tar

Figure 4: Runtime test of SFFPS with different support thresholds.

Table 1: The security features of frequent paths.

Frequent paths	sup	V	P
E	2	1.5	3
F	3	1	3
G	4	0	4.75
EF	2	2.5	7
FG	4	1	8.75

Frequent 2-Path

$$EF, \text{sup}(EF) = 2, \text{Pos}(EF) = \{5, 9\}; \text{FG}, \text{sup}(FG) = 3. \text{Pos}(FG) = \{6, 10, 12\}.$$

The security features of frequent 1-path included in the function nodes are calculated as before, and the security features of frequent 2-path are calculated as follows. Table 1 shows the security features of all the frequent paths.

$$\begin{aligned} V(EF) &= V(E) + V(F) = 1.5 + 1 = 2.5; \\ P(EF) &= P(E) + P(F) = 3 + 4 = 7. \end{aligned} \tag{3}$$

6. Experimental Results

Experiments are performed on a PC with Intel® Core™ 3.6 GHz CPU and 16 G main memory, running on Windows 8. We evaluate the runtime and scalability of the algorithm SFFPS and calculate the fault feature ranks of nodes and important paths. To test the algorithms in the same coding environment, all the programs are written in Java using MyEclipse. Datasets used in the experiment are open-source software programs of Cflow and Tar obtained from open-source software library (https://sourceforge.net).

6.1. Runtime and Scalability Tests of SFFPS. By testing the runtime and scalability of SFFPS, two newest versions of each Cflow and Tar are selected. The support threshold is from 0.005 to 0.01 for runtime test, and the upper threshold 0.01 is used for scalability test. The total runtime is composed of three parts, node fault feature calculation, frequent pattern mining, and weight appending. Figure 4 is the runtime test of SFFPS with different support thresholds and Figure 5 is the scalability test with different length percentages of the sequence when the support threshold is set to 0.01.

From Figure 4, SFFPS performs well in the support threshold range [0.005, 0.010]. This is due to the adjacency table which is for the storage of the complex network model. The calculation of the out-degree and in-degree of the nodes is made easier, which improves the calculation of node security feature. Furthermore, as the sequence model is based on the start order of each function, the detailed invocation and end time of a node are ignored, and the length of the sequence model is simplified. Also, position value index is used for the mining and pattern growth of the paths, which avoids candidate generation, and index methods are always effective. Finally, the weight appending process achieves efficiency because fewer nodes are involved by the strategy of nonrepetition.

From Figure 5, SFFPS shows good scalability on the software Cflow. With the increase of the length of the sequence, the execution time of SFFPS is essentially a linear growth. From the experimental data, the number of frequent sequences is also increasing. This indicates that the functions of Cflow are uniformly distributed. However, the time overhead of software Tar is quite expensive around 40% of sequence length; the number of frequent sequences increases rapidly from 194 when the percentage is 20% to 1123. After that, the time overhead and the number of frequent sequences reduces. This indicates that there are more core functions in software Tar and there are more invocations of core functions in the early stage of the program.

6.2. The Security Features of the Function Nodes. Tables 2 and 3 show the security feature rank and value of the function nodes in the newest versions of Cflow and Tar.

From Tables 2 and 3, the security features of the same function nodes are relatively stable for different versions of the same software. So, in the process of version evolution, it can be inferred and predicted that the same function should

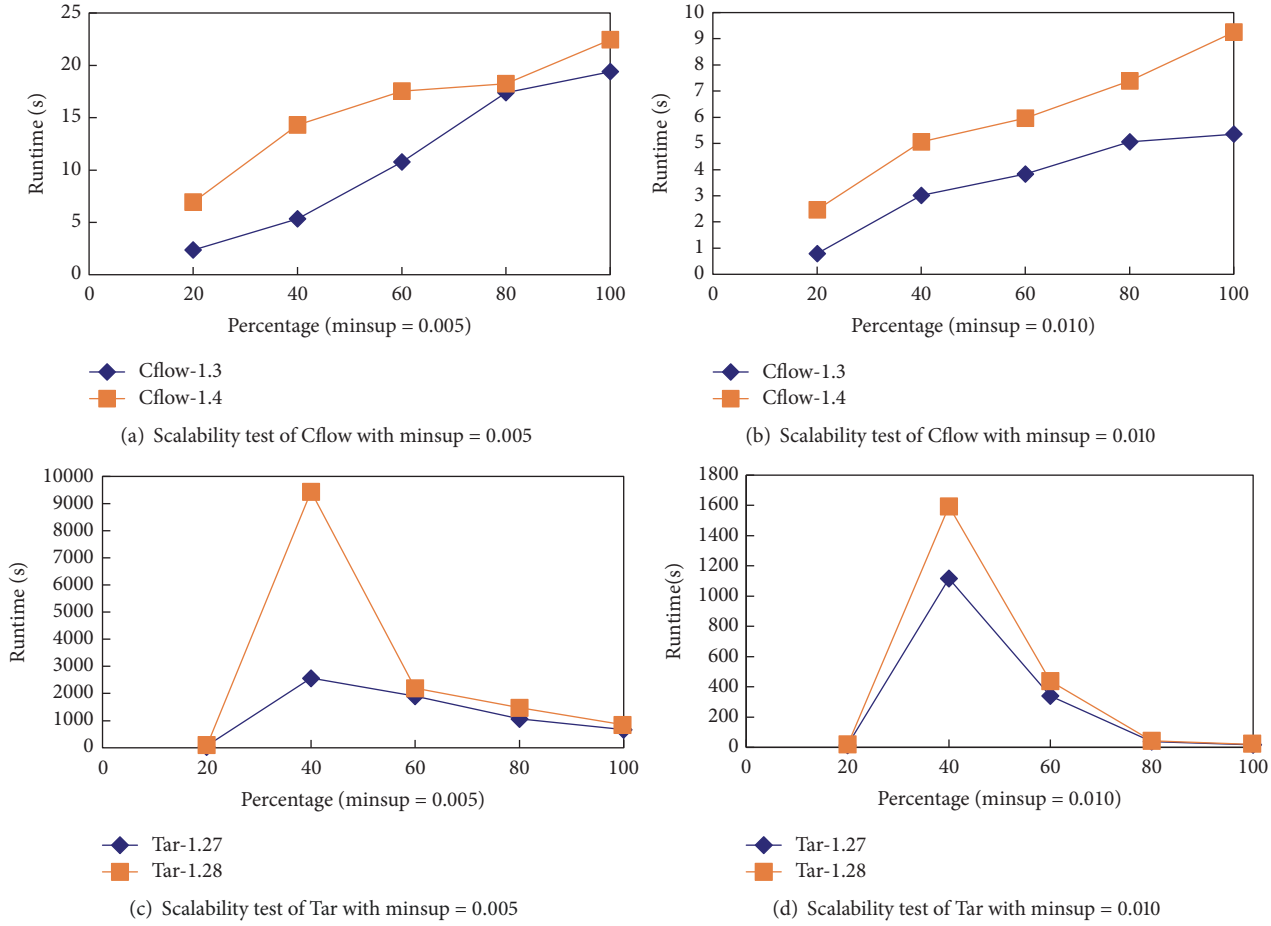


Figure 5: Scalability test of SFFPS with different percentages of sequence length.

Table 2: Rank and value of function node security features in Cflow.

Vulnerability			Propagation		
Function name	Cflow-1.3 (rank/value)	Cflow-1.4 (rank/value)	Function name	Cflow-1.3 (rank/value)	Cflow-1.4 (rank/value)
parse_variable_declaration	1/45.05	1/47.07	nexttoken	1/33.73	1/36.17
parse_declaration	2/40.77	2/41.85	tokpush	2/19.74	3/20.96
yyparse	3/37.50	4/38.82	putback	3/19.32	4/19.32
parse_typedef	4/19.33	6/20.78	get_token	4/17.86	5/19.08
tree_output	5/18.50	5/25.56	linked_list_append	5/15.47	2/21.06
func_body	6/18.15	7/18.28	lookup	6/13.79	6/13.94
parse_function_declaration	7/17.80	8/17.99	hash_symbol_hasher	7/12.90	8/13.04
parse_dcl	8/16.92	9/17.41	hash_symbol_compare	7/12.90	8/13.04
expression	9/16.17	10/16.42	yy_load_buffer_state	8/12.86	7/13.27
initializer_list	10/12.53	13/12.72	yylex	9/9.93	11/10.54

have approximate rank in a new software version. Also, the function rank in the old version can be used as a basis for the version upgrade process with function nodes remove, merger, or update. The nodes with larger rank changes should be given more attention.

Tables 4 and 5 show the frequent paths of Cflow-1.4 in the top 10 security feature ranks of vulnerability and propagation.

There are double meanings of the paths listed in Tables 4 and 5. One is that the paths are frequent, which first affirms that the occurrence possibility of the path is relatively large. The other one is that the security feature values of the paths are larger, which evaluates the security risk of the path. Only when both of them work together can we make a persuasive security measurement.

Table 3: Rank and value of function node security features in Tar.

Vulnerability			Propagation		
Function name	Tar-1.27 (rank/value)	Tar-1.28 (rank/value)	Function name	Tar-1.27 (rank/value)	Tar-1.28 (rank/value)
dump_file0	1/42.84	1/44.09	to_chars	1/17.28	1/17.28
create_archive	2/32.02	2/33.21	assign_string	2/11.67	2/11.67
dump_file	3/25.92	4/27.04	to_octal	3/9.64	3/9.64
dump_regular_file	4/19.37	5/19.37	tar_copy_str	4/7.13	4/7.13
dump_hard_link	5/17.37	6/17.37	set_next_block_after	5/6.64	5/6.64
start_header	6/15.62	8/15.62	find_next_block	6/6.16	6/6.16
dump_dir0	7/15.37	7/16.37	start_header	7/5.84	7/5.84
dump_dir	8/10.68	9/11.18	finish_header	7/5.84	7/5.84
close_archive	9/7.50	11/8.00	current_block_ordinal	7/5.84	7/5.84
_open_archive	10/7.25	10/8.75	flush_archive	8/5.83	8/5.83

Table 4: Vulnerability rank and value of frequent paths (minsup = 0.01).

Paths	Rank/value
is_printable, include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler, set_active	1/16.75
is_printable, include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline	1/16.75
is_printable, include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler	1/16.75
direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline	2/15.75
include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler, set_active	2/15.75
include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler	2/15.75
include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline	2/15.75
direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler	2/15.75
direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler, set_active	2/15.75
is_printable, include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol	3/14.75

In addition, the frequency of the path can be used to predict the function nodes that are going to be affected, and the security features of the path can be used to evaluate the possible impact scale of the abnormal path. For example, the main consideration of random fault detection is the vulnerability. According to the first path of Table 4, from the perspective of frequency, if the first three functions of a fault path are "is_printable," "include_symbol," and "direct_tree," then the next functions which are likely to be affected are "include_symbol," "print_symbol," "gnu_output_handler," and so on. From the perspective of security features, the path displays higher rank and value in vulnerability, which indicates the fault location is relatively accurate. If it is a hostile attack detection, the attacker expects a wider range effect, so the propagation should be considered more. In this case, the analysis method is similar.

7. Conclusion

In this paper, a novel algorithm, SFFPS, is proposed to define and measure the security feature of dynamic execution path in software. Complex network model and sequence model are constructed for the record of invocation relationship and function execution order. The node degree in the complex network is used for security feature analysis from a structural perspective before real fault occurrence. The paths extracted from the sequence model are used for frequency test and weighted by the node security features. Finally, frequent dynamic execution paths with top security feature rank are mined as important paths which should be of greater concern. With the experiment, SFFPS can effectively mine the important paths from the newest versions of software programs Cflow and Tar. SFFPS can be applied as a basis for software evolution, a tool for software internal structure

Table 5: Propagation rank and value of frequent paths (minsup = 0.01).

Paths	Rank/value
is_printable, include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler, set_active	1/36.35
is_printable, include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline	2/34.48
is_printable, include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler	2/34.48
include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler, set_active	3/34.42
direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler, set_active	3/34.42
include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler, set_active	4/32.67
is_printable, include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name	5/32.60
direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline	6/32.54
include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline, gnu_output_handler	6/32.54
include_symbol, direct_tree, include_symbol, print_symbol, gnu_output_handler, print_symbol, print_level, print_function_name, newline	6/32.54

analysis, and a guidance to fault location and attack detection, which are helpful for software quality assurance.

Conflicts of Interest

There are no conflicts of interest related to this paper.

Acknowledgments

This work is supported by the National Key R&D Program of China (2016YFB0800700), the National Natural Science Foundation of China under Grant nos. 61472341, 61772449, and 61572420, the Natural Science Foundation of Hebei Province, China, under Grant nos. F2016203330 and F2015203326, the Advanced Program of Postdoctoral Scientific Research under Grant no. B2017003005, and the Doctoral Foundation of Yanshan University under Grant no. B1036.

References

[1] Y.-T. Ma, K.-Q. He, B. Li, and J. Liu, "Empirical study on the characteristics of complex networks in networked software," *Ruan Jian Xue Bao/Journal of Software*, vol. 22, no. 3, pp. 381–407, 2011.

[2] X. Wang and W. Yichen, *Research Progress on Error Propagation Model in Software System[J]*, vol. 43, Computer Science, 2016.

[3] H. L. Vu, K. K. Khaw, T. Y. Chen, and F.-C. Kuo, "A new approach for network vulnerability analysis," *IEEE Conference on Local Computer Networks*, vol. 58, no. 4, pp. 200–206, 2008.

[4] X. J. Qin, L. Zhou, Z. N. Chen, and S. . Gan, "Software vulnerable trace's solving algorithm based on lazy symbolic execution," *Chinese Journal of Computers. Jisuanji Xuebao*, vol. 38, no. 11, pp. 2290–2300, 2015.

[5] J. Wang, Y.-H. Liu, and X.-L. Liu, "Model for cascading faults in complex software," *Jisuanji Xuebao/Chinese Journal of Computers*, vol. 34, no. 6, pp. 1137–1147, 2011.

[6] D. Wei, Y. Li, Y. Zhang, and Y. Deng, "Degree centrality based on the weighted network," in *Proceedings of the 2012 24th Chinese Control and Decision Conference, CCDC 2012*, pp. 3976–3979, China, May 2012.

[7] K. Okamoto, W. Chen, and Y. Li X, "Ranking of Closeness Centrality for Large-Scale Social Networks," in *International Workshop on Frontiers in Algorithmics*, pp. 186–195, Springer-Verlag, 2008.

[8] M. Kitsak, S. Havlin, G. Paul, M. Riccaboni, F. Pammolli, and H. E. Stanley, "Betweenness centrality of fractal and nonfractal scale-free model networks and tests on real networks," *Physical Review E: Statistical, Nonlinear, and Soft Matter Physics*, vol. 75, no. 5, Article ID 056115, 2007.

[9] X. Wu, M. Zhang, and Y. Han, "Research on centrality of node importance in scale-free complex networks," in *Proceedings of the 31st Chinese Control Conference, CCC 2012*, pp. 1073–1077, chn, July 2012.

[10] X. Yan, C. Li, L. Zhang, and Y. Hu, "A new method optimizing the subgraph centrality of large networks," *Physica A: Statistical Mechanics and its Applications*, vol. 444, pp. 373–387, 2016.

[11] L. Page, "The PageRank citation ranking: Bringing order to the web," *Stanford Digital Libraries Working Paper*, vol. 9, no. 1, pp. 1–14, 1998.

[12] S. Xu and P. Wang, "Identifying important nodes by adaptive LeaderRank," *Physica A: Statistical Mechanics and its Applications*, vol. 469, pp. 654–664, 2017.

[13] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 419–429, IEEE, Zürich, Switzerland, June 2012.

[14] B.-Y. Wang and J.-H. Lü, "Software networks nodes impact analysis of complex software systems," *Ruan Jian Xue Bao/Journal of Software*, vol. 24, no. 12, pp. 2814–2829, 2013.

[15] G. Huang, P. Zhang, Y. Li, and J. Ren, "Mining the important nodes of software based on complex networks," *ICIC Express Letters*, vol. 9, no. 12, pp. 3263–3268, 2015.

[16] J.-T. Tang, T. Wang, and J. Wang, "Shortest path approximate algorithm for complex network analysis," *Journal of Software*, vol. 22, no. 10, pp. 2279–2290, 2011.

[17] B. Zhang, G. Huang, H. He, and J. Ren, "Approach to mine influential functions based on software execution sequence," *IET Software*, vol. 11, no. 2, pp. 48–54, 2017.

[18] T. Nguyen, W. Weimery, C. Le Gouesy, and S. Forrest, "Using execution paths to evolve software patches," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, pp. 152-153, USA, April 2009.

[19] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, and A. B. Bener, "Mining trends and patterns of software vulnerabilities," *The Journal of Systems and Software*, vol. 117, pp. 218–228, 2016.

[20] B. Zou, M. Yang, E.-R. Benjamin, and H. Yoshikawa, "Reliability analysis of Digital Instrumentation and Control software system," *Progress in Nuclear Energy*, vol. 98, pp. 85–93, 2017.