PLOS ONE

RESEARCH ARTICLE

Mining dynamic noteworthy functions in software execution sequences

Bing Zhang[1,2], Guoyan Huang[1,2]*, Yuqian Wang[1,2], Haitao He[1,2], Jiadong Ren[1,2]

1 College of Information Science and Engineering, Yanshan University, Qinhuangdao, Hebei, China, **2** The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province, Qinhuangdao, Hebei, China

* hgy@ysu.edu.cn

Citation: Zhang B, Huang G, Wang Y, He H, Ren J (2017) Mining dynamic noteworthy functions in software execution sequences. PLoS ONE 12(3): e0173244. doi:10.1371/journal.pone.0173244

Editor: Zhong-Ke Gao, Tianjin University, CHINA

Received: November 18, 2016

Accepted: February 18, 2017

Published: March 9, 2017

Data Availability Statement: All relevant data are generated from the experiments by the author may be contacted at bingzhang@ysu.edu.cn, or downloaded from https://figshare.com/s/5cc89633d9f468c079a3 and DOI: https://doi.org/10.6084/m9.figshare.4700407.v1.

Funding: This work was supported by National Natural Science Foundation of China Award Number: 61472341, Recipient: Guoyan Huang; National Natural Science Foundation of China Award Number: 61572420, Recipient: Jiadong Ren; Natural Science Foundation of Hebei Province Award Number: F2014203152, Recipient: Jiadong Ren; Natural Science Foundation of Hebei Province

Abstract

As the quality of crucial entities can directly affect that of software, their identification and protection become an important premise for effective software development, management, maintenance and testing, which thus contribute to improving the software quality and its attack-defending ability. Most analysis and evaluation on important entities like codes-based static structure analysis are on the destruction of the actual software running. In this paper, from the perspective of software execution process, we proposed an approach to mine dynamic noteworthy functions (*DNFM*)in software execution sequences. First, according to software decompiling and tracking stack changes, the execution traces composed of a series of function addresses were acquired. Then these traces were modeled as execution sequences and then simplified so as to get simplified sequences (*SFS*), followed by the extraction of patterns through pattern extraction (*PE*) algorithm from *SFS*. After that, evaluating indicators *inner-importance* and *inter-importance* were designed to measure the noteworthiness of functions in *DNFM* algorithm. Finally, these functions were sorted by their noteworthiness. Comparison and contrast were conducted on the experiment results from two traditional complex network-based node mining methods, namely *PageRank* and *DegreeRank*. The results show that the *DNFM* method can mine noteworthy functions in software effectively and precisely.

Introduction

The identification of important entities (function, class, method, implement, etc.) has a remarkable theoretical and practical significance for software designing, development, maintenance and management. For instance, software reliability can be increased by special protection on influential nodes in software. Studies on these noteworthy entities can not only help cut the workload of software testing but also help enhance the accuracy of software maintenance, thus reducing the maintenance costs. Thus, to improve software maintenance and development is an involved and costly task. According to Gartner, global software expenditures for 2010 amounted to $229 billion, with large vendors such as Microsoft and IBM reporting multi-billion dollar costs for software development each year. Most of the development

Award Number: F2016203330, Recipient: Haitao He; and Graduate innovative Foundation of Hebei Province Award Number: 2016SJBS010, Recipient: Bing Zhang.

Competing interests: The authors have declared that no competing interests exist.

cost—an estimated 50 to 90 percent of the total costs—is due to software maintenance. Despite the high costs, software is notoriously unreliable, and software bugs can wreak havoc on software producers and consumers alike—a *NIST* survey estimated the annual cost of software bugs to be about $59.5 billion. Therefore, knowledge on improving the maintenance by effectively identifying which components to debug, test, or refactor motivated us to establish suitable model for software and design a method to mine dynamic noteworthy functions in software system [1].

Today, complex network is always a research hotspot [2–4]. Many researchers study the characteristics of complex network in software system [5–7]. Studies show that software systems have complex network phenomena, namely small-world effect [8] and scale-free property [9]. With further research on software systems, the development and maturing of complex network theory [10, 11] provides a series of classic ways [12–16] for software functions analysis. It is worthy to be noticed that there have been many well-known researches in this field before. *Bonacich* et al. [17] proposed an algorithm using degree centrality [18–21] to evaluate the importance of nodes for the first time in 1972, seen from which the bigger degree centrality is, the more important a node is. In addition, *Brin* and *Page* [22] presented a *PageRank* algorithm where a Web page's rank in search results is determined by the number of other pages link to it. Subsequently, methods based on *betweenness* [23, 24], *closeness* [25], *eigenvector* [26] and other metrics appeared gradually. Almost all these methods model software structure as complex networks, suggesting that a large number of complex links among nodes make nodes noteworthy. Besides, many new algorithms also have been proposed gradually, such as *HITS* [27], *LeaderRank* [28]], *NodeRank* [29]. Such algorithms are all based on random-walk model. They take connectivity among nodes and the importance of neighborhood nodes into consideration to determine the importance of nodes. And most of them have been applied in undirected and unweighted network, which still has some certain limitations.

Mapping the software structure or execution traces to complex networks, on which the mining of crucial nodes have certain effects and advantages. However, relevant methods are only designed from a static point of view to analyze software network and to measure the importance of nodes through dependency and connectivity among nodes, such as betweenness, in-degree, out-degree [30]. What's more, in the process of constructing the network model, these will ignore some dynamic characteristics of the software in the execution process, such as the order of functions, loop, recursion and other control forms, the length of function set. Therefore, if the software structure is analyzed only from the perspective of complex network, dynamic behavior characteristics of software execution process couldn't be captured accurately [31].

Based on the above mentioned, we proposed a dynamic noteworthy functions mining (*DNFM*) algorithm to help evaluate functions from the perspective of dynamic behavior characteristics in software execution sequences. Firstly, we decompiled software, traced the change of stack during software execution process so as to obtain original software execution traces formed by a series of function addresses. Then, we modeled these original traces as execution sequences and simplified them by Repetitive patterns eliminating algorithm [32], and designed a pattern extraction (*PE*) algorithm to extract patterns from *SFS*. After that, we designed two metrics *inner-importance* and *inter-importance* which are aimed to measure noteworthiness of functions in *DNFM* algorithm, followed by the categorization of functions according to their noteworthiness. Finally, we did three groups of experiment, and compared the results of *DNFM* with the results of two classic algorithms *PageRank* and *DegreeRank*.

Methods

Relevant definitions

DNFM approach analyzes the characteristics of software mainly at function level. Since the execution process of software is mostly the mutual call procedure between functions, software execution traces can be modeled as function call sequences expressed as follows.

$$S =< S_1 \dots S_i \dots S_n > (1 < i < n) \quad (1)$$

In S, S_i represents a feature vector including all information of a function in software execution process, which is expressed as $< Flg, Fname >$. Here, Flg is the entrance-exit flag to marked functions. As the function is a caller, Flg can be denoted as E, otherwise be X. $Fname$ is the name of function. These processes are shown in Fig 1(a) and 1(b) in more detail.

The sequence S is composed of a series of continuous marked functions orderly. Thus, S can accurately reflect the actual running process of software, especially the order of function calling and the relationship between different functions. As shown in Fig 1(c), the original traces are modeled as complex network, the weight on edges are only equal to the times of function addresses with flag E appearing in traces. Seen from the network, the actual process of function calling is not clear yet.

In S, there exist many execution-sequence-pattern (*ESP*), which are defined as follows.

Definition 1 ***ESP (execution-sequence-pattern)*** *ESP* for function call sequences is defined as $P(< Si, \dots, Sj >)$ where Si and Sj are the elements derived from sequence S, satisfies: $Si.Flg = E$, $Si.FName(fi) = Sj.Fname(fj)$, $Sj.Flg = X$. Besides, all flags and functions between Si and Sj are in pair, for instance, sequence segments like $< E, A >, < E, B >, < E, A >, < X, A >, < X, B >, < X, A >$ is an *ESP*, but $< E, A >, < E, B >, < E, A >, < X, A >$ isn't, although it meets the condition. Besides, one function call sequence can own many different *ESP*, and each *ESP* can appear many times in S.

In practice, the relationship between different *ESP* is very complex. Due to loops, an execution trace may contain repeated interesting patterns. What's more, different function call relations would result in different relationships between *ESP*. Suppose there exist $ESP_1 = Sp, \dots, Sq(p < q)$ and $ESP_2 = Ss, \dots, St(s < t)$ in S, there may exist three relationships between these two *ESP* which are shown below.

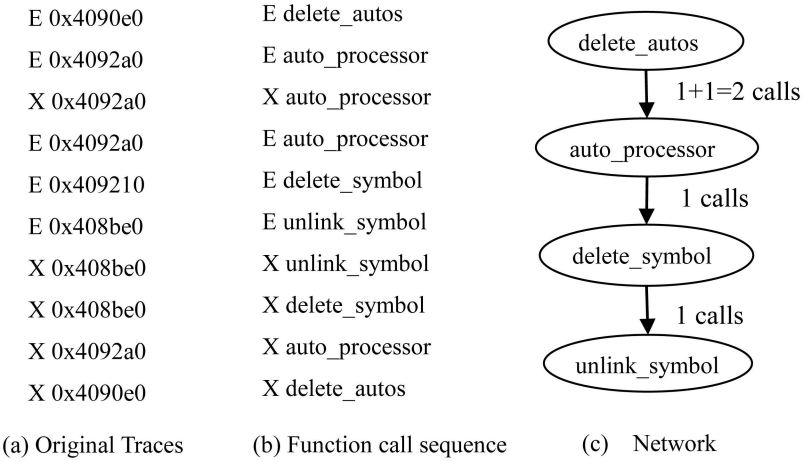

Fig 1. Modeling original Traces as function call sequence and network.

doi:10.1371/journal.pone.0173244.g001

1. As $S = S1, \ldots, ESP_1, ESP_2, \ldots, Sn$, when $ESP_1 = ESP_2$, ESP_1 and ESP_2 are reciprocal **Repetitive-pattern**.
2. As $S = S1, \ldots, ESP_1, \ldots, Sn$, when $ESP_1 \supset ESP_2$, ESP_1 and ESP_2 are reciprocal **Contain-Pattern**.
3. As $S = S1, \ldots, ESP_1, ESP_2, \ldots, Sn$, when $ESP_1 \neq ESP_2$, ESP_1 and ESP_2 are reciprocal **Parallel-pattern**.

Let $I < f1, f2, \ldots, fm >$ be a set of functions which has appeared in S. When S is simplified by the Repetitive patterns eliminating algorithm, *SFS* is obtained. According to the definition of *Frequency* of function and *Length of* ESP, two indexes *inner-importance* and *inter-importance* for each function in set I can be calculated, which are used to measure the noteworthiness of functions. The related definitions are as follows.

Definition 2 ***Frequency*** Frequency indicates how many times an *ESP* or a function occurs in the simplified function call sequence. For an *ESP*, it is denoted as $F(p)$, while for a function, it is denoted as $F(f)$.

Definition 3 ***Length of ESP*** Length of *ESP* is the number of vectors in a *ESP*, such as, the length of pattern $P(< S1, S2 \ldots, Sm >)$, which is denoted by $L(P)$, is m.

Definition 4 ***inner-importance (IN)*** *IN* means the probability of function *fi* which appears in *SFS*. It is defined as follows.

$$IN(f_i) = \frac{F(f_i)}{L(SFS)} \tag{2}$$

Here, $F(fi)$ is the frequency of function *fi*, and $L(SFS)$ is the length of simplified function call sequence *SFS*.

Definition 5 ***inter-importance (IT)*** In *S*, the function *fi* may appear in many different *ESPs* such as *P*1, *P*2, *P*3. . .*Pm*, and each *ESP* may appear many times. Thus, the *inter-importance* of each function can be defined as below.

$$IT(f_i) = \frac{\sum_{i=1}^{m}((F(P_i) * L(P_i)))}{L(SFS) * L(SFS)} \tag{3}$$

Here, $F(pi)$ is the frequency of *Pi*, and $L(pi)$ is the length of *Pi*, and $L(pi)$ is the length of *Pi*. Accordingly, $L(SFS)$ is the length of simplified function call sequence *SFS*.

Definition 6 ***Noteworthiness*** The noteworthiness of function $D(fi)$ is determined by the combination of its *inner-importance* and *inter-importance*, which is defined as below.

$$D(f_i) = \sqrt{(IN(f_i))^2 + IT(f_i))^2} \tag{4}$$

Definition 7 ***Shared-node rate*** *Shared-Node* rate describes the possibility that a node in set ρ occurs in set σ or τ. Let the number of elements in them all equate N, then *Shared-Node* rate equals the number of nodes shared by ρ and $\sigma \cup \tau$ divided by N, which is represented as follows.

$$\Psi(\rho, \sigma, \tau) = \frac{\mid \rho \cap (\sigma \cup \tau) \mid}{N} \tag{5}$$

Similarly, the ***Node similarity*** can be defined as below.

$$\Gamma(\rho, \sigma) = \frac{\mid \rho \cap \sigma) \mid}{N} \tag{6}$$

PageRank and DegreeRank

PageRank algorithm was created to rank web pages based on the number of other web pages linking to it. The *PageRank* of one web page is defined as follows.

$$PageRank(p_i) = \frac{1-q}{N} + q \sum_{p_j \in M(p_i)} \frac{PageRank(p_j)}{L(p_j)} \tag{7}$$

In formula (7), $p_1, p_2, \ldots, p_N$ denote a series of pages, $M(p_i)$ is the set of pages which link to page p_i, and $L(p_j)$ is the set of pages to which pape p_i links. N is the number of total pages, and $d(0 < d < 1)$ is a decay factor. In addition, q denotes teleporting, which means that the user maybe skip to another random web page from the current web page with a very low probability, and there is no hyperlink between the two web pages. Considered that the user couldn't skip from the current web page to the random web page directly, q is designed to describe the probability all out of pure mathematics sense.

The *DegreeRank* algorithm considers that the bigger the degree of note is, the more important the node is. Suppose there is an undirected graph which is formed of g nodes. The degree of node v_i C_D is the number of links between v_i and nearest neighborhood nodes, which is defined as below.

$$C_D(N_i) = \sum_{j=1}^{g} x_{ij} (i \neq j) \tag{8}$$

Where $C_D(N_i)$ is the degree of v_i, $\sum_{j=1}^{g} x_{ij}$ is the number of links between v_i and nearest neighborhood nodes except the links between vi and itself.

Framework of DNFM approach

DNFM approach can capture the dynamic behavioral characteristics of software execution process effectively, and in the meanwhile it can consider the properties of nodes and patterns in a function call sequence. The whole process for *DNFM* is shown in Fig 2. At first, we decompiled

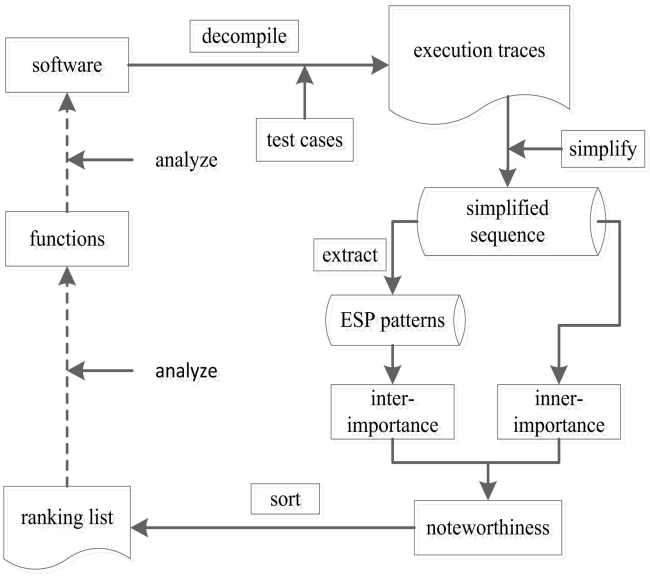

Fig 2. Framework of *DNFM* approach.

doi:10.1371/journal.pone.0173244.g002

the software by different test cases to obtain the software execution traces, and mapped them to function call sequences. In order to reduce the impact result from loop structures and respective patterns on *DNFM* algorithm, we applied the Repetitive patterns eliminating algorithm to simplify the function call sequence. After that, we extracted patterns from *SFS*. Accordingly, we set two metrics *inner-importance* and *inter-importance* combined to measure the noteworthiness of functions. At last, we sorted out functions by the noteworthiness and formed a function ranking list.

Repetitive patterns eliminating algorithm

In general, the function call sequence is very complex and numerous. Due to loops, a trace can have repeated patterns. That's to say, this means that there exist a large number of repetitive patterns, which will waste a lot of time during algorithm execution process. In order to improve the efficiency of the algorithm and make the sequence clearer, it is essential to delete continuous repeated patterns in *S*. The pseudo code of the Repetitive patterns eliminating algorithm [24] is shown as below.

Algorithm 1: Repetitive patterns eliminating algorithm

Inputs: S;
Output: the simplified function call sequence (SFS);
1. **for** all ESP in S
2. **if** exists m Repetitive patterns p, p_1, p_2, ..., p_{m-1} in S
3. Delete (p_1, p_2, ..., p_{m-1})
4. $p \rightarrow SFS$
5. **end if**
6. **end for**

In Line 1, the algorithm scans the original function call sequence *S* in searching all continuous repeated patterns. In Line 2 to Line 3, if there exist continuous repeated *ESP*, the algorithm will judge whether all flag have been matched so as to ensure all functions' entrance-flag are matched to its corresponding exit-flag, and then the repeated patterns are deleted. As shown in Line 4, an *ESP p* is left for *SFS*. At last, the *SFS* is generated.

Suppose a function call sequence is shown as Fig 3(a), there exist obviously repetitive patterns because the *ESP* < (*E ngx_regex_init*), (*Xngx_regex_init*) > is executed three times continuously. Repeated *ESP* would waste a lot of time during algorithm execution process and increase noteworthiness of function *ngx_regex_init*, we need to delete those repeated *ESP*. Thus we obtain the simplified function call sequence which is shown as Fig 3(b).

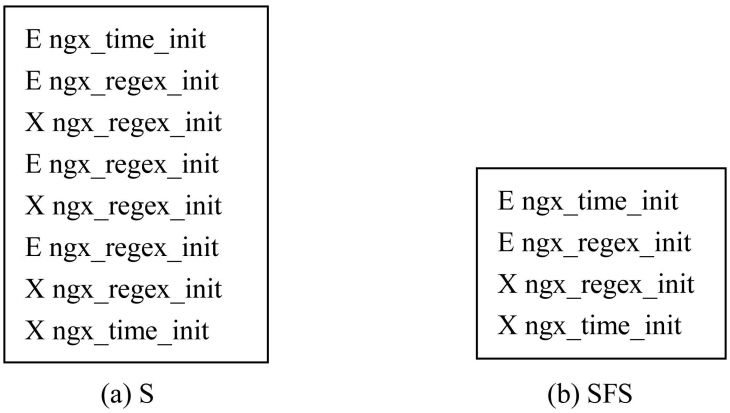

Fig 3. An example about simplifying *S* to *SFS*.

doi:10.1371/journal.pone.0173244.g003

Pattern Extraction algorithm

Seen from the definitions of inner-importance and inter-importance, we know that the noteworthiness of functions is related to *ESP*, including its frequency $F(P)$ and its length $L(P)$. However, the elements in *SFS* are mainly function vectors like $< Flg, Fname >$, thus in this section, we designed a Pattern Extraction algorithm to extract all patterns from *SFS*, of which the pseudo codes are shown as follows.

Algorithm 2: Pattern Extraction algorithm

```
Inputs: SFS
Output: ESP set P_set
1. for each vector S_i in SFS
2.   if (S_i.flag == 'E')
3.     for each vectors S_j (i < j) in SFS
4.       if (S_j.flag == 'X' and S_i.Fname == S_j.Fname)
5.         if (checkEX(Si, Sj)) //check whether flags of entrance-exit are
matched
6.           ESP ← P(S_i, S_j)
7.         end if
8.         if (CheckPattern(ESP, P_set)) //check whether there exist the same
pattern
9.           F(P)++; //frequency of the pattern adds 1
10.        else
11.          add_pattern (ESP, P_set); // save the pattern in P_set
12.        end if
13.      end if
14.    end for
15.  end if
16.end for
```

Line 1 to Line 3 of algorithm 2 traverse the elements in the whole simplified function call sequence. For a vector Si in *SFS*, if its entrance-exit flag is in pairs and correspond function name S_j. *Fname* is the same as the *Fname* of S_i, from which it can be judged that the sequence between S_i and S_j can form an *ESP*, these processes are shown from Line 4 to Line 6. Then, in Line 8 to Line 9, as the pattern exists in P_{set}, the frequency of the pattern increases one. Otherwise, the pattern *ESP* is put into P_{set} in Line 11. In the whole process, each vector in *SFS* can be checked and all patterns can be obtained to form P_{set}.

DNFM algorithm

During the above process, the simplified function call sequence *SFS* and *ESP set* P_{set} are obtained. After that, the *inter-importance* and *inner-importance* are computed and the *Noteworthiness* for each functions in *I* is got as well. Then, the noteworthiness of functions can be sorted by its *Noteworthiness*. Here, we propose a *DNFM* algorithm to conduct it. The pseudocode is described as follows.

Algorithm 3: DNFM algorithm

```
Inputs: SFS; P_set; I
Output: function rank list
1. for each f_i in I
2.   for each S_i in SFS
3.     if (S_i.Fname == f_i)
4.       F(f_i)++;
5.       F(p_i)++;
6.     end if
7.   end for
8.   INi = calculateIN(F(f_i), L(SFS));
9.   for each p_i in P_set
```

```
10.    if(f_i in p_i)
11.      Lt+ = F(p_i)*L(p_i)
12.    end if
13.  end for
14.  IT_i = calculateIT(L_t, L(SFS));
15.  D(f_i) = Calculate(IN_i, ITi);//calculate the Noteworthiness
16. end for
17. function rank list ← Sort(Noteworthiness);//sort functions by
Noteworthiness
18. return function rank list;
```

First, in order to calculate the inner-importance of each function, the algorithm analyzes each vector in *SFS* and counts the frequency of each function and its corresponding patterns, which are shown in Line 4 and Line 5. Then, the algorithm calculates the *inner-importance* for each function according to the definition 4 in Line 8. Suppose that there is a function f_i whose frequency is 6 according to the execution result of the algorithm and the length of *SFS* is 100, thus, the IN_i should be 6 divided by 100 on the basis of formula of *inner-importance*.

Then the *inter-importance* is computed. For each function *f* in set *I*, there might exist more than one *ESP* whose frequency and length are different. Assume that function f_i in *ESP* p_1, p_2, whose length and frequency are 5, 1 and 10, 2 respectively, thus IT_j is equal 0.0025 according to the definition 5 in Line 14. *IT* is actually an accumulative value. When the algorithm scans a new *ESP*, the amount of *IT* might be accumulated until all *ESP* have been found.

After that, the *DNFM* algorithm calculates the *Noteworthiness* for each functions by combining its *inner-importance* and *inter-importance* in Line 15. Finally, all functions can be sorted by *Noteworthiness* in line 17, thus we can obtain the top-k noteworthy functions in function rank list. It is noted that the higher a function ranks, the more noteworthy a function is.

Results

Objects and data

In this section, three software are used for the experiments. (1) Nginx- a lightweight web server reverse proxy server and e-mail (imappop3) proxy server using bsd-like agreement. (2) Deadbeef-a good music player which can play cue, mp3, ogg, flac, ape music file and so on. (3) Cflow- a tool to analyze a collection of *C* source files and print a graph, charting control flow within the program. We decompile the software and trace the change of stack after processing different operations on them so as to obtain some complete software execution traces from three different software. Accordingly three algorithms including *DNFM*, *PageRank* and *DegreeRank* were performed on the model established by these complete software execution traces for analyzing complex software structure and finding most noteworthy functions.

Comparative analysis

In this part, we compare the *DNFM* algorithm with *PageRank* and *DegreeRank*, the two classic algorithms on mining important nodes based on complex network. Thus, we perform the three algorithm on the three open source software *Nginx*, *Deadbeef* and *Cflow*, and do the contrast analysis on experiment results.

The experiments are conducted on *64 bit Windows 7 ultimate*, *Xeon CPU E5-2603 @1.80GHz*, *16G* of *RAM* and *Ubuntu14.04*. The length of original data sequence (LOS) and simplified sequence (LSS), and time consumption(TC) for removing adjacent repetitive patterns in experiments are listed in Table 1.

In Table 1, the Repetitive patterns eliminating algorithm is effective to eliminate the adjacent repetitive patterns, however, it is a waste of time. As the software execution sequence

Table 1. LOS, LSS and TC in each trial for different software.

	Nginx	Deadbeef	Cflow
LOS	6450	13048	7276
LSS	2680	5110	3914
TC	106.336s	461.275s	224.787s

doi:10.1371/journal.pone.0173244.t001

length increases, execution time becomes longer. Despite of this, what we concerned about is to mine dynamic noteworthy functions efficiently.

Before performing *PageRank* and *DegreeRank* algorithm, the software execution traces need to be modeled as complex network $G = < V, E, W >$. If there exists an edge between node V_i and V_j, the weight is assigned as 1, otherwise is 0. The networks are similar with the network displayed in Fig 4. What's the different are the number of nodes and edges for different software networks. Here, Table 2 gives these parameters for different software complex networks.

Based on software complex network and function call sequence, the functions were ranked, which are shown in Tables 3, 4 and 5. Here, we treat the result of *DNFM* algorithm as bases, and list top 15 functions compared with the top 15 functions in the ranking list got by *PageRank* and *DegreeRank* algorithm. ''–'' means the function is not in the top 15 function list of *PageRank* or *DegreeRank*. The number represents the order of a function in top 15 function list of *DNFM*.

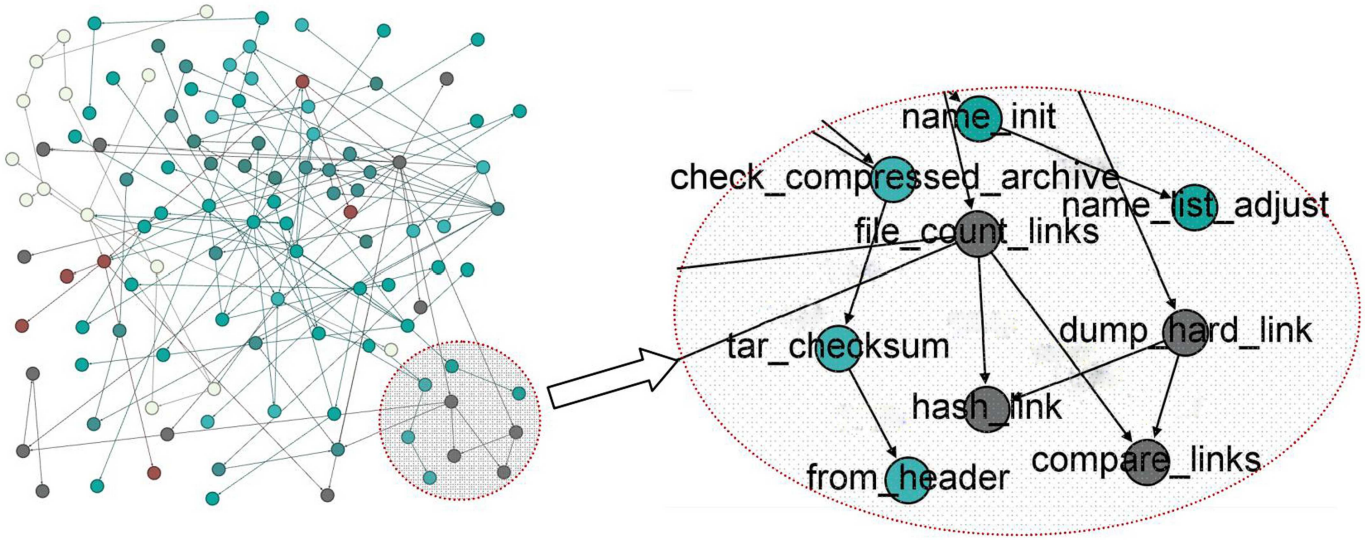

Fig 4. Software Complex Network.

doi:10.1371/journal.pone.0173244.g004

Table 2. The parameters of software complex networks.

	Nginx	Deadbeef	Cflow
Nodes	244	162	100
Edges	517	436	144

doi:10.1371/journal.pone.0173244.t002

Table 3. Results of three algorithms performed on *Nginx*.

Fname	Noteworthiness	DNFM	PageRank	DegreeRank
ngx_palloc	0.185821	No.1	No.1	No.9
ngx_pcalloc	0.086567	No.2	No.2	No.2
ngx_array_push	0.081343	No.3	No.5	No.3
ngx_pnalloc	0.045522	No.4	No.10	-
ngx_http_merge_locations	0.032836	No.5	-	No.7
ngx_array_init	0.029104	No.6	No.8	No.10
ngx_strlow	0.026119	No.7	-	-
ngx_hash_key	0.022388	No.8	-	-
ngx_conf_read_token	0.021642	No.9	-	-
ngx_http_merge_servers	0.020896	No.10	-	No.8
ngx_hash_add_key	0.020149	No.11	-	-
ngx_alloc	0.017164	No.12	No.7	-
ngx_conf_handler	0.013437	No.13	-	No.11
ngx_cpystrn	0.013433	No.14	No.13	-
ngx_http_add_variable	0.010448	No.15	No.9	No.14
Node similarity (Γ)			8/15	8/15
Shared-Node Rate (Ψ)			10/15	

doi:10.1371/journal.pone.0173244.t003

In Table 3, the experiment results show that there exist 8 functions which appeared in two top 15 function lists of *DNFM* and *PageRank* and 8 functions which appeared in two top 15 function lists of *DNFM* and *DegreeRank*, in which the *Share-node rate* for *DNFM* algorithm is equal to 10/15. As is shown in Table 4, for *Deadbeef*, there exist 9 functions which appeared in the experiment results of *DNFM* and *PageRank*, and 9 functions which appeared in the experiment results of *DNFM* and *DegreeRank*, in which the *Share-node rate* reaches 13/15. As the experiment result of *Cflow* is shown in Table 5, there exist 5 functions which appeared in

Table 4. Results of three algorithms performed on *Deadbeef*.

Fname	Noteworthiness	DNFM	PageRank	DegreeRank
mutex_lock	0.168689	No.1	No.2	No.6
mutex_unlock	0.167906	No.2	No.1	No.10
pl_lock	0.081409	No.3	No.3	No.4
pl_unlock	0.081409	No.4	No.4	No.3
conf_unlock	0.060274	No.5	No.5	No.12
conf_lock	0.060274	No.6	No.6	No.8
conf_get_str_fast	0.045010	No.7	No.9	No.15
conf_get_int	0.025832	No.8	-	No.7
plt_unref	0.015656	No.9	-	-
plt_ref	0.015656	No.10	-	-
plug_get_output	0.014873	No.11	-	-
tf_compile_plain	0.013699	No.12	No.12	-
conf_get_str	0.009393	No.13	-	No.13
conf_set_str	0.009393	No.14	No.10	-
tf_compile_field	0.009002	No.15	-	-
Node similarity (Γ)			9/15	9/15
Shared-Node Rate (Ψ)			13/15	

doi:10.1371/journal.pone.0173244.t004

Table 5. Results of three algorithms performed on *Cflow*.

Fname	Noteworthiness	DNFM	PageRank	DegreeRank
nexttoken	0.139499	No.1	No.4	No.2
get_token	0.094022	No.2	-	-
yylex	0.094021	No.3	-	No.9
tokpush	0.094021	No.4	-	-
hash_symbol_hasher	0.072049	No.5	No.10	-
hash_symbol_compare	0.065406	No.6	No.11	-
lookup	0.063362	No.7	No.12	No.8
putback	0.043945	No.8	-	No.15
ident	0.037813	No.9	-	-
get_symbol	0.023505	No.10	-	-
add_reference	0.021972	No.11	-	-
cleanup_stack	0.021972	No.12	-	-
reference	0.017374	No.13	-	-
expression	0.014308	No.14	-	No.12
mark	0.014308	No.15	No.14	-
Node similarity (Γ)			5/15	5/15
Shared-Node Rate (Ψ)			9/15	

doi:10.1371/journal.pone.0173244.t005

experiment results of *DNFM* and *PageRank* and 5 functions which appeared in experiment results of *DNFM* and *DegreeRank*, and the *Shared-node rate* of *Cflow* equals 9/15. The *Node similarity* of each software is all lower than its *Shared-node rate*. All the above experiment data prove that *DNFM* algorithm based on software function call sequence has more advantages than the algorithms based on complex software network in finding noteworthy functions. Compared the functions obtained by *PageRank* with those by *DegreeRank* algorithm, it can be found that functions got by *DNFM* algorithm contain those which can be got by *PageRank* algorithm (such as, functions like *ngx_palloc*, *ngx_http_merge_locations* and *ngx_conf_handler* in *Ngnix*, *mutex_lock*, *mutex_unlock*, *conf_unlock* and *plt_unref* in *Deadbeef*, and *tokpush*, *get_symbol*, add_reference and expression in *Cflow*) but cannot be got by *DegreeRank* algorithm, or vice versa. Therefore, the method proposed in this paper, for it combines the advantages of those classic methods as *PageRank* algorithm and *DegreeRank* algorithm, is of higher efficiency and accuracy, and its results are more representative.

Discussion and conclusions

In this paper, by taking functions as nodes and function calling as the order, we mapped software execution traces as function call sequences, and proposed a *DNFM* algorithm to find noteworthy functions in software sequences by analyzing those function call sequences. Firstly, we exploited an *Eliminate Repetitive patterns* algorithm to simplify initial function call sequences so as to reduce the influence from ring structure on mining noteworthy functions, and then generated patterns set from *SFS* by a designed pattern extraction algorithm. Secondly, after considering the frequencies and lengths of relevant *ESPs* and functions, we introduced two important evaluation indicators, because in the process of experiment, what the most important is to evaluate the impotence of each function. Thus we defined a new index *Noteworthiness* to measure the noteworthiness of functions. Finally, we compared *DNFM* with another two classic algorithms, namely, *PageRank* and *DegreeRank* algorithms which are conducted on complex network. The complex network weakly represents the software executions. It cannot reflect the sequence of function calls, which only describes whether there exists

relationships between two functions. Comparing the results from these traditional algorithms mostly applied on complex network, the proposed model and method in this paper are much suitable for software analysis. The results also verify that the *DNFM* algorithm can identify noteworthy functions most effectively, and that the *shared-Node rate* of *DNFM* algorithm is highest in different software, even though the *Node similarity* between them is lower. For instance, the *Node similarity* of software *Cflow* is about 5/15 = 33.3%, but the *Shared-Node rate* achieves 9/15 = 60%. All of these prove that the *DNFM* algorithm can work on different software dynamic execution process and can identify noteworthy functions successfully, effectively and precisely. However, it should be noted that there is a disadvantage of the *DNFM* algorithm, that is, it will waste a lot of time in simplifying mega function call sequence, which needs to be solved in the near future.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 61472341, No. 61572420, the Natural Science Foundation of Hebei Province P. R. China under Grant No. F2014203152, No. F2016203330 and the Graduate innovative Foundation of Hebei Province under Grant No. 2016SJBS010. The authors are grateful to the valuable comments and suggestions of the reviewers.

Author Contributions

Conceptualization: BZ YQW.

Data curation: BZ.

Formal analysis: BZ GYH.

Funding acquisition: GYH HTH JDR.

Investigation: BZ YQW.

Methodology: BZ YQW.

Project administration: BZ JDR.

Resources: BZ YQW.

Software: BZ YQW.

Supervision: GYH JDR.

Validation: BZ.

Visualization: BZ.

Writing – original draft: BZ YQW.

Writing – review & editing: JDR.

References

1. Wang BY, Lü JH. Software networks nodes impact analysis of complex software systems. Journal of Software. 2014; 24(12):2814–2829. doi: 10.3724/SP.J.1001.2013.04397
2. Gao ZK, Fang PC, Ding MS, Jin ND. Multivariate weighted complex network analysis for characterizing nonlinear dynamic behavior in two-phase flow. Experimental Thermal and Fluid Science. 2015; 60:157–164. doi: 10.1016/j.expthermflusci.2014.09.008

3. Gao ZK, Yang YX, Fang PC, Zou Y, Xia CY, Du M. Multiscale complex network for analyzing experimental multivariate time series. Europhysics Letters. 2015; 109(3):30005. doi: 10.1209/0295-5075/109/30005
4. Gao ZK, Cai Q, Yang YX, Dang WD, Zhang SS. Visibility Graph from Adaptive Optimal Kernel Time-Frequency Representation for Classification of Epileptiform EEG. International Journal of Neural Systems. 2017; 27(4):1750005. doi: 10.1142/S0129065717500058 PMID: 27832712
5. Ren JD, Wu HF, Yin TT, Bai L, Zhang B. A Novel Approach for Mining Important Nodes in Directed-Weighted Complex Software Network. Journal of Computational Information Systems. 2015, 11(8): 3059–3071.
6. Wang L, Dong J, Ren JD. Critical node measuring method improved based on pagerank and betweenness. ICIC Express Letters, Part B: Applications. 2016, 7(11): 2381–2387.
7. Ren JD, Wang CY, Liu QS, Wang G, Dong J. Identify influential spreaders in complex networks based on potential edge weights. International Journal of Innovative Computing, Information and Control. 2016, 12(2): 581–590.
8. Watts DJ, Strogatz SH. Collective dynamics of 'small-world'networks. Nature. 1998, 393(6684): 440–442. doi: 10.1038/30918 PMID: 9623998
9. Barabási AL, Albert R. Emergence of scaling in random networks. Science. 1999, 286(5439): 509–512. PMID: 10521342
10. Albert R, Barabási A-L. Statistical mechanics of complex networks. REVIEWS OF MODERN PHYSICS. 2002; 74(1):47–97. doi: 10.1103/RevModPhys.74.47
11. Newman MEJ. The structure and function of complex networks. SIAM Rev. 2003; 45(2):167–256. doi: 10.1137/S003614450342480
12. Kitsak M, Gallos LK, Havlin S, Liljeros F, Muchnik L, Stanley HE, et al. Identification of influential spreaders in complex networks. Nature Physics. 2010; 6(11):888–893. doi: 10.1038/nphys1746
13. Basaras P, Katsaros D, Tassiulas L. Detecting influential spreaders in complex, dynamic networks. Computer. 2013; 46(4):24–29. doi: 10.1109/MC.2013.75
14. Bae J, Kim S. Identifying and ranking influential spreaders in complex networks by neighborhood coreness. Physica A Statistical Mechanics & Its Applications. 2014; 394(4):549–559. doi: 10.1016/j.physa.2013.10.047
15. Liu JG, Ren ZM, Guo Q. Ranking the spreading influence in complex networks. Physica A Statistical Mechanics & Its Applications. 2013; 392(18):4154–4159. doi: 10.1016/j.physa.2013.04.037
16. Gao S, Ma J, Chen Z, Wang G, Xing C. Ranking the spreading ability of nodes in complex networks based on local structure. Physica A Statistical Mechanics & Its Applications. 2014; 403(6):130–147. doi: 10.1016/j.physa.2014.02.032
17. Bonacich P. Technique for analyzing overlapping memberships. Sociological Methodology. 1972; 4 (4):176–185.
18. Chen D, Lü L, Shang MS, Zhang YC, Zhou T. Identifying influential nodes in complex networks. Fuel & Energy Abstracts. 2012; 391(4):1777–1787. doi: 10.1016/j.physa.2011.09.017
19. Liu Y, Wei B, Du Y, Xiao F, Deng Y. Identifying influential spreaders by weight degree centrality in complex networks. Chaos Solitons & Fractals. 2016; 86:1–7. doi: 10.1016/j.chaos.2016.01.030
20. Lan W, Zhou K, Feng J, Chi Z. Research on Software Cascading Failures. IEEE. 2010:310–314. doi: 10.1109/MINES.2010.214
21. Hou G, Wang X, Zhou K. Network Model Construction and Cascading Effect Analysis for Software Systems. Software Engineering. IEEE. 2012:9–12. doi: 10.1109/WCSE.2012.10
22. Brin BS, Page L. The anatomy of a large-scale hypertextual Web search engine. Computer Networks & Isdn Systems. 2012. doi: 10.1016/j.comnet.2012.10.007
23. Alsayed A, Higham DJ. Betweenness in time dependent networks. Chaos Solitons & Fractals. 2015; 72:35–48. doi: 10.1016/j.chaos.2014.12.009
24. Zhang XZ, Zhao GL, Lv TY, Yin Y, Zhang B. Analysis on Key Nodes Behavior for Complex Software Network. Information Computing and Applications, Springer Berlin Heidelberg.; 2012; 7473:59–66.
25. Du Y, Gao C, Chen X, Hu Y, Sadiq R, Deng Y. A new closeness centrality measure via effective distance in complex networks. Chaos. 2015; 25(3):440–442. doi: 10.1063/1.4916215 PMID: 25833434
26. Bonacich P. Some unique properties of eigenvector centrality. Social Networks. 2007; 29(4):555–564. doi: 10.1016/j.socnet.2007.04.002
27. Kleinberg JM. Authoritative sources in a hyperlinked environment. Journal of the ACM. 1999; 46 (5):604–632. doi: 10.1145/324133.324140

28. Lü L, Zhang YC, Yeung CH, Zhou T. Leaders in Social Networks, the Delicious Case. Plos One. 2011; 6 (6):e21202. doi: 10.1371/journal.pone.0021202 PMID: 21738620

29. Bhattacharya P, Iliofotou M, Neamtiu I, Faloutsos M. Graph-based analysis and prediction for software evolution. International Conference on Software Engineering. 2012; 8543(1):419–429. doi: 10.1109/ICSE.2012.6227173

30. Zhang XZ. Analysis on Dynamic Behavior for Open-source Software Execution Network. Computer Science. 2011; 38(10):242–248.

31. Cai KY, Yin BB. Software execution processes as an evolving complex network. Information Sciences An International Journal. 2009; 179(12):1903–1928. doi: 10.1016/j.ins.2009.01.011

32. Huang GY, Zhang B, Ren R, Ren JD. A novel approach to efficiently mine structural patterns from software execution sequence. Journal of Computational Information Systems. 2015; 11(3):1109–1119.