Hindawi
Security and Communication Networks
Volume 2019, Article ID 7140480, 11 pages
https://doi.org/10.1155/2019/7140480

Research Article

Mining the Key Nodes from Software Network Based on Fault Accumulation and Propagation

Huang Guoyan,[1,2] Wang Qian ,[1,2] Liu Xinqian ,[1,2] Hao Xiaobing,[1,2] and Yan Huaizhi[3]

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

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

Received 14 October 2018; Accepted 20 January 2019; Published 7 March 2019

Guest Editor: Chunhua Su

The increasement of software complexity directly results in the augment of software fault and costs a lot in the process of software development and maintenance. The complex network model is used to study the accumulation and accumulation of faults in complex software as a whole. Then key nodes with high fault probability and powerful fault propagation capability can be found, and the faults can be discovered as soon as possible and the severity of the damage to the system can be reduced effectively. In this paper, the algorithm MFS_AN (mining fault severity of all nodes) is proposed to mine the key nodes from software network. A weighted software network model is built by using functions as nodes, call relationships as edges, and call times as weight. Exploiting recursive method, a fault probability metric FP of a function, is defined according to the fault accumulation characteristic, and a fault propagation capability metric FPC of a function is proposed according to the fault propagation characteristic. Based on the FP and FPC, the fault severity metric FS is put forward to obtain the function nodes with larger fault severity in software network. Experimental results on two real software networks show that the algorithm MFS_AN can discover the key function nodes correctly and effectively.

1. Introduction

With the development of computer technology and the expansion of software applications [1], the scale and complexity of software systems increase continuously. Software faults directly lead to the rise of system failure ratio, and their reliability is becoming more and more difficult to guarantee. In the test and maintenance process, developers cannot deal with the software problems with a clear purpose [2]. Therefore, if some potentially useful information can be found from software source code or dynamic execution process to help software workers understand the structural characteristics of software quickly, it will be of great significance for improving software development and maintenance efficiency [3–5]. Affected by the achievements in the complex network field, some researchers regard software system as a software network for scientific research. This provides a novel research idea and platform for better understanding and measuring the internal topology structure of complex software system and receives great attention.

The knowledge of complex network has been introduced into software engineering by using network model to represent the structural characteristics of a software system, and researchers have found many novel features of the structure from different points of view [6, 7]. Valerde et al. [8] apply complex network to construct the topology structure of software and propose a method to model the software as an undirected network for the first time. In the method, the node is regarded as the software class and the edge is regarded as the call relationship among the classes. With experiments, they find the "scale-free" and "small-world" properties in software network. Myers et al. [9] use directed network to represent the collaboration relationship among the classes of software. They learn that indegree and outdegree distribution of the network obey the power-law distribution with different exponents. Pan et al. [10] adopt a binary software network to

represent class units or package units and their dependence relationships in a software system, as well as a community detection method to detect the best module partition of the software system. The optimal module partition is compared with the real module partition in the system to guide the optimization design of the module when a software version is updated. Thung et al. [11] propose a new method to simplify the complexity of a software network. Then, they measure the importance of classes from different properties (betweenness, closeness, etc.). Furthermore, they condense the class network which only contains some important classes. By the method, they are able to depict the overall design for software and make the design model easy to understand. Mohammed et al. [12] construct a mapping of research system to identify software security techniques used in the software development process, which enables software developers to understand the existing software security approach better. Thus, software security problems are urgent to be solved.

Measuring the importance of nodes accurately in software network is the premise to improve the security and reliability of software [13]. In software network, a few key nodes have an important effect on the overall stability, reliability, and robustness of the system [14], such as the impact of cascading failure propagation. If there are faults in these nodes, it can result in partial or total system crashes and irreversible results. Identifying these nodes and providing them with key protection help to prevent system crash caused by deliberate attacks. Researchers have defined the importance of nodes in software network from different aspects. Freeman [15] utilizes the betweenness to measure the node importance and points out that a node is more important in software network if its betweenness is bigger. Callaw et al. [16] consider that a node is more important in software network if its degree is bigger, because the node with bigger degree connects with more nodes. However, it does not consider the overall structure of a software network and has some limitations. Kitsak [17] makes it clear that the location of a node in network has a great impact on its importance and exploits the k-shell decomposition method to measure the node importance. The metric result is proved to be better than the betweenness and degree of centrality. Turnu et al. [18] measure the quality of software by analysing the degree distribution of nodes in a software network. They define the structure entropy to describe the degree distribution of nodes and prove that the statistical information of the structure entropy in a software network can be related to the number of software bugs. It further proves that there is a relationship between the structure characteristics of a software network and the quality of the software. Wang et al. [19] define the influential nodes in a network by studying the weighted software network at the function level. They analyse the relationship between the statistical characteristics of software network and the influential nodes through experiments. Bhattacharya et al. [20] predict software evolution based on the static graph topology analysis and propose NodeRank value to measure the importance of a node. The fault of a function is not only caused by itself but also affected by other functions. Huang et al. [21] define the importance of a node based on the dependence relationship and the information propagation among functions. Their algorithm MIN can effectively mine the influential nodes in a software network, but its assignment to the probability of information propagation has certain subjectivity.

In complex networks, random walk model judges the importance of a node by considering its own connectivity degree and the importance of neighbouring nodes around it. Typical methods are PageRank, NodeRank [20], and so on. In software network, the CK metric set proposed by Chidamber and Kemerer [22] indicates that the number of classes that are coupled to a given class named CBO can affect the propensity of the class node to contain defects. If the CBO of a class is larger, it is more sensitive when other parts change. So it is harder for software workers to maintain. Ren et al. [23] believe that the more numbers of modules, classes, or functions that are directly or indirectly dependent on the function nodes, the greater the cost of constructing it and the probability of error. Ren et al. [24] also believe that when a function node is the role of the calling function, it may accumulate the defect of the called function node with a certain probability. When a function node is the role of the called function, it may propagate its defects to its caller with a certain probability. Based on the random walk model and combined with the directed weighted feature of software network, the following FP and FPC are proposed.

In summary, this paper focuses on the call dependence relationship among functions and the fault accumulation and propagation of dynamic execution process. Firstly, according to the dynamic execution information of software, we build a weighted software network model. Then, utilizing recursive method, the fault probability metric FP of a function is defined in accordance with fault accumulation characteristic, and the fault propagation capability metric FPC of a function is proposed on the basis of fault propagation characteristic. Finally, by combining FP and FPC, the fault severity metric FS is put forward and the algorithm MFS_AN (mining fault severity of all nodes) is proposed to calculate the FS and obtain the function nodes with larger fault severity in software network.

The rest of this paper is organized as follows. Section 2 describes the process of mining key nodes in a software network based on the fault accumulation and propagation. The experiment results are given in Section 3. Conclusion and future work are mentioned in Section 4.

2. Mining Key Nodes Based on Fault Accumulation and Propagation

2.1. Weighted Function Execution Network. In software network, the different execution times among functions reflect the tightness degree of nodes' interaction. In order to incorporate this difference, this paper constructs a weighted software network model.

Definition 1 (WFEN (Weighted Function Execution Network)).

$$WFEN = (NSet, ESet, Weight) \tag{1}$$

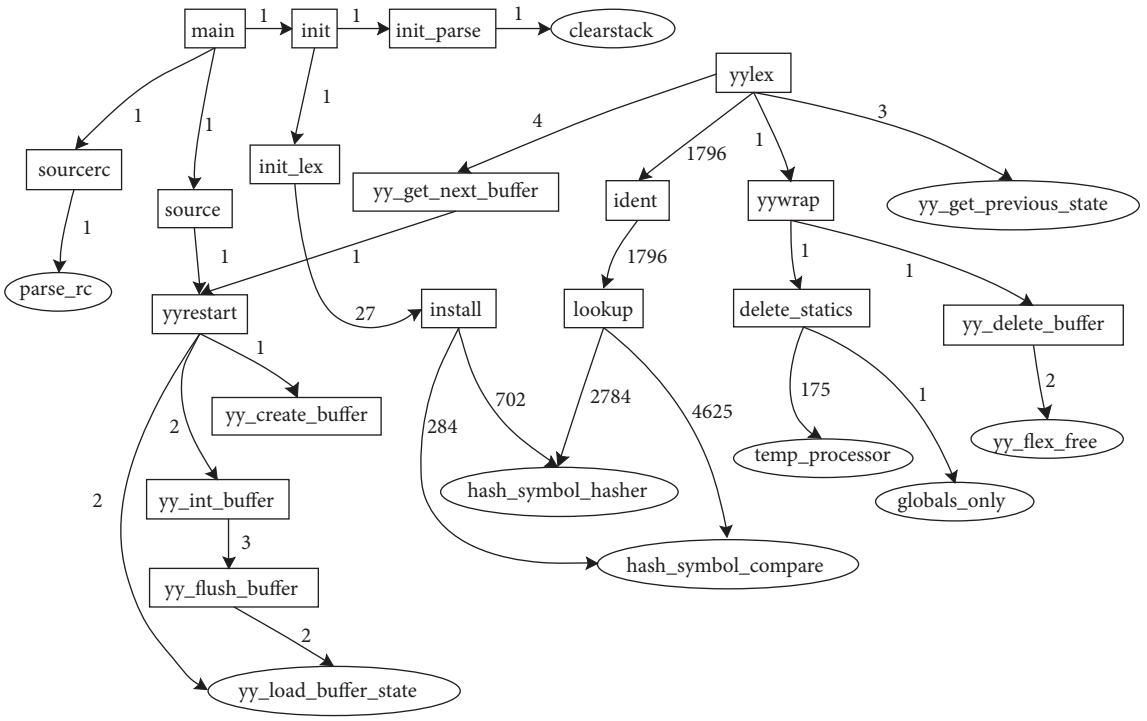

Figure 1: A portion of a weighted function execution network.

where NSet is the function node set of a software network, ESet is the edge set which is the function call relationship during the software execution process, and Weight denotes the execution times that a function calls another one.

Figure 1 represents a portion of a weighted function execution network.

As the software system works, a function is a calling function and also a called function. In the execution process of a function node u, the nodes called directly by u are the direct outdegree neighbor node of u, and the set of these direct out-degree neighbor nodes is called as the Direct Out-degree Neighbor Set (DONS). Similarly, set of the indegree neighbor nodes which call node u directly is named as the Direct In-degree Neighbor Set (DINS).

Definition 2 (DONS (Direct Out-degree Neighbor Set)).

$$DONS(u) = \{v_i \mid u \longrightarrow v_i\}, \quad u, v_i \in NSet \tag{2}$$

Definition 3 (DINS (Direct In-degree Neighbor Set)).

$$DINS(u) = \{v_i \mid v_i \longrightarrow u\}, \quad u, v_i \in NSet \tag{3}$$

2.2. The Fault Probability. Figure 2 shows a more common topology structure of software network. By analyzing these three different topologies, we study the fault accumulation characteristics of functions in software network and obtain the fault probability of a function.

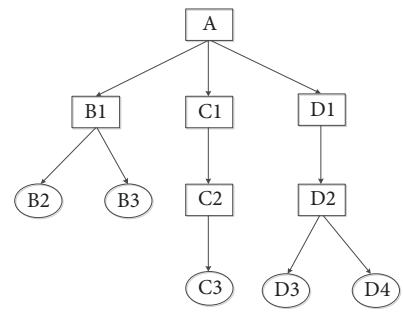

Figure 2: Common topologies in software network.

For nodes B1 and C1, they have the same size of the call function set; that is, the number of call nodes is equal, but the call relationship between these nodes is different. For nodes B1 and D1, they have the same size of execution routes, but the node D1 has a larger call function set and a more complex execution process. Therefore, the influences of the node B1, C1, and D1 on the node A are different.

With the structure shown in Figure 2, we can learn that the function fault is caused not only by itself, but also by its call functions. Moreover, for the objective function, each node in its DONS has different influence on the objective function. For this, we define the fault probability quantitative standard FP of each node which is the accumulation of infection coming from its call nodes. Based on the call relationships among the functions in DONS, the computational formula of the FP is given as follows.

Definition 4 (FP (the fault probability of a node)).

$$FP(u) = \alpha + \sum_{i=1}^{N} P_{u \longrightarrow v_i} * FP(v_i), \quad v_i \in DONS(u) \tag{4}$$

$$P_{u \longrightarrow v_i} = \frac{Weight(u, v_i)}{\sum_{j=1}^{n} Weight(j, v_i)}, \quad j \in DINS(v_i) \tag{5}$$

where α is the fault probability of u caused by itself ($0 \leq \alpha \leq 1$), v_i is a node in $DONS(u)$, N is the size of the $DONS(u)$, $P_{u \longrightarrow v_i}$ is the probability infected by the direct neighbors of u, j is a node in $DINS(v_i)$, and n is the size of the $DINS(v_i)$.

Example 5. Figure 3 is a simple weighted function execution network.

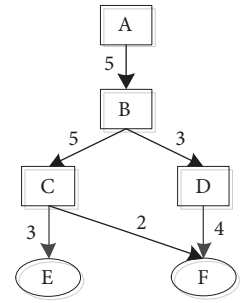

Figure 3: A simple weighted function execution network.

In Figure 3, an example shows how to calculate the FP of a function node. In real world, the size of each function with various definitions is different, and the fault probability of each function is also different. But the setting of specific fault values is more complicated. The main work of this paper is to show the correlation of faults and not to pay attention to the fault calculation method of the node itself. To be universal, we set the fault probability of function node itself to 0.5. That is to say, suppose the probability of fault occurring and not occurring is the same. The function node set NSet = {A, B, C, D, E, F}, and the Direct Out-Degree Neighbor Set of each node is as follows:

$$\begin{aligned} DONS(A) &= \{B\}; \\ DONS(B) &= \{C, D\}; \\ DONS(C) &= \{E, F\}; \\ DONS(D) &= \{F\}; \\ DONS(E) &= DONS(F) = \Phi. \end{aligned} \tag{6}$$

For nodes E and F, which belong to leaf nodes in the software network, then $FP(E) = FP(F) = \alpha = 0.5$; according to the definition of FP, the fault probability of other nodes in the software network is as follows:

$$\begin{aligned} FP(C) &= \alpha + \left(\frac{3}{3} * FP(E) + \frac{2}{2+4} * FP(F)\right) \\ &= 1.1666667; \\ FP(D) &= \alpha + \left(\frac{4}{2+4} * FP(F)\right) = 0.8333334; \\ FP(B) &= \alpha + \left(\frac{5}{5} * FP(C) + \frac{3}{3} * FP(D)\right) = 2.5; \\ FP(A) &= \alpha + \left(\frac{5}{5} * FP(B)\right) = 3.0. \end{aligned} \tag{7}$$

Through the calculation of FP, the fault probability of a function node is identified. According to the above calculation results, the fault probability of each function node in Figure 3 is in the order of: A>B>C>D>E=F.

The fault probability of a node (FP) in Definition 4 is really not probability. It is just a metric of a node, if the FP of a node is higher and the node more likely has faults. So it may take arbitrary large value. Just because FP measures recursive weighted out degree of node u, it happens to embody the process of fault accumulation in a software system. Therefore, in the case of probability cumulative, total more than 1, this is a possibility, not restricted by 1.

Via the above analysis, based on the fault accumulation characteristics of a function and the recursive method, we utilize the formula (4) to calculate the fault probability FP for each function in software network. Then the algorithm MFP_AN (Mining fault probability of all nodes) is proposed to get the FP of all nodes in software network.

In Algorithm 6, we show the process of the method MFP_AN. In line (1), we first initialize a FPList to store the information and fault probability FP of all function nodes. In lines (2-9), a looping procedure calling the procedure MFP to calculate and store the FP for all function nodes is presented. In the procedure MFP, we show the process that the FP of a node is calculated by a recursively process. In line (1), we first define and initialize some related variables. Lines (2-21) describe the process to compute the current node affected by its out-degree neighbor nodes recursively and obtain the FP of the target node.

Algorithm 6 (mining fault probability of all nodes (MFP_AN)).

Input: node set NSet, out-degree adjacency table outDegreeList, in-degree adjacency table inDegreeList

Output: the measuring result FPList of nodes

Process:

```
(01) Initialize FPList;
(02) for (each U ∈ NSet)
(03)     if (tempList.contains(U))
(04)         FPList.add(U,tempList.get(U));
(05)         break;
(06)     end if
(07)     value = MFP (U,outDegreeList,inDegreeList);
(08)     FPList.add(U,value);
(09) end for
(10) return FPList;
```

Procedure (MFP (U: a node; outDegreeList: out-degree adjacency table; inDegreeList: in-degree adjacency table))

```
(01) Initialize α, Fa=0, P=0, FP=0, tempList;
(02) if (outDegreeList[U.index]!=null)
(03)   for (each V ∈ outDegreeList[U.index])
(04)     if (inDegree[V.index]!=null)
(05)       Initialize sum=0;
(06)       for (each I ∈ inDegree[V.index])
(07)         count = Label(I,V);
(08)         sum += count;
(09)       end for
(10)       for (each I ∈ inDegree[V.index])
(11)         if (I.index == U.index)
(12)           count_UV = Label(U,V);
(13)           P = count_UV/sum;
(14)           break;
(15)         end if
(16)       end for
(17)     end if
(18)     Fa += P * MFP
         (V,outDegreeList,inDegreeList);
(19)   end for
(20) end if
(21) FP= α + Fa;
(22) tempList.add(U,FP);
(23) return FP;
```

2.3. The Fault Propagation Capability. Similarly, this section defines the fault propagation capability metric FPC of a function according to the fault propagation characteristics.

Definition 7 (FPC (the fault propagation capability of a node)).

$$FPC(u) = \frac{K_u^{in}}{K_{\max}^{in}} + \sum_{i=1}^{N} P_{v_i \longrightarrow u} * FPC(v_i), \quad v_i \in DINS(u) \tag{8}$$

$$P_{v_i \longrightarrow u} = \frac{Weight(v_i, u)}{\sum_{j=1}^{n} Weight(v_i, j)}, \quad j \in DONS(v_i) \tag{9}$$

where K_u^{in} is the in-degree of a node u, $K_{\max}^{in}$ is the maximum value of in-degree in the network, $K_u^{in}/K_{\max}^{in}$ denotes the fault propagation capability of the objective function itself, v_i is a node in $DINS(u)$, N is the size of the $DINS(u)$, $P_{v_i \longrightarrow u}$ is the probability of u called by functions in the $DINS(u)$, j is a node in $DONS(v_i)$, and n is the size of the $DONS(v_i)$.

Example 8. Based on Figure 3, an example shows how to calculate the FPC of a function node. The function node set NSet={A, B, C, D, E, F}, $K_{\max}^{in}$=2 and the Direct In-Degree Neighbor Set of each node is as follows:

$$\begin{aligned} &DINS(A) = \Phi; \\ &DINS(B) = \{A\}; \\ &DINS(C) = DINS(D) = \{B\}; \\ &DINS(E) = \{C\}; \\ &DINS(F) = \{C, D\}. \end{aligned} \tag{10}$$

According to the definition of FPC, the fault propagation capability of all nodes in the software network is as follows:

$$\begin{aligned} FPC(A) &= \frac{K_A^{in}}{K_{\max}^{in}} = 0; \\ FPC(B) &= \frac{K_B^{in}}{K_{\max}^{in}} + \left(\frac{5}{5} * FPC(A)\right) = 0.5; \\ FPC(C) &= \frac{K_C^{in}}{K_{\max}^{in}} + \left(\frac{5}{5+3} * FPC(B)\right) = 0.8125; \\ FPC(D) &= \frac{K_D^{in}}{K_{\max}^{in}} + \left(\frac{3}{5+3} * FPC(B)\right) = 0.6875; \\ FPC(E) &= \frac{K_E^{in}}{K_{\max}^{in}} + \left(\frac{3}{3+2} * FPC(C)\right) = 0.9125; \\ FPC(F) &= \frac{K_F^{in}}{K_{\max}^{in}} + \left(\frac{2}{3+2} * FPC(C) + \frac{4}{4} * FPC(D)\right) \\ &= 2.0125. \end{aligned} \tag{11}$$

Through the calculation of FPC, the fault propagation capability of a function node is identified. According to the above calculation results, the fault propagation capability of each function node in Figure 3 is in the order of F>E>C>D>B>A.

Algorithm 9 (mining fault propagation capability of all nodes (MFPC_AN)).

Input: node set NSet, out-degree adjacency table outDegreeList, in-degree adjacency table inDegreeList, inDegreeMax

Output: the measuring result FPCList of nodes

Process:

```
(01) Initialize FPCList;
(02) for (each U ∈ NSet)
(03)   if (tempList.contains(U))
```

```
(04)        FPCList.add(U,tempList.get(U));
(05)        break;
(06)     end if
(07)     value = MFPC (U,outDegreeList,
         inDegreeList,inDegreeMax);
(08)     FPCList.add(U,value);
(09) end for
(10) return FPCList;
```

Procedure (MFPC (U: a node; outDegreeList: out-degree adjacency table; inDegreeList: in-degree adjacency table; inDegreeMax: maximum value of in-degree))

```
(01) Initialize Fp=0, P=0, inDegree=0, FPC=0, tempList;
(02) if (inDegreeList[U.index]!=null)
(03)    inDegree = inDegree[U.index].size;
(04)    for (each V ∈ inDegreeList[U.index])
(05)       if (outDegree[V.index]!=null)
(06)          Initialize sum=0;
(07)          for (each I ∈ outDegree[V.index])
(08)             count = Label(V,I);
(09)             sum += count;
(10)          end for
(11)          for (each I ∈ outDegree[V.index])
(12)             if (I.index == U.index)
(13)                count_VU = Label(V,U);
(14)                P = count_VU/sum;
(15)                break;
(16)             end if
(17)          end for
(18)       end if
(19)       Fp += P * MFPC
           (V,outDegreeList,inDegreeList,inDegreeMax);
(20)    end for
(21) end if
(22) FPC= inDegree/inDegreeMax + Fp;
(23) tempList.add(U,FPC);
(24) return FPC;
```

Similarly, via the above analysis, based on the fault propagation characteristics of a function and the recursive method, we use the formula (8) to calculate the fault propagation capability FPC for each function in software network. Then the algorithm MFPC_AN (mining fault propagation capability of all nodes) is proposed to get the FPC of all nodes in software network. Algorithm 9 is similar to Algorithm 6.

2.4. The Fault Severity. Some researchers believe that the type of fault determines the behaviour of transmission [25, 26]. That is, different faults in the same software have different laws of propagation behaviour. The research focuses on the study of fault characteristics. However, other researchers believe that the system architecture determines the behaviour of fault propagation [27]. That is, the same fault in different architectures can be evolved into system failure with different types or different severity levels. This view is based on the analysis of the system structure. It focuses on the regularity of the propagation of faults in the architecture. This paper mainly studies the latter. Therefore, this article firstly calculates the fault probability FP and the fault propagation capability FPC of a function node, respectively. Then the two points are taken into account in this function node. Supposing it fails, the possible fault severity FS of the software system is calculated. Under this premise, we study the function fault characteristics of software system based on architecture.

In Sections 2.2 and 2.3, the fault probability and the fault propagation capability of a function node have been studied, respectively. The former is measured from the Out-Degree Neighbour of a function node, or to say that is the node affected by others. The latter is measured from the in-degree neighbour of a function, or to say that is the effect of the node on others. However, only a comprehensive consideration of these two aspects can fully measure the severity of the damage to a software system.

A node is more likely to have faults if its FP is higher, and it should be paid more attention. However, if a function only has faults but it does not spread its own faults, then the node will not cause very serious consequences to software system, while if a function is not only prone to fail but also has a strong capability to spread its faults to others, then it will cause very serious consequences to software system. Therefore, from the perspective of the fault severity, the fault probability FP and the fault propagation capability FPC of a function are directly proportional. The definition of FS (The fault severity) is given as follows.

Definition 10 (FS (the fault severity)).

$$FS(u) = FP(u) * FPC(u), \quad u \in NSet \tag{12}$$

where $FP(u)$ is the fault probability of a function node u and $FPC(u)$ is the failure propagation capability of u. They jointly determine the fault severity to a software system when the function node u fails. And if a node is with a bigger FS, it will have greater impact on the software system and then it is more critical.

First, we obtain the fault probability set FPList and the failure propagation capability set FPCList of software network through Algorithms 6 and 9, respectively. Then, we use formula (12) to calculate the fault severity FS, and the algorithm MFS_AN (mining fault severity of all nodes) is proposed to discover the top-k key nodes from software network.

In Algorithm 11, the process of the method MFS_AN is presented. Line (1) first initializes an empty FSList set that

stores the FS of all function nodes. Lines (2-7) present a looping process that calculates the FS. In line (8), the FS in the FSList are sorted. In Line (11), the first k function nodes in the FSList are selected as the key nodes of software network.

Algorithm 11 (mining fault severity of all nodes (MFS_AN)).

Input: node set NSet, FPList, FPCList

Output: the top-k key nodes list Knodes

Process:

(01) **Initialize** FSList;

(02) **for** (each U ∈ NSet)

(03) FP(U)=FPList.get(U);

(04) FPC(U)=FPCList.get(U);

(05) FS(U)=FP(U) ∗ FPC(U);

(06) FSList.add(U,FS(U));

(07) **end for**

(08) FSList.sort();

(09) Knodes = FSList.get(K);

(10) **return** Knodes

3. Experiment and Analysis

In this section, we verify the method MFS_AN by testing two kinds of classic tool software Tar and Cflow obtained from the open source community. Tar is a file compression and decompression tool. Cflow is a C program analysis tool for tracking the calling process of functions in the C program. In the Linux environment, we can extract the functions and the dependence relationships of open-source software with the help of tool pvtrace. The results are output to files as text (such as graph.dot). The nodes and the dependence relationships then can be graphically displayed by means of the visualization tool Graphviz. As the main function must be very important to software, so it is excluded in the following experimental verification. In addition, before the experiment, we pretreat the experimental data and delete the loop in the software network, so that recursion can be finished successfully.

3.1. The Distribution of FS. By tracking the execution process of Tar and Cflow, the dynamic execution information of the two types of software is obtained, and the weighted function execution network WFEN is constructed as the basis of experimental data. The fault probability FP and the fault propagation capability FPC of all functions are obtained by mapping the node set and the call relationships of software network to Algorithm 6 (MFP_AN) and Algorithm 9 (MFPC_AN). The return values of Algorithms 6 and 9 are mapped to Algorithm 11 (MFS_AN). The fault severity FS of all nodes and the key nodes in software network are obtained. Figures 4 and 5 show the fault severity scores and the distribution of key nodes in the different versions of Tar and Cflow.

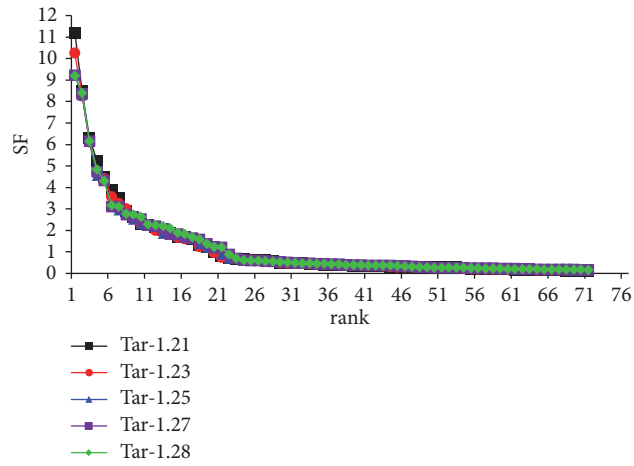

Figure 4: SF value distribution of Tar.

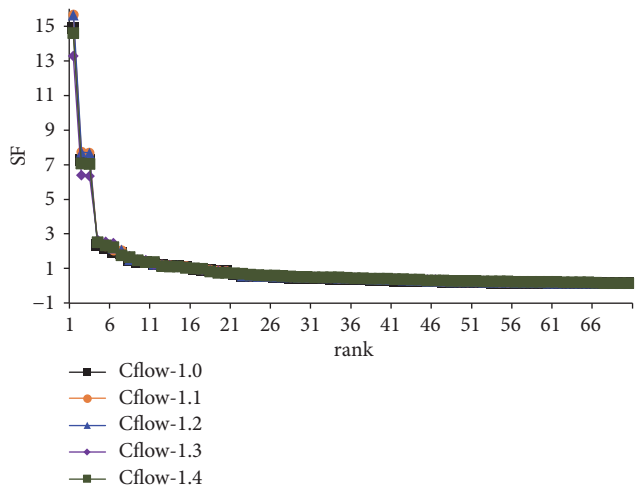

Figure 5: SF value distribution of CFlow.

From the results distribution shown in Figures 4 and 5 (the first 70 nodes are chosen because of the number of nodes in different versions is different), we can summarize the following rules:

(1) We can find that every result distribution obeys the power-law distribution. With the distribution, we verify that software network shows the scale-free properties of complex network.

(2) There are a few nodes with big FS and most of nodes with small FS. But their criticality and impact on the overall software architecture can be reflected in the higher scores.

(3) Their curves are basically at the same trend in different versions of the Tar and Cflow. In other words, in different versions, if the function nodes have the same criticality, the fault severity to software system is no big difference.

By analysing the node criticality in the two types of software from Figures 4 and 5, the key nodes in the software network are defined according to the FS, as the hierarchical structure of FS distribution is obvious, according to the turning point of curves in the graph, to select Top-10 as the key nodes of Tar and Cflow software network, respectively. In

Table 1: The rank of function nodes by SF for Tar versions.

	Rank/value				
Node	Tar-1.21	Tar-1.23	Tar-1.25	Tar-1.27	Tar-1.28
flush_archive	1/11.178	1/10.253	1/9.238	1/9.221	1/9.208
dump_file	2/8.509	2/8.256	2/8.306	2/8.357	2/8.408
dump_file0	3/6.287	3/6.120	3/6.150	3/6.150	3/6.150
find_next_block	4/5.220	4/4.676	5/4.323	5/4.313	5/4.304
update_archive	5/4.460	5/4.412	4/4.556	4/4.729	4/4.830
gnu_flush_write	6/3.849	6/3.567	6/3.112	6/3.107	7/3.103
_gnu_flush_write	7/3.475	7/3.246	8/2.744	8/2.740	9/2.737
create_archive	8/2.873	8/3.004	7/2.944	7/3.104	6/3.170
dump_regular_file	9/2.621	9/2.642	9/2.509	9/2.629	10/2.629
open_archive	10/2.311	10/2.292	10/2.299	10/2.534	8/2.768

Table 2: The rank of function nodes by SF for Cflow versions.

	Rank/value				
Node	Cflow-1.0	Cflow-1.1	Cflow-1.2	Cflow-1.3	Cflow-1.4
nexttoken	1/14.878	1/15.642	1/15.642	1/13.266	1/14.610
yylex	2/7.282	2/7.728	2/7.728	3/6.341	3/7.037
get_token	3/7.255	3/7.670	3/7.670	2/6.382	2/7.063
yyparse	4/2.370	4/2.492	4/2.530	4/2.656	4/2.514
parse_declaration	5/2.175	5/2.308	5/2.355	5/2.547	5/2.349
parse_dcl	6/1.949	7/2.046	6/2.188	6/2.455	6/2.241
expression	7/1.890	6/2.069	7/2.069	7/1.831	7/1.775
yyrestart	8/1.479	9/1.479	9/1.479	10/1.479	8/1.663
func_body	9/1.377	8/1.510	8/1.510	9/1.513	11/1.381
append_to_list	10/1.371	10/1.347	10/1.347	--	--

Table 3: In/Out-degree statistics of ranking top-5 and back-5 nodes in Cflow-1.4.

Rank	Node	K_{in}	K_{out}
1	nexttoken	14	2
2	get_token	1	1
3	yylex	1	5
4	yyparse	1	5
5	parse_declaration	1	4
-5	clear_active	1	0
-4	set_active	1	0
-3	compare	1	0
-2	depmap_alloc	1	0
-1	register_output	1	0

Tables 1 and 2, we present the key nodes and their rank for different versions of the Tar and Cflow.

From the data shown in Tables 1 and 2, the following rules can be summarized:

(1) For a given function node, the criticality ranking in different versions is basically stable. Although there is a change about the ranking of a specific function node in different software versions, the change is very small. For example, in Table 1, the ranking range of the function node find_next_block is [4, 5] and the criticality ranking of the function node dump_file has been stable at 2 in different versions.

(2) Due to the ranking stability of the key nodes in software evolution, we have sufficient reason to predict the position of a function node in a new software version. For instance, in Table 2, the function node yylex is always more critical in each version, and then we can predict that it is likely to still be more critical in the next up-to-date version.

3.2. Correctness Verification

3.2.1. Degree Distribution of FS. In order to illustrate the correctness of the key nodes, taking Cflow-1.4 as an example, we use indegree K_{in} and outdegree K_{out} these two indicators, respectively, as the criticality characterization of a function in software network.

According to the data in Table 3, it can be explained that although the criticality of a function node is not directly related to degree, they have a certain positive correlation. The outdegree values of top-5 are bigger; then their fault probability is greater, and the in-degree values are also bigger; then their fault propagation capability is greater as well, so the overall fault severity is greater. While the back-5 are all leaf

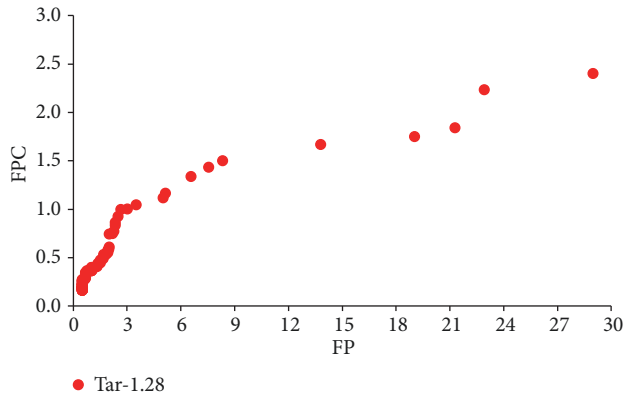

Figure 6: Joint distribution of FP and FPC in Tar-1.28.

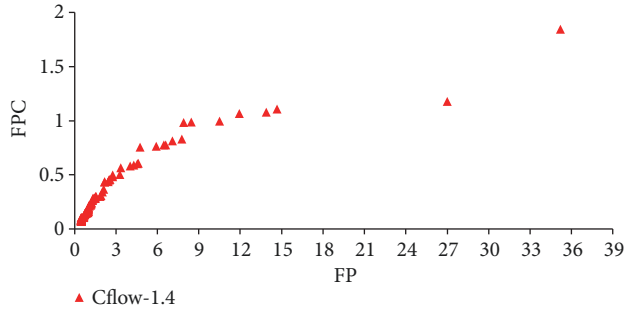

Figure 7: Joint distribution of FP and FPC in Cflow-1.4.

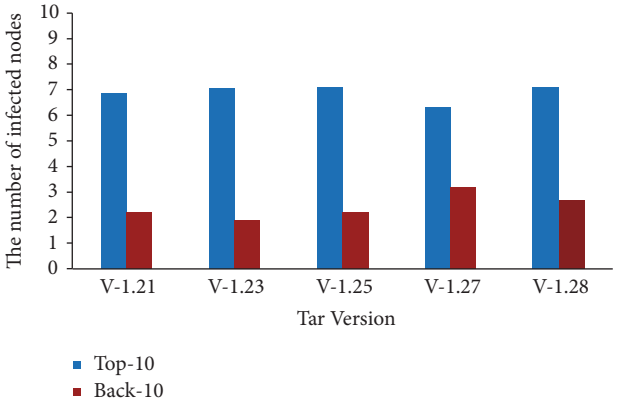

Figure 8: IC Model simulation results for different Tar versions.

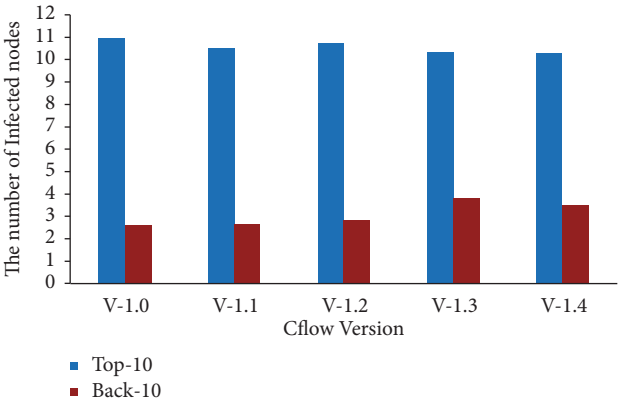

Figure 9: IC Model simulation results for different Cflow versions.

nodes with one in-degree, even if they fail, the range of fault propagation is limited. The different versions of Tar and other versions of Cflow are similar to Table 3 and they will not be described in detail here.

3.2.2. Joint Distribution of FP and FPC. Figures 6 and 7 show the joint distribution of the FP and FPC in Tar-1.28 and Cflow-1.4 software networks. As shown, most functions are located at the lower left corner of the graph; it means that the FP and FPC of these functions are relatively small; a small number of functions are located in the middle of the graph; it means that the FP and FPC of them are relatively big; only a very small number of functions are at the upper right corner of the graph; it signifies that the FP and FPC of these functions are big. Such functions are not only prone to fail but also have a strong fault propagation capability. If they fail, the fault severity to system disruption will be greater. In order to ensure the stability of software system, we should pay more attention to such functions and guarantee their correctness and robustness.

3.2.3. IC Model. In social network, the Independent Cascade Model (IC Model) is a propagation model of researching influence maximization problem. It is a probabilistic model. When a node u is activated, it tries to activate its inactivated neighbor node v with probability P_{uv}. This attempt is only done once, and these attempts are independent of each other. That is to say, the activation of u to v is not affected by other nodes.

In software network, when a failed node u is called, it propagates faults to the neighbor node that calls it with a probability P_{uv}. If the node u can affect a number of nodes, the severity of its failure is significant. This is very similar to the maximization of influence in social network. Therefore, we use IC model to verify that the proposed method MFS_AN does help to measure the fault severity of a node.

According to the mining results of the two kinds of software Tar and Cflow, the top-10 nodes and back-10 nodes are selected as source nodes, respectively. Through the IC model to simulate the number of nodes they can affect after being called, which in turn shows the severity of their failure. Due to the randomness of the IC model, we repeated the simulation 10 times for each version of each kind of software and then averaged the results, as shown in Figures 8 and 9.

As can be seen from Figures 1 and 2, if the functions fail, the number of nodes that can be affected by the top-10 function nodes is about 4 to 5 times that of the back-10 function nodes. This shows that if the top-10 function nodes fail, the fault severity of the software system will be 4 to 5 times that of the back-10 function nodes. Therefore, the ranking of node importance by the MFS_AN method does help to measure the fault severity of a node.

Table 4: The comparison of node rankings in Cflow-1.4.

Cflow-1.4)	Rank/value	
Node	MFS_AN	Degree
nexttoken	1/14.610	1/16
get_token	2/7.063	10/2
yylex	3/7.037	6/6
yyparse	4/2.514	6/6
parse_declaration	5/2.349	7/5
parse_dcl	6/2.241	5/7
expression	7/1.775	4/8
yyrestart	8/1.663	6/6
tree_output	9/1.489	2/14
linked_list_iterate	10/1.383	5/7

Table 5: The comparison of node rankings in Tar-1.28.

Tar-1.28	Rank/value	
Node	MFS_AN	Degree
flush_archive	1/9.208	9/4
dump_file	2/8.408	8/5
dump_file0	3/6.150	3/13
update_archive	4/4.830	1/14
find_next_block	5/4.304	7/6
create_archive	6/3.170	4/9
gnu_flush_write	7/3.103	11/2
open_archive	8/2.768	9/4
_gnu_flush_write	9/2.737	11/2
dump_regular_file	10/2.629	4/9

3.3. Comparison with Degree Method. The algorithm MFS_AN measures the node criticality from two aspects of the outdegree and indegree in the whole network structure. In directed graph, the degree centrality algorithm is a classical algorithm to measure the node criticality from outdegree and indegree. Thus, this paper compares the algorithm MFS_AN with degree centrality algorithm (denoted as Degree). Tables 4 and 5 show the comparative results of Cflow-1.4 and Tar-1.28.

The node ranking lists presented in Tables 4 and 5 are different, and the Degree method has the phenomenon that the same metric value of multiple nodes results in the same ranking. The reason is that the MFS_AN method starts from the fault accumulation and propagation characteristics of a function and focuses on out-degree neighbor nodes and In-Degree Neighbor nodes that have direct or indirect relationship with the current function node. It considers the global influence of the node. While the Degree method only pays attention to the direct out-degree and in-degree neighbor node of the current function node, it ignores the indirect influence of other nodes. However, in software network, the nodes are not isolated and they realize the complicated software function by calling each other. Therefore, compared with Degree method, MFS_AN method can identify the structure of a software more clearly and mine the key nodes of software network more accurately.

In summary, the algorithm MFS_AN proposed in this paper is correct and effective for the node criticality evaluation in software network. By using the algorithm MFS_AN to identify the key nodes in software network, it is helpful to reduce the software fault severity and improve the robustness and stability of software.

4. Conclusions and Future Work

In this paper, a novel algorithm MFS_AN is proposed to evaluate the criticality of nodes in software network by combining the two characteristics of fault probability and fault propagation capability together. And function nodes with larger fault probability and stronger fault propagation capability are regarded as the key nodes. With experiment, we analyse the FS distribution of the nodes in different software versions, realize the evolution law of software, and prove the algorithm MFS_AN can discover the key function nodes correctly and effectively in software network. On the other hand, the criticality of a function node is not directly related to degree, but it has a certain positive correlation. Furthermore, we could understand the software structure more easily and reduce the workload of testing and maintenance process to a maximum extent. In the future research, we will focus on how to divide the software module based on the key nodes.

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 is supported by the National Key R&D Program of China under Grant No. 2016YFB0800700, the National Natural Science Foundation of China under Grant Nos. 61472341, 61772449, 61572420, 61807028, and 61802332, the Natural Science Foundation of Hebei Province China under Grant No. F2016203330, and the Advanced Program of Postdoctoral Scientific Research under Grant No. B2017003005.

References

[1] F. T. Imam and T. R. Dean, "Modelling functional behavior of eventbased systems: a practical knowledge-based approach," *Procedia Computer Science*, vol. 96, pp. 617–626, 2016.

[2] G. McGraw, "Four software security findings," *The Computer Journal*, vol. 49, no. 1, pp. 84–87, 2016.

[3] S. K. Punia, A. Kumar, and A. Sharma, "Evaluation the quality of software design by call graph based metrics," *Global Journal of Computer Science and Technology*, vol. 14, no. 2, pp. 59–64, 2014.

[4] G. Huang, B. Zhang, R. Ren et al., "An algorithm to find critical execution paths of software based on complex network," *International Journal of Modern Physics C*, vol. 26, no. 09, Article ID 1550101, pp. 1550101-1–1550101-16, 2015.

[5] S. Lamzabi, S. Lazfi, H. Ez-Zahraouy et al., "Pair-dependent rejection rate and its impact on traffic flow in a scale-free network," *International Journal of Modern Physics C*, vol. 25, no. 07, pp. 1450019-1–1450019-10, 2014.

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

[7] C. Y. Chong and S. P. Lee, "Analyzing maintainability and reliability of object-oriented software using weighted complex network," *Journal of Systems & Software*, vol. 110, no. C, pp. 28–53, 2015.

[8] S. Valverde and R. V. Sole, "Hierarehieal small worlds in software architecture," *Dynamics of Continuous Discrete & Impulsive Systems*, vol. 14, p. 1, 2003.

[9] C. R. Myers, "Software systems as complex networks: structure, function, and evolvability of software collaboration graphs," *Physical Review E: Statistical, Nonlinear, and Soft Matter Physics*, vol. 68, no. 4, pp. 046116-1–046116-15, 2003.

[10] W. Pan, B. Li, B. Jiang et al., "Recode: software package refactoring via community detection in bipartite software networks," *Advances in Complex Systems. A Multidisciplinary Journal*, vol. 17, no. 7n08, Article ID 1450006, 2014.

[11] F. Thung, D. Lo, M. H. Osman et al., "Condensing class diagrams by analyzing design and network metrics using optimistic classification," in *Proceedings of the 22nd International Conference on Program Comprehension, ICPC '14*, pp. 110–121, ACM, Hyderabad, India, June 2014.

[12] N. M. Mohammed, M. Niazi, M. Alshayeb et al., "Exploring software security approaches in software development lifecycle: a systematic mapping study," *Computer Standards & Interfaces*, vol. 50, pp. 107–115, 2016.

[13] W.-F. Pan, B. Li, Y.-T. Ma, Y.-Y. Qin, and X.-Y. Zhou, "Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks," *Journal of Computer Science and Technology*, vol. 25, no. 6, pp. 1202–1213, 2010.

[14] J. Liu, "The structure and the volatility of the software," in *Blue*, H. KeQing, Ed., pp. 156–164, Software Network (Beijing:Science Press), 2008.

[15] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social Networks*, vol. 1, no. 3, pp. 215–239, 1978.

[16] D. S. Callaway, M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Network robustness and fragility: percolation on random graphs," *Physical Review Letters*, vol. 85, no. 25, pp. 5468–5471, 2000.

[17] M. Kitsak, L. K. Gallos, S. Havlin et al., "Identification of influential spreaders in complex networks," *Nature Physics*, vol. 6, no. 11, pp. 888–893, 2010.

[18] I. Turnu, M. Marchesi, and R. Tonelli, "Entropy of the degree distribution and object-oriented software quality," in *Proceedings of the 2012 3rd International Workshop on Emerging Trends in Software Metrics, WETSoM 2012*, pp. 77–82, Switzerland, June 2012.

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

[20] P. Bhattacharya, M. Iliofotou, I. Neamtiu et al., "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.

[21] 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.

[22] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[23] J. Ren, H. Wu, T. Yin, L. Bai, and B. Zhang, "A novel approach for mining important nodes in directed-weighted complex software network," *Journal of Computational Information Systems*, vol. 11, no. 8, pp. 3059–3071, 2015.

[24] J. Ren, H. Wu, R. Gao et al., "Identifying important nodes in complex software network based onripple effects," *Icic Express Letters Part B Applications An International Journal of Research and Surveys*, p. 7, 2016.

[25] J. Zhang, "The calculating formulae, and experimental methods in error propagation analysis," *IEEE Transactions on Reliability*, vol. 55, no. 2, pp. 169–181, 2006.

[26] M. Shafique, S. Rehman, P. V. Aceituno et al., "Exploiting program-level masking and error propagation for constrained reliability optimization," in *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*, pp. 1–9, Austin, Tex, USA, June 2013.

[27] M. Hiller, A. Jhumka, and N. Suri, "PROPANE: an environment for examining the propagation of errors in software," *Acm Sigsoft Software Engineering Notes*, vol. 27, no. 4, pp. 81–85, 2003.