Mining Important Functions in Software Network by Node Vulnerability

Bing Zhang¹,², Shengting Sun¹,², Xiaobing Hao¹,²

¹School of Information Science and Engineering, Yanshan University, Qinhuangdao, Hebei P.R.China
²The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province, Qinhuangdao City, China, 066004

Abstract. Given that analysis on the vulnerability of functions is helpful to the detection and improvement of software security, this paper aims to propose an efficient methods to identify the vulnerable nodes (ITVN) in different software by the interdependence of functions. First, the dynamic software execution process was constructed as Software Execution Dependency Network (SEDN) based on Complex network theory. Second, by analyzing the dependency relationship among functions, the algorithm calVulAndScoOfNodes (CVSN) was designed to compute the vulnerability and the affected scope of each node for further analysis. Third, in order to measure the functions vulnerability in the whole software network, the algorithm calVulDegreeOfNodes (CVDN) was put forward to calculate the vulnerable degree of each node. Finally, the Vulnerable Nodes in different software were obtained by ITVN. Experimental results show that the vulnerable nodes selected as important nodes are well-reasoned in software network by testing different software, and the measures are effective for evaluating nodes vulnerability.

1. Introduction

As nowadays functional design is diversifying, it becomes more difficult to understand software structure [1], one of the most important factors to the quality of the final software products [2]. There is no doubt that relevant studies conducted on it is convenient for software maintenance and version updating. As the characteristics shown in the execution process [3-4] is helpful to understand software structure, the discovering of some new potential characteristics is thus significant to ensure software quality and control its execution [5-6]. In recent years, researchers in the field of statistical physics and complex system turn to consider software system as complex software networks by taking software components like packages, classes or methods as nodes and their relationships as edges [7], which provides us a new approach to study complex software systems. Therefore, understandings on the topology structure, components and their interactions in software network has attracted increasing attention from scholars and researchers [8-12].

During the execution process, software functions carry most of software attribute features and typology information which are important in the stability and robustness of software system. Researchers pointed out that mechanisms such as cascading and spreading are highly affected by key nodes, even a tiny fraction [13]. The identifying of key nodes is thus of remarkable theoretical and practical significance. Freeman [14] adopted betweenness to evaluate the importance of nodes, and pointed out that nodes with larger betweenness may be more important than others in software network. However, it is very time-consuming to calculate the betweenness of each node. Callaw et al.
[15] took node degree to measure the importance of nodes of which those with larger degrees are regarded as key nodes. However, as a node connects two communities in a network, though without large degree, it is still considered as an important node because the two communities will not be connected if the node is removed, so it is fair to say this method is low-relevant to software global structural information. Kitsak et al. [16] realized that the positions of nodes are important in global network and proposed k-shell decomposition analysis accordingly to obtain the ranking index of nodes. However, the k-shell decomposition concept fails to implement the ranking of spreaders in the same k-shell index, though the positions of nodes in global networks is confirmed and illustrated. Bae et al. [17] proposed a novel measure called coreness centrality to estimate the spreading influence of a node in a network by referring to the k-shell indices of its neighbour nodes. Based on the idea that a powerful spreader has more connections to other nodes in the core of the network, it pointed out that the number of neighbours of a node has greater influences on its spreading ability. Also, Liu et al. [18] presented an improved method to generate the ranking list to evaluate the node spreading influence by considering the shortest distance between a target node or node set of the highest k-core value. They took a new perspective to understand the relationship between the k-shell location and the shortest node distance to the core of network. Li Ding-wei et al. [19] introduced a weighted software network model to represent the structure of object-oriented software and defined an indicator IC to measure the importance of nodes in the network. It needs to get the Extra-Class method reachability set of each function in each class which to say this process is excessively time-consuming. Ugander et al. [20] found that the number of connected sub-graphs between neighbour nodes has higher influence on the importance of nodes than absolute number of neighbour nodes. Chen et al. [21] proposed a local centrality measure as a tradeoff between the low-relevant degree centrality and other time-consuming measures which, however, neglected the nearest neighbours in identifying important nodes. In fact, if a function node is a caller, it may accumulate the callees’ vulnerabilities. However most of the existing approaches failed to take vulnerability into consideration when mining important nodes in complex software network.

Based on these researches, we paid special attention to the structure of software network and the relevant information in the dynamic execution process from the perspective of vulnerable nodes. According to the characteristics in software execution we build the Software Execution Dependency Network (SEDN) to represent the structure of software. Then, index Node Vulnerability (NV) was defined to identify vulnerable node by relying on their relationships with other nodes in SEDN. Likewise, index Vulnerable Degree (VD) was also defined to measure the degree of the vulnerability of each node in the whole system.

The rest of this paper is organized as follows. Section 2 introduces some of the preparatory work and the basic definitions. Section 3 gives a detailed description of identifying the vulnerable nodes and proposes a measurement to evaluate the degree of vulnerability. In Section 4 the experiment results are presented, while relevant conclusions are drawn in the section 5.

2. Materials and Methods

2.1. Preliminaries

In software system, dynamic software execution process can be abstracted as complex network called Software Execution Dependency Network (SEDN), in which nodes represent software components, such as functions, methods, classes, objects or modules, and edges denote the interactions or relationships between components and weight on edges indicates whether there exists call relationship between each pair of components. We use a triple \( \{ V, E, W \} \) to describe the SEDN, where \( V \) is a set of nodes in the network, like \( \{ v_1, v_2, ..., v_i, ..., v_n \} \). \( E \) is a set of directed edges \( \{ <v_1, v_2>, <v_2, v_3>, ..., <v_i, v_j>, ... \} \). \( W \) is the possibility of \( v_i \) influenced by \( v_j \), which represent the value that \( v_i \) may accumulate the vulnerabilities from \( v_j \). It is considered as the weight of each edge and defined in formula (1).
Definition 2.1. Vulnerable Accumulation (VA). VA is the possibility that \(v_i\) may accumulate vulnerabilities from \(v_j\), which is defined as follows.

\[
VA_j = \frac{IFCall}{\text{outDegree}(v_j)} \ast \frac{IFBeCall}{\text{inDegree}(v_j)}
\]

(1)

Here, if \(v_i\) calls \(v_j\), IFCall is 1, otherwise IFCall is 0. Likewise, if \(v_j\) is called by \(v_i\), the IFBeCall is 1, otherwise IFBeCall is 0. The denominator is the out-degree of \(v_i\) and the in-degree of \(v_j\) respectively.

Definition 2.2. Node Vulnerability (NV). NV is defined to measure the vulnerability of node \(v_i\).

\[
NV(v_i) = \sum_{j \in S}[VA_j + VA_j \ast NV(v_j)]
\]

(2)

In most cases, the possibility of \(v_i\) accumulating vulnerabilities not only depends on the nodes which \(v_i\) can reach in a single-step, but also in multi-steps. The direct influence is defined in formula (3), while the indirect influence is given in formula (4).

\[
NV(v_i)_{\text{single-step}} = \sum_{j \in S} VA_j
\]

(3)

In formula (3), \(S\) is a set of nodes that \(v_i\) can reach in one step, that's to say, \(v_j\) refer to the set of neighbour nodes of \(v_i\). The \(NV(v_i)_{\text{single-step}}\) of \(v_i\) is the sum of possibility influenced by all nodes which \(v_i\) can reach in one step.

\[
NV(v_i)_{\text{multi-step}} = \sum_{j \in S} [VA_j \ast NV(v_j)]
\]

(4)

In formula (4), if the out degree of \(v_j\) is 0, then the \(NV(v_j)\) is 0. The \(NV(v_i)\) multi-step of \(v_i\) is the sum of possibility influenced by all nodes of which \(v_i\) reaches in multiple steps in each path. In general, there may have a loop between \(v_i\) and \(v_j\), which dramatically increases the number of paths. To address this issue, we considered the ring structure between \(v_i\) and \(v_j\) for only one time to generate paths, as it covers all the Node Vulnerability from \(v_i\) or \(v_j\) once they have been called.

Here, we take the node as a vulnerable Node if its Node Vulnerability is larger than the average NV of all nodes. At last, these nodes formed a vulnerable Node set, which is called VN. The average NV of all nodes can be defined as follows.

\[
aveNV = \frac{\sum_{i=1}^{n}NV(v_i)}{|V|}
\]

(5)

Definition 2.3. Vulnerable Degree (VD). VD is defined to measure the vulnerability degree of one node relative to the whole system. The VD of node \(v_i\) is given as follows.

\[
VD(v_i) = \frac{NV(v_i) - aveNV}{\sum_{i=1}^{n}NV(v_i)}
\]

(6)

The Vulnerable Degree \(VD(v_i)\) can be negative, which means the \(v_i\) function is safety and steady in software system. The bigger the Vulnerable Degree is, the easier the \(v_i\) function infects, the more important the \(v_i\) function is.

Besides, Affected Scope (AS) of \(v_i\) is the number of nodes which can affect \(v_i\). For instance, \(v_i\) calls \(v_j\), if there is a defect in \(v_j\), the vulnerability will affect \(v_i\).

2.2. Mining the Vulnerable Nodes

As far as is concerned, if node \(v_i\) depends on other nodes strongly, either directly or indirectly, it shows that node \(v_i\) is easily affected by other nodes. In other words, node \(v_i\) can accumulate vulnerabilities from others easily, which are considered as the vulnerable nodes in all system. In order to identify the vulnerable nodes, we calculated the vulnerability and affected scope of each node in
algorithm 1 first. Then we computed the vulnerability degrees of nodes in the whole system in algorithm 2, and mined important nodes in the system in algorithm 3 by referring to node vulnerability. The details are shown as follows.

2.2.1. Computing the vulnerability and affected scope of each node. In order to measure the ability that nodes accumulate vulnerabilities, we calculated the vulnerability and affected scope of each node in the software network. We use adjacent table to store each node and its outDegreeList (the child nodes of vi). There is always a direct or indirect calling relationship between vi and vj. If vj contains a vulnerability, then vi may accumulate it. Similarly, vi may accumulate other vulnerabilities if it calls other function nodes.

**Algorithm 1 calVulAndScoOfNodes (CVSN)**

**Inputs:** set V = {v1, v2, ..., vi, ...}, set E = {<v1, v2>, <v2, v3>, ..., ...}, outDegreeList[v1]

**Outputs:** NV(vi), scope

1. for (each vi in V)
2. if (!seqQueue.isExistTheSameOne(vi))
3. seqQueue.EnQueue(vi);
4. scope++;
5. end if
6. NV(vi) = 0;
7. if (outDegreeList[vj]!=null \&\& outDegreeList[vj].size() != 0)
8. for (each vj in outDegreeList[vj])
9. if (!seqQueue.isExistTheSameOne(vj.get Index()))
10. seqQueue.EnQueue(vj.get Index());
11. scope;
12. end if
13. out = outDegreeList[vj.get Index()].size();
14. in = inDegreeList[vj.get Index()].size();
15. VAij = (1/out)*(1/in);
16. NV(vj) += [VAij + VAij * calVulAndScoOfNodes(vj)]
17. end for
18. end if
19. vi. Affected Scope(AS) ← scope;;
20. return NV(vi);
21. end for

As shown in the algorithm 1, we operated on each node in the network in a looping process. In the process, we defined a queue to record the affected scope of vi in line 2 to line 5 and line 9 to line 12 respectively. Then we calculated the vulnerability of each node. We defined VAij in line 15, the vulnerable accumulation of vi from vj in a path. Line 16 is a recursive procedure in which we add all VA that vi can reach in one path. In line 19, we stored the nodes which can affect vi and called the number of them as Affected Scope. And the Node Vulnerability is returned in line 20.

2.2.2. Calculating the vulnerable degree of each node. It is difficult to conclude the common feature of software because the NV of each node is not in the same order of magnitude in different size of software. In order to compare the vulnerable degree of different software, we proposed an index VD to measure each node in the software network as follows.

**Algorithm 2 calVulDegreeOfNodes (CVDN)**

**Inputs:** set V = {v1, v2, ..., vi, ...}, set E = {<v1, v2>, <v2, v3>, ..., ...}, outDegreeList[v1]

**Output:** VD(vi) set

1. for (each node vi in V)
2. weight += calVulAndScoOfNode(vi);
3. end for
(4) aveNV = weight/|V|;
(5) for (each node \(v_i\))
(6) \[ VD(v_i) = (NV(v_i) - aveNV)/weight*100\%; \]
(7) \[ VD(v_i) \text{ set} \leftarrow VD(v_i); \]
(8) end for

As shown in algorithm 2, we calculated the Vulnerable Degree of each node in turn. Line 1 to line 3 is the whole process of computing the vulnerability of all nodes. In line 4 we obtained the average vulnerability of the system. And in line 5 to line 7 the vulnerable degree of each node is calculated.

2.2.3. Identifying the vulnerable nodes. As the measuring results were obtained from Algorithm 1, we chose the vulnerable nodes with larger value of NV than aveNV (the average NV value of the whole system) of the software network. In Algorithm 3, we described the process of mining the vulnerable nodes as follows.

**Algorithm 3 IdenTheVulNodes (ITVN)**

**Inputs:** set V = \(\{v_1, v_2, \ldots, v_i, \ldots\}\), set E = \(\langle v_{s1}, v_{e2} \rangle, \langle v_{s2}, v_{e3} \rangle, \ldots, \ldots\\), outDegreeList[\(v_i\)]

**Output:** resultList

(1) Initialize resultList;
(2) for (each \(v_i\) in V)
(3) \[ \text{value} = NV(v_i); \]
(4) if (value > aveNV)
(5) \[ \text{resultList.add}(v_i); \]
(6) end if
(7) end for
(8) return resultList

In Algorithm 3, we initialized the resultList as the measurement list of all nodes in Line 1. Line 2 to 7 is a looping process to store the value of each node. The sorting process is presented in line 4 to line 6, and vulnerable nodes are added to the resultList when the value is bigger than the aveNV.

3. Results and Discussion

In this section, approaches mentioned above will be tested by four open source software gzip, tar, cflow, and nginx. Gzip is a compression software for Linux system file, tar is a decompression software for Linux, cflow is an analysis tool for C program to extract the relationship of function calls and nginx is high-performance HTTP reverse proxy server. In Linux operating system, software execution network is derived from using open source packages Gcc complied GNU and adding some debugging information. The parameters of different software network formed by software executions are listed in Table 1.

|         | Gzip | Tar | Cflow | Nginx |
|---------|------|-----|-------|-------|
| Nodes   | 50   | 92  | 102   | 353   |
| Edges   | 64   | 110 | 172   | 690   |

3.1. Analysis of the NV of nodes

We ran algorithm calVulAndScoOfNode on software gzip, tar, cflow and nginx. By doing so, we calculated the NV and affected scope of each function node. The analysis results of different software were shown in Figure 1(a-d) respectively.

Seen from the above, the abscissa is the affected scope of \(v_i\), which represents that if \(v_i\) is a vulnerable node, a number of other nodes will easily affect it. The ordinate is the proportion of nodes with the vulnerability degree. For instance, in Figure 1(a), the affected scope is 49 and the node proportion is less than 5\%, which represent that there are less than 5\% nodes which have the character of affecting other 49 nodes in SEDN. There are some implicit information in these figures.

- The affected scope distributions of software are very similar. With the decreasing of the affected scope, the number of nodes in corresponding scope is increasing.
The affected scopes of few nodes are relatively small, usually between 0 and 10, which means that few nodes control the whole software and they are very important. From the perspective of software engineering, the structure of these functions is relatively complex and the in-degree of these nodes in SEDN may be small, which means that these functions are rarely invoked. As these nodes which call other nodes directly or indirectly have greater vulnerability than others and can be influenced by others easily, thus they are considered to be vulnerable nodes.

Figure 1: Affected scope distribution of Gzip, Tar, Cflow and Nginx

Figure 2: Vulnerable Degree distribution of different software
3.2. Analysis of the VD of nodes

We referred to Vulnerable Degree (VD) to evaluate the degree of the software vulnerability. The VD of \( v \) reflects the average contribution to the whole system. The distribution of vulnerable degree of different software is shown in Figure 2.

As shown in Figure 2, there are similar trends of vulnerable degrees among different software. With the ranking of vulnerable degree from small to large, the number of nodes has a decreasing trend. The ratio between the number of nodes and the total number of nodes in software network tends stable when the VD is larger than 1.5 for software Cflow/Nginx, 2 for Tar and 2.5 for Gzip. And the proportion of nodes is less than 10% which means that most nodes are in a stable state and only few nodes are vulnerable nodes, which should be improved and protected in software.

3.3. Vulnerable Nodes

At last, combined with the NV and VD of nodes, we select top K functions as the vulnerable nodes in software in software Gzip, Tar, Cflow and Nginx. While the VD is larger than 1.5 for software Cflow/Nginx, 2 for Tar and 2.5 for Gzip in Figure 5, then K computed respectively are equal to 9 for Cflow/Nginx, 10 for Tar and 7 for Gzip. The Vulnerable Nodes for different software are listed in Table 2.

| Gzip (K=7)       | Tar (K=10)       | Cflow (K=9)      | Nginx (K=9)      |
|------------------|------------------|------------------|------------------|
| inflate          | find_next_block  | get_token        | http_send_header|
| unzip            | flush_read       | collect_symbols  | http_not_modified_header_filter|
| treat_file       | transform_stat_info | append_to_list   | http_headers_filter|
| deflate          | init_buffer      | nexttoken        | http_ssi_body_filter|
| get_method       | gnu_flush_read   | parse_opt        | http_userid_filter|
| zip              | sparse_member_p  | delete_parms     | output_chain     |
| -                | extract_dir      | delete_autos     | http_ssi_header_filter|
| -                | open_archive     | yylex            | http_postpone_filter|
| -                | transform_member_name | -               | -                |

4. Conclusions

In this paper, we identified vulnerable nodes in the software network based on the calling relationships among function nodes so as to get a better understand of software structure. We also mapped a directed network SEDN to discover the relationships among these function nodes during execution process. Besides, we created a measurement NV to denote the vulnerability of each node and the algorithm CVSN to calculate the corresponding value, given the fact that a node can accumulate the vulnerability information from other nodes. For further comparison and contrast, we proposed the new definition VD or measure the vulnerable degrees of different software, which was shown in algorithm CVDN. Based on the NV of each node, the algorithm ITVN was presented to mine the vulnerable nodes in software, and the characteristics of those nodes were analyzed among different software. Based on the experiment results, the method was finally proved to be feasible and effective in obtaining vulnerable nodes in software.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61802332, 61807028, and 61772449, the Natural Science Foundation of Hebei Province China under Grant No. F2016203330. The authors are grateful to the valuable comments and suggestions of the reviewers.
References

[1] Turnu I., Marchesi M., Tonelli R.(2012) Entropy of the degree distribution and object-oriented software quality. Proceedings of the 3rd International Workshop on Emerging Trends in Software Metrics. IEEE Press, pp. 77-82.

[2] Pan W. F., Li B., Ma Y. T., et al.(2010) 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.

[3] Huang G., Zhang B., Ren R., et al.(2015) An algorithm to find critical execution paths of software based on complex network. International Journal of Modern Physics C. vol. 26, no. 9, pp. 1550101.

[4] Lamzabi S., S. Lazfi, Ez-Zahraouy H., et al.(2014) Pair-dependent rejection rate and its impact on traffic flow in a scale-free network. International Journal of Modern Physics C. vol. 25, no. 7, pp. 1450019.

[5] Bouwers E., Correia J. P., Van D. A., et al.(2011) Quantifying the analyzability of software architectures. Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on. IEEE, pp. 83-92.

[6] Bhattacharya P., Iliofotou M., Neamtiu I., et al.(2012) Graph-based analysis and prediction for software evolution. Software Engineering (ICSE), 2012 34th International Conference on. IEEE, pp. 419-429.

[7] Myers C. R.(2003) Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Physical Review E. vol. 68, no. 4, pp. 046116.

[8] Albert R., Barabási A. L.(2002) Statistical mechanics of complex networks. Reviews of modern physics. vol. 74, no. 1, pp. 47.

[9] Newman M. E. J.(2003) The structure and function of complex networks. SIAM review. vol. 45, no. 2, pp. 167-256.

[10] Boccaletti S., Latora V., Moreno Y., et al.(2006) Complex networks: Structure and dynamics. Physics reports. vol. 424, no. 4, pp. 175-308.

[11] Zhang J., Zhou C., Xu X., et al.(2010) Mapping from structure to dynamics: A unified view of dynamical processes on networks. Physical Review E. vol. 82, no. 2, pp. 026116.

[12] Zeng A., Li L.(2011) Coarse graining for synchronization in directed networks. Physical Review E. vol. 83, no. 5, pp. 056123.

[13] Zemanová L., Zhou C., Kurths J.(2006) Structural and functional clusters of complex brain networks. Physica D: Nonlinear Phenomena. vol. 224, no. 1, pp. 202-212.

[14] Freeman L. C.(1978) Introducing a new section on computer program abstracts. Social Networks. vol. 1, no. 2, pp. 103-104.

[15] Callaway D. S., Newman M. E. J., Strogatz S. H., et al.(2000) Network robustness and fragility: Percolation on random graphs. Physical review letters. vol. 85, no. 25, pp. 5468.

[16] Ma L., Ma C., Zhang H. F., et al.(2016) Identifying influential spreaders in complex networks based on gravity formula. Physica A: Statistical Mechanics and its Applications. vol. 451, pp. 205-212.

[17] Bae J., Kim S.(2014) Identifying and ranking influential spreaders in complex networks by neighborhood coreness. Physica A: Statistical Mechanics and its Applications. vol. 395, pp. 549-559.

[18] Liu J. G., Ren Z. M., Guo Q., “Ranking the spreading influence in complex networks,” Physica A: Statistical Mechanics and its Applications, vol. 392, no. 18, pp. 4154-4159, 2013.

[19] Ding-wei L., Bing L., Peng H., et al.(2012) Ranking the Importance of Classes via Software Structural Analysis. Future Communication, Computing, Control and Management, Springer Berlin Heidelberg, pp. 441-449.

[20] Ugander J., Backstrom L., Marlow C., et al.(2012) Structural diversity in social contagion. Proceedings of the National Academy of Sciences. vol. 109, no. 16, pp. 5962-5966.
[21] Chen D., Lü L., Shang M. S., et al. (2012) Identifying influential nodes in complex networks. Physica a: Statistical mechanics and its applications. vol. 391, no. 4, pp. 1777-1787.