International Journal of Innovative
Computing, Information and Control
Volume **10**, Number **1**, February **2014**

 ISSN 1349-4198
pp. 1–10

FINDING KEY FRAGMENTS IN FUNCTION EXECUTION SEQUENCE BASED ON UTILITY PATTERN MINING

Jiadong Ren[1,2], Qingshan Liu[1,2,*], and Jun Dong[1,2]

[1]College of Information Science and Engineering
Yanshan University
[2]The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province
No. 438, Hebei Ave., Qinhuangdao 066004, P. R. China
jdren;dongjun@ysu.edu.cn; *:qsliuysu@gmail.com

Abstract. *Flowing the growth of software complexity, how to find critical components of software system efficiently and accurately becomes increasingly important for software maintenance and update. The available methods usually exploit information of software structure only, and do not conduct comprehensive analysis from different aspects. In this paper, a novel software analysis framework Key Fragments Mining (KFM) is proposed. It takes function execution sequence into account, and processes the sequence with methods of utility pattern mining. By considering both the static structure and dynamic sequence, our framework can obtain better result. Firstly, every function is assigned with a weight by exploiting the software system architecture. Secondly, key fragments are mined from the function execution sequence with the help of KFM by calculating the utility of every fragment. Experimental results show that KFM has better performance and can serve as an effective solution to the problem of mining key fragments from function execution sequences.*

Keywords: Software network; Function execution sequence; Key fragments; Utility pattern mining

1. **Introduction.** With the prosperity of software industry, both of the scale and complexity of software are increasing rapidly, which makes it very consuming to optimize and maintain software system. To deal with this issue, engineers try to find some critical functions from the program to yield twice the result with half the effort.

Due to the emphasis of overall construction, complex network has been treated as an effective way to analyze software system. Researchers try to map software into complex network by regarding software components (packages, classes, functions et al.) as nodes and taking interactions (inheritance, call et al.) between these components as directed or undirected edges [4], such networks are called as software network. With the scale of software growing sharply, how to analyze software with the achievement of complex network has attracted many attentions [5].

Valverde et al. [6] firstly introduced complex network into software system analysis. They take the class network in object-oriented software as research object and use undirected network to represent the structure of software. The results showed that the structure of software systems present very obvious "Small World" and "Scale Free" feature. Myers et al. further used directed network to represent the structure of the software system and analyzed a large number of open-source softwares. The results were consistent with the ones obtained by Valverde et al. Callaw et al. [7] used degree centrality as metric to evaluate the importance of a function. But their method only considered the local structure of the network, its performance was not outstanding. Freeman et al. [8] introduced betweenness centrality into evaluation mechanism, and nodes with large

betweenness centrality were treated as important ones. However, betweenness centrality as global metric needs to calculate the shortest paths from a node to all other nodes with considerable computational complexity. Based on PageRank, K. Inoue et al. [9] proposed ComponentRank to evaluate software components. They analyzed the actual use relations of components and the propagating of significance through use relations. Li Ding-wei et al. [10] represented object-oriented software with a weighted network model and defined IC indicator kits to measure the importance of functions. Huan Luo et al. [11] proposed a specified PageRank method called VertexRank to evaluate the significance of every function.

But in fact, only considering the importance of a function based on the software structure is not enough. When executing a program, some functions can run many times in different parts of the process, while others may only be executed once. In this scenario, we should consider the execution frequency of functions as another factor that has influence on the importance of functions, in premise of eliminating the influence of loop in the program. Besides, it often occurs that a single function may not play a very important role in the system, but it is part of a critical fragment of the program. In this case, it also worth paying attention to make sure the fragment functions well.

Utility pattern mining (abbreviated as UPM) emerges as an upgraded version of frequent pattern mining (abbreviated as FPM). It successfully overcomes the disadvantages that FPM may discover a large amount of frequent but low revenue patterns and lose the information on valuable patterns having low frequencies. Hong Yao et al. [12] firstly given the theoretical model and definition of UPM. Later, two algorithms, UMining and UMining_H [13], were proposed by the same authors to calculate high utility patterns. However, these methods do not satisfy the downward closure property of Apriori [14] and overestimate too many patterns. Liu et al. [15] proposed the anti-monotone property with TWU (transaction weighted utilization), and developed the TwoPhase algorithm by adapting Apriori to generate a complete set of candidates with high TWUs in the first phase. Erwin et al. [16] proposed the CTU-PROL algorithm that uses the TWU property with FP-growth. Mengchi Liu et al. [17] proposed an algorithm HUI-Miner for high utility itemset mining using a structure called utility-list. Junqiang Liu et al. [18] proposed a high utility itemset growth approach that works in a single phase without generating candidates. Cheng-Wei Wu et al. [19] proposed algorithm UP-Span to address the problem of mining high utility episodes from complex event sequences.

In order to mine key fragments from function execution sequences, every function is assigned with a weight calculated based on software structure. Considering the similarity of software network and Internet [11], and to balance the effect and resource consuming, we choose PageRank [20] as our criterion to estimate the importance of every function. A novel approach Key Fragments Mining (KFM) is proposed which is based on utility pattern mining. KFM finds key fragments by exploiting the information contained in system structure and function execution sequence. It is valuable for engineers to optimize software under a more comprehensive consideration. The major contributions of this paper are summarized as follows.

Firstly, system structure and function execution sequence are integrated together to analyze softwares.

Secondly, utility pattern mining is incorporated into software analysis and a novel framework is presented for mining key function fragments. An efficient algorithm named KFM (Key Fragment Mining) is proposed for mining key fragments in function execution sequences.

The rest of this paper is organized as follows. Section 2 gives formal definitions of our framework. Section3 is the detail of algorithm KFM. The experimental results on two

open source softwares are shown in Section 4. Conclusions and future work of our research are given in Section 5.

2. **Preliminaries.** Directed Function Call Network is constructed, abbreviated as DFCN. In DFCN, nodes represent functions, edges represent call relationships between functions, directed edge from function f_1 to function f_2 represents f_1 calls f_2.

Definition 2.1. ***FES (Function Execution Sequence)***
let N^+ be a set of time points and f be a function, $\varepsilon = \{f_1, f_2, ..., f_n\}$ is a finite set of functions. A function execution sequence $FES =< (f_1, T_1), (f_2, T_2), ..., (f_n, T_n) >$ is an ordered sequence of functions, where each function f_i associates with a time point $T_i \in N^+$ and $T_i < T_j$ for all $1 \leq i < j \leq n$.

Example 2.1. *For an execution flow in Figure 1, the function execution sequence is $FES = \langle (A,1), (B,2), (E,3), (C,4), (A,5), (B,6), (C,7), (D,8), (B,9), (A,10), (B,11), (C, 12), (E,13)\rangle$*

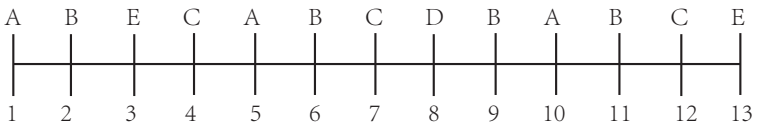

Figure 1: A simple example of function execution sequence

Definition 2.2. ***Function fragment***
A function fragment α is a non-empty ordered set of functions of the form $< f_1, f_2, ..., f_k >$ where function f_i is executed before the function f_j for all $1 \leq i < j \leq k$.

Example 2.2. *For example, $\alpha =< A, B, C >$ is a function fragment in Figure 1.*

For a function fragment $\alpha =< f_1, f_2, ..., f_k >$, the time interval $[T_s, T_e]$ is called the *occurrence* of α, if (1) α occurs in $[T_s, T_e]$, (2) the first function f_1 is executed at time T_s and the last function f_k is executed at time T_e. The set of all occurrences of α is denoted as $occSet(\alpha)$. For example, the set of all occurrences of $\alpha =< A, B, C >$ in Figure 1 is $occSet(< A, B, C >) = \{[5, 7], [10, 12]\}$.

Maximum Fragment Length (abbreviated as MFL) is a user specified parameter used to limit the maximum length of the fragment. Every occurrence $[T_s, T_e]$ of fragment α should satisfy $(T_e - T_s + 1) \leq MFL$.

Definition 2.3. ***Total utility of function execution sequence***
The total utility of a function execution sequence FES is defined as:

$$u(FES) = \sum_{i=1}^{n} u(f_i, T_i), (f_i, T_i) \in FES, \tag{1}$$

$u(f_i, T_i)$ is the weight of function f_i at T_i, n is the number of items in FES.

Example 2.3. *For example, Table 1 shows the weight of every function. The utility of function execution sequence depicted in Figure 1 is u(FES) = u(A, 1) + u(B,2)+ u(E, 3) + u(C, 4)+ u(A, 5) + u(B, 6) + u(C, 7) + u(D, 8) + u(B, 9) + u(A, 10) + u(B, 11) + u(C, 12) + u(E, 13) = (2+1+1+2+2+1+2+3+1+2+1+2+1) = 20.*

Table 1: Weights of functions

function	A	B	C	D	E
weight	2	1	2	3	1

Definition 2.4. ***Utility of a function fragment***
For a function fragment $\alpha =< f_1, f_2, ..., f_k >$*, the utility of the fragment is defined as:*

$$u(\alpha) = ((\sum_{i=1}^{k} u(f_i)) * n)/u(FES), f_i \in \alpha, \tag{2}$$

$u(f_i)$ *is the weight of function* f_i*,* n *represents the frequency of fragment* α *in the function sequence, i.e. the number of occurrences in* $occSet(\alpha)$*,* $u(FES)$ *is the total utility of function execution sequence.*

Example 2.4. *For example, the utility of fragment* $\langle A, B, C\rangle$ *in Figure 1 is* $u(\langle A, B, C\rangle) = (2 + 1 + 2) * 2/20 = 10/20$.

Let $\text{occ}(\alpha) = [T_s, T_e]$ be an occurrence of the fragment $\alpha =< f_1, f_2, ..., f_k >$, where each function $f_i \in \alpha$ associates with a time point $T_i \in [T_s, T_e]$ and $\text{occ}(\alpha)$ satisfies MFL. The fragment weighted utilization of α w.r.t. $\text{occ}(\alpha)$ is defined as:

$$\text{FWU}(\alpha, occ(\alpha)) = [\sum_{i=1}^{(k-1)} u(f_i, T_i) + \sum_{i=e}^{(s+MEL-1)} u(f_i, T_i)]/u(FES). \tag{3}$$

For example, when MFL=5, the FWU of the fragment $\alpha =< A, B, C >$ w.r.t. occurrence [5,7] is $FWU(\langle A, B, C\rangle, [5, 7]) = [u(A, 5)+u(B, 6)+u(C, 7)+u(D, 8)+u(B, 9)]/u(FES) = (2 + 1 + 2 + 3 + 1)/20 = 9/20$.

Definition 2.5. ***FWU (Fragment-Weighted Utilization of a fragment)***
Let $occSet(\alpha) = [occ_1, occ_2, ..., occ_{\text{k}}]$ *be the set of all the occurrences of* α*, where each occurrence* occ_i *satisfies MFL. The fragment-weighted utilization of* α *in a function execution sequence is defined as:*

$$\text{FWU}(\alpha) = (\sum_{i=1}^{k} FWU(\alpha, \text{occ}_i))/u(FES), \text{occ}_i \in occSet(\alpha). \tag{4}$$

Example 2.5. *For example, when MFL=3, the FWU of the fragment* $\alpha =< A, B >$ *with* $\text{occSet}(\alpha) = \{[1, 2], [5, 6], [10, 11]\}$ *is* $FWU(\langle A, B\rangle)$ *= { [u(A, 1)+ u(B, 2)+ u(E, 3)]+ [u(A, 5)+ u(B, 6)+ u(C, 7)]+ [u(A, 10)+ u(B, 11)+ u(C, 12)]}/u(FES)=14/20.*

Definition 2.6. ***HWUF (High Weighted Utilization Fragment)***
A fragment α *is called High Weighted Utilization Fragment (abbreviated as HWUE) if its FWU is no less than the minimum utility threshold (min_utility) specified by user.*

Definition 2.7. ***KF (Key Fragment)***
A fragment α *is Key Fragment (abbreviated as KF) if its utility* $u(\alpha)$ *is higher than the user-specified minimum utility threshold (min_utility), we also call it high utility fragment. Otherwise, the fragment is a low utility fragment.*

Theorem 2.1. ***FWDC (Fragment-Weighted Downward Closure property)***
Let α*,* β *be two different fragments, and* β *is prefixed with* α*. The Fragment-Weighted Downward Closure (abbreviated as FWDC) property states that if* $\text{FWU}(\alpha) < min\_utility$*,* β *is a low utility fragment.*

Proof: Let $occSet(\alpha) = [occ_1, occ_2, ..., occ_x]$, $\text{occSet}(\beta) = [occ'_1, occ'_2, ..., occ'_y]$. Because β is prefixed with α, $|occSet(\alpha)| \geq |occSet(\beta)|$. According to Definition 2.5, $\text{FWU}(\alpha) = (\sum_{i=1}^{x} FWU(\alpha, occ_i))/u(FES) \geq \text{FWU}(\beta) = (\sum_{i=1}^{y} FWU(\beta, occ'_i))/u(FES) \geq u(\beta)$. If

FWU(α) < *min_utility*, $u(\beta)$ < *min_utility*, it turns out that β is low utility fragment (Definition 2.7).

3. Mining key fragments in function execution sequence.

3.1. **Framework of our approach.** We try to mine key fragments of software based on two aspects: the function invocation relationship and the function execution sequence. Firstly, functions and function invocation relationships are extracted out from software source code, to obtain the directed function call network and function execution sequence. Secondly, PageRank is used to analyze the directed function call network to assign every function a weight. Because a function can be executed many times in a loop structure, it can cause huge interference when we mine key fragments. So the loop in function execution sequences should be removed and treated as executing once. Finally, key fragments are mined from function execution sequence based on the weight obtained in step two. The framework is shown in Figure 2.

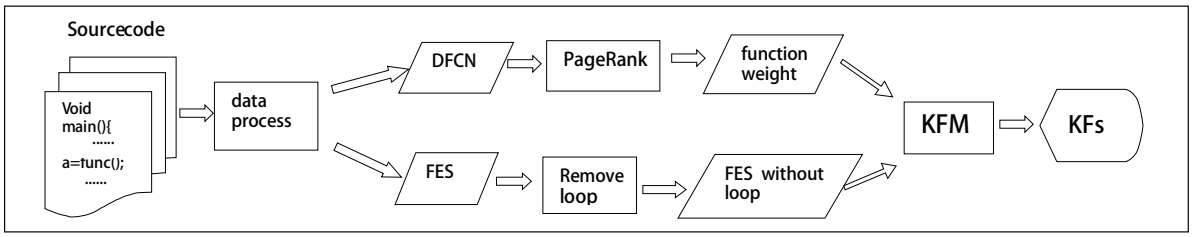

Figure 2: Framework of mining key fragments

3.2. **Algorithm for mining key fragments.** In order to obtain all the key fragments without omission, a prefix-growth paradigm based program KFM (Key Fragments Mining) is proposed to process function execution sequences. FWDC (Theorem2.1) is applied into the program to make the progress faster.

The algorithm scans function execution sequence once to evaluate every function node and finds the global potential fragments. Procedure of KF-Span is used to expand every potential fragment and screen out all the key fragments.

Algorithm 1 KFM: Key Fragments Mining

Input: (1)**FES**: function execution sequence;
(2)**min_utility**: minimum utility threshold;
(3)**MFL**: maximum fragment length.
Output: KF_Set: The complete set of key fragments;
Step 1. Scan FES once to find high utility 1- fragment and calculate their FWUs.
Step 2. **for each** function α **do**
Step 3. **If** (FWU(α) $\geq$ *min_utility*) **then**
Step 4. **If** ($u(\alpha) \geq$ *min_utility*) **then**
Step 5. add α into KF_Set
Step 6. KF-Span(α, MFL , min_utility) #call function KF-Span
Step 7. return KF_set

For each function α in the execution sequence, KFM calculates its FWU based on definition 2.5, and FWDC is firstly applied here (Line 3). If FWU(α) is less than *min_utility*, the algorithm will stop to evaluate next function, which means any fragment prefixed

Algorithm 2 KF-Span

Input: (1) α: potential fragment;
(2) **MFL**: maximum fragment length;
(3) **min_utility**: minimum utility threshold;
Output: KF_Set: The set of key fragments prefixed with α.

Step 1. get occSet of α
Step 2. **for each** occ(α) = [T_s,T_e] $\in$ occSet(α) **do**
Step 3. **for each** time point t between $[T_e + 1, T_s + MTD - 1]$ **do**
Step 4. NFS=$\{f \mid$ function f executed at time point $t\}$
Step 5. $\beta = \alpha$
Step 6. **for each** function $f \in NFS$ **do**
Step 7. $\beta = join(\beta, f)$
Step 8. **If** FWU(β) $\leq$ *min_utility* **then**
Step 9. **break**
Step 10. **If** u(β) $\geq$ *min_utility* **then**
Step 11. add β into KF_Set
Step 12. return KF_set

with α will not be processed in the future, and lots of meaningless calculations will be reduced. If FWU(α) is greater than *min_utility*, KFM will calculate the utility of α to determine whether it is a key fragment or not (Line 4). Then the algorithm processes fragments prefixed with α with KF-Span to find more KFs(Line 6). Table 2 shows the occurrences, FWUs and utilities of all 1-fragments in Figure 1 calculated by KFM when MFL= 3.

Table 2: Example of the first step of KFM

function	*occurrence*	*FWU*	*utility*
A	{[1,1], [5,5], [10,10]}	14/20	6/20
B	{[2,2], [6,6], [9,9], [11,11]}	18/20	4/20
C	{[4,4], [7,7], [12,12]}	14/20	6/20
D	{[8,8]}	6/20	3/20
E	{[3,3], [13,13]}	6/20	2/20

After a potential fragment α is passed to KF-Span, the occurrence set of α i.e. occSet(α) is obtained firstly. For each occurrence occ(α) = [T_s,T_e] in occSet(α), all functions executed between time interval $[T_e + 1, T_s + MTD - 1]$ are collected into the set NFS (Next Functions Set). Then every function f in the set NFS is concatenated to the end of α serially to form a new fragment β (Line 7). If the FWU of β doesn't satisfy *min_utility*, the program will not process fragments prefixed with β but continue to process next occurrence where FWDC is adopted again (Line 8). If FWU(β) is greater than *min_utility*, the utility of β will be calculated and evaluated (Line 10).

For example, with $\alpha =< A, B >$ w.r.t. $occ(\alpha) = [1, 2]$, MFL=4, the NFS of α is $NFS(\alpha, [1, 2]) = \{E, C\}$. Function E will be concatenated with α to form a new fragment $< A, B, E >$, then its FWU and utility is calculated and tested. After that, function C will be concatenated with $< A, B, E >$ and $< A, B, E, C >$ is generated as the new fragment to be evaluated.

4. **Experiment Analysis.** In this section, we evaluate the performance of the proposed algorithms. Experiments are performed on a computer with a 3.20 GHz Intel Core i5-3470 Processor with 4 gigabytes of memory, running 64 bit Windows 7 ultimate. All of the algorithms are implemented in Python. We get the directed function call network and function execution sequence with the help of *pvtrace* on ubuntu 14.04 LTS. The approach is tested with two open source software *tar* and *gzip*, and for each one we choose five most recent versions as our experimental subjects, the software source code is available from http://mirrors.ustc.edu.cn/gnu/. Three different rank criteria are analyzed and the efficiency of FWDC is tested.

4.1. **Analysis of function rank criteria.** Three different criteria are adopted to rank function nodes, such as *Degree*, *Betweenness* and *PageRank*. Figure 3 is part of the DFCN of software *tar* showing the detail about function *dump_file0* and *to_chars*. Table3 shows the top-10 functions of *tar-1.28* under different criterion.

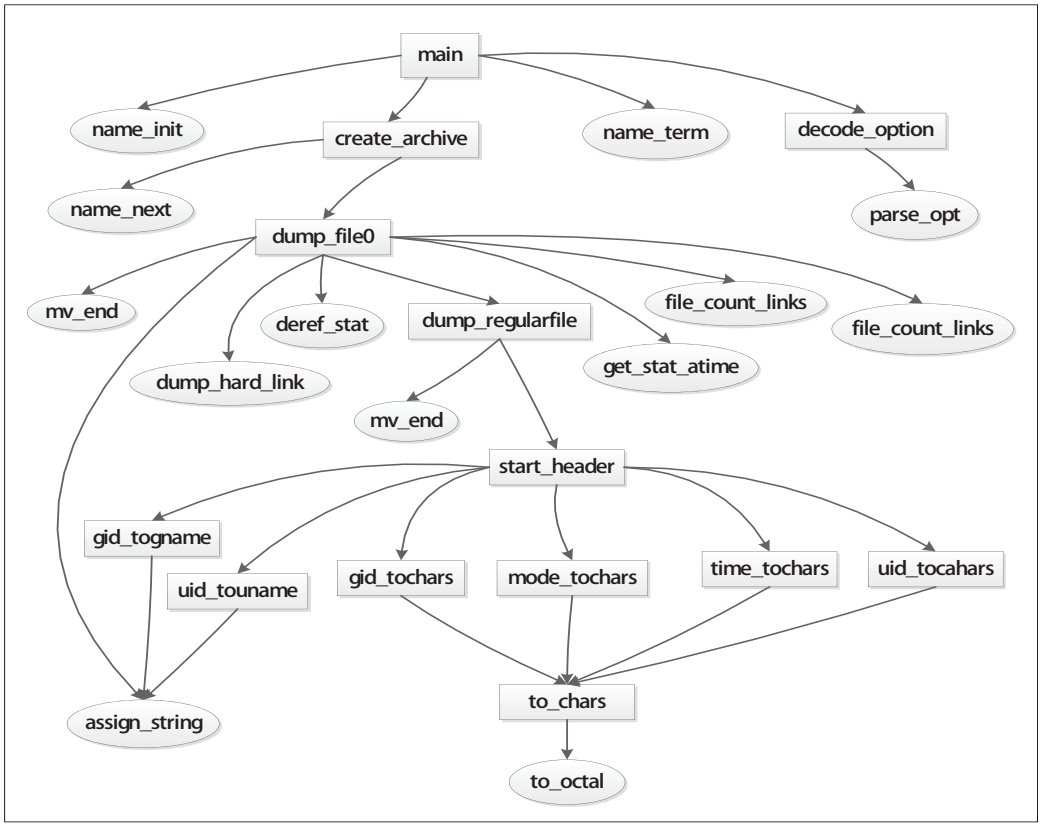

Figure 3: Part of DFCN of *tar-1.28*

Table3 shows that the results of these three criteria are different. The first function selected by Degree and Betweenness is *dump_file0*. From Figure 3 we can see that *dump_file0* has a large out-degree, which makes *dump_file0* exist on many paths. So *Degree* and *Betweenness* recognize it as the most important function. But when we try to optimize *dump_file0*, most of the work is refining its callees, so recognizing *dump_file0* as the most important function is meaningless. PageRank selects *to_chars* as the first function and the score of *to_octal* is very similar with *to_char*'s. Function *to_chars* has a large in-degree, which means it is called by many other functions, so optimize *to_chars* can generate much influence to the software. But it doesn't mean that only functions with large in-degree are

important, like function *to_octal*, its in-degree is one, but because it is called by *to_chars* which is very influential, the performance of *to_octal* also has a significant impact on the software. We can see that PageRank makes better use of the information of overall network structure. According to the analysis above, PageRank is chosen as the criterion to value function nodes in this paper.

Table 3: Top-10 functions of *tar-1.28* under different criteria

rank	*PageRank*	*Degree*	*Betweenness*
1	to_chars	dump_file0	dump_file0
2	to_octal	start_header	dump_regular_file
3	_gun_flush_write	create_archive	dump_file
4	gun_flush_write	dump_regular_file	create_archive
5	tar_copy_str	to_chars	start_header
6	assign_string	main	flush_archive
7	flush_archive	close_archive	gun_flush_write
8	checkpoint_run	_open_archive	find_next_block
9	sys_write_archive_buffer	tar_stat_destroy	_gun_flush_write
10	_transform_name_to_obstack	decode_options	finish_header

4.2. **Results Analysis of KFM.** The algorithm is tested on *tar* and *gzip*. The function execution sequences are generated by executing "*tar -cvf* " for *tar* and "*gzip -c*" for *gzip*. Table 4 shows characteristics of datasets used in experiments. The data of version 1.24, 1.25 and 1.26 of *tar* are similar, 1.27 and 1.28 are similar. For *gzip*, version 1.6, 1.5 and 1.3.13 are similar. It is shown in Table4 that the length of FES is reduced dramatically after eliminating the loop, which illustrates the importance of loop eliminating. To evaluate the performance of the proposed algorithms, we compare three versions of the algorithm named as follows. The baseline algorithm without strategy FWDC is denoted as KFM-basic. The algorithm adopted strategy FWDC is denoted as KFM-FWDC. The traditional frequent pattern mining algorithm is denoted as FPM.

Table 4: Information about the experiment datasets

$software$	Length of FES (original)	Length of FES (loop removed)	$\#Functions$	$\#edges$
$tar-1.24$	86499	374	227	177
$tar-1.28$	64284	406	242	191
$gzip-1.6$	35965	80	56	48
$gzip-1.4$	50645	79	55	47
$gzip-1.3.12$	20550	76	55	47

Figure 4 shows the number of candidates on *tar-1.28* under various minimum utility thresholds when the maximum fragment length equals to 4, and there is no candidate when the minimum utility threshold is greater than 0.2. Figure 5 shows the situation under different maximum fragment length when the minimum utility threshold equals to 0.2. As shown in these two figures, KFM-FWDC generates much fewer candidates than KFM-basic. The reason is that strategy FWDC effectively reduces the number of candidates by removing unpromising functions from the function execution sequence.

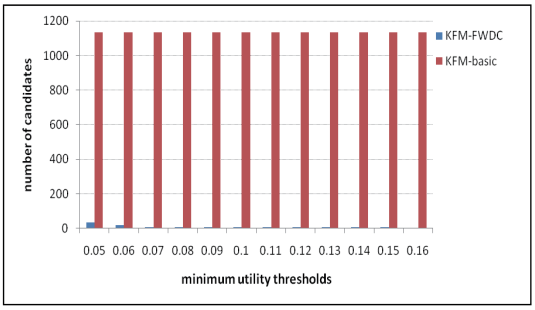

Figure 4: The number of candidates on tar-1.28 under different min_utility with MFL equals 4

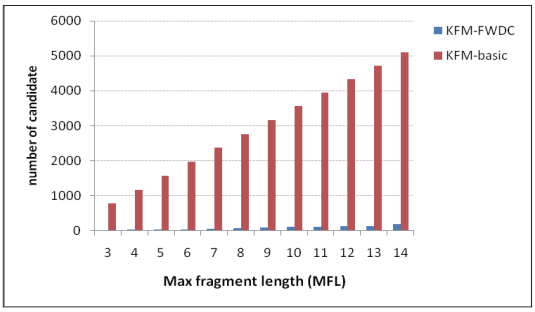

Figure 5: The number of candidates on tar-1.28 under different MFL with min_utility equals 0.2

Figure 6 shows the execution time of algorithms on *tar-1.28* under various maximum fragment lengths as minimum utility threshold equals to 0.2. Figure 7 is the execution time under different minimum utility threshold with maximum fragment length equals to 4. As shown in Figure 6 and Figure 7, KFM-FWDC runs about 20%-50% faster than KFM-basic, because the former algorithm produces much fewer candidates than the later. Figure 8 and Figure 9 illustrate the performance on *gzip-1.6*, which consistents with *tar-1.28*.

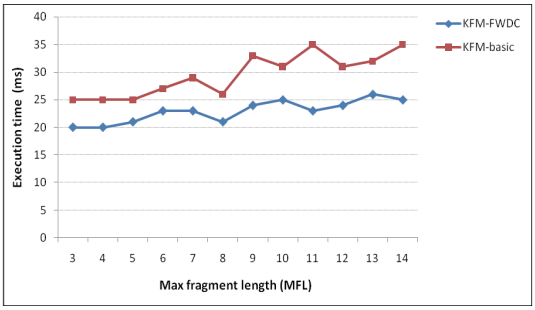

Figure 6: The execution time on tar-1.28 under different MFL with min_utility equals 0.2

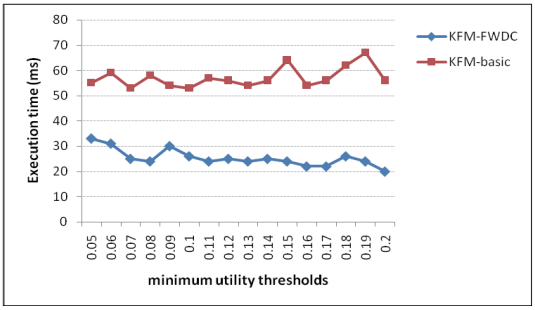

Figure 7: The execution time on tar-1.28 under different min_utility with MFL equals 4

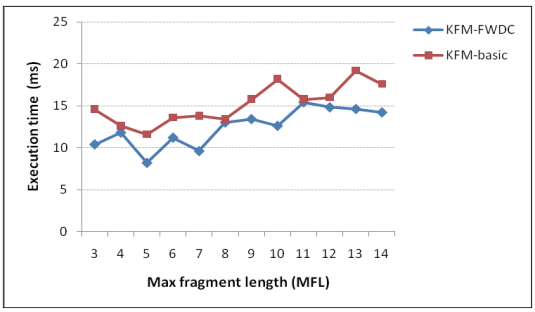

Figure 8: The execution time on gzip-1.6 under different MFL with min_utility equals 0.17

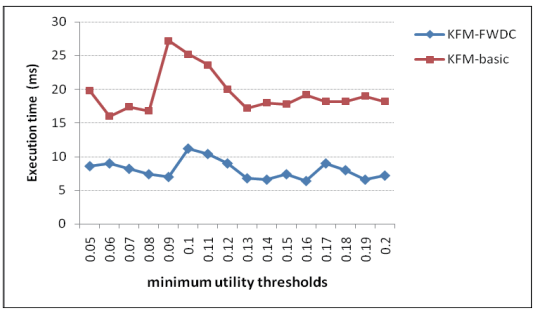

Figure 9: The execution time on gzip-1.6 under different min_utility with MFL equals 4

The key fragments found by KFM and FPM are analyzed. Figure 10 shows the total utility of top-n KFs which is calculated according to $\sum_{i=1}^{n} u(KF_i)$. Figure 11 shows the

average utility of top-n KFs which is calculated according to $\sum_{i=1}^{n} u(KF_i)/(\sum_{i=1}^{n} F(KF_i))$ where $F(KF_i)$ is the frequency of KF_i. The charts show that KFM can find more important fragments than FPM, as KFM considers more factors than FPM.

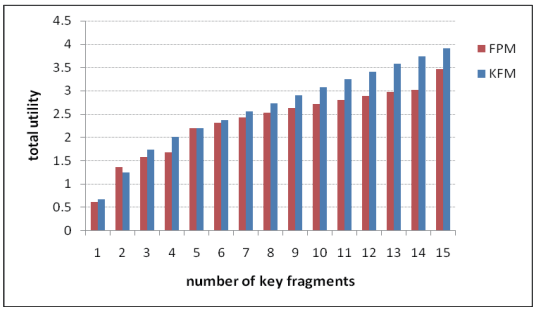

Figure 10: The total utility of top-n KFs found by KFM and FPM on tar-1.28

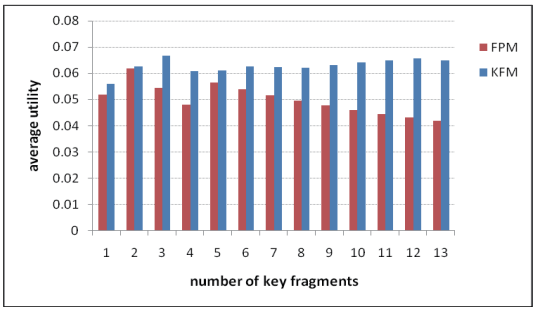

Figure 11: The average utility of top-n KFs found by KFM and FPM on tar-1.28

5. **Conclusions.** In this paper, a novel framework is proposed for mining key fragments in function execution sequences. It incorporates utility pattern mining into software analysis, and considers weight of functions and frequency of fragments when measuring the importance of fragments. Software system is modeled as Directed Function Call Network and Function Execution Sequence simultaneously, and then weights of functions are obtained through exploiting Directed Function Call Network. After that, KFM utilizes the weights to calculate utilities of every potential fragment in the function execution sequence whose loop has been removed. In order to accelerate the speed of mining complete set of key fragments, an effective strategy, namely FWDC (Fragment-Weighted Downward Closure property), is integrated with KFM. It not only reduces the number of candidates produced in the mining processes but also improves the performance of the mining task in terms of execution time. Experimental results on two open source softwares show that KFM with FWDC can find important fragments efficiently.

In the future, there are other methods which can be used to calculate the weight of functions more accurately, such as VertexRank, ComponentRank, LoopRank and so on. In addition, performance of the framework on very large datasets should be evaluated and better pruning strategy based on software characteristics may be proposed.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61170190, No. 61472341 and the Natural Science Foundation of Hebei Province China under Grant No. F2013203324 , No. F2014203152 and No. F2015203326. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

[1] Adilson E. Motter and Ying-Cheng Lai, Cascade-based attacks on complex networks, *Physical Review E*, vol.66, no.6, 2002.

[2] Tao Zhou, Bing-Hong Wang, Catastrophes in scale-free networks, *Chinese Physics Letters*, vol.22, no.5, pp.1072-1075, 2005.

[3] R. Pastor-Satorras, A. Vespignani, Immunization of complex networks, *Physical Review E*, vol.65, no.3, 2002.

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

[5] Jie Zhang, Changsong Zhou, Xiaoke Xu, Michael Small, Mapping from structure to dynamics: a unified view of dynamical processes on networks, *Physical Review E*, vol.82, no.2, 2010.

[6] S. Valverde, R.F. Cancho and R.V. Sole, Scale Free Networks from Optimal Design, *Europhysics Letters*, vol.60, no.4, pp.512-517, 2002.
[7] D. S. Callaw, M. E. J. Newman and S. H. Strogatez, Network robustness and fragility: Percolation on random graphs, *Europhysics Letters*, vol.85, no.25, pp.5468-5471, 2000.
[8] L. C. Freeman, Centrality in social network: Conceptual clarification, *Social Networks*, vol.1, no.3, pp.215-239, 1979.
[9] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, Component rank: relative significance rank for software component search, *Proc. of 25th IEEE International Conference on Software Engineering*, Washington, USA, pp.14-24, 2003.
[10] Li Ding-wei, Li Bing and Pan Wei-feng, Ranking the Importance of Classes via Software Structural Analysis, *Future Communication, Computing, Control and Management*, Vol.141, pp.441-449, 2012.
[11] Huan Luo, Yuan Dong, Yiying Ng and Shengyuan Wang, VertexRank: Importance Rank for Software Network Vertices, *Proc. of the IEEE 38th Annual Computer Software and Applications Conference*, Washington, USA, pp.251-260, 2014.
[12] H. Yao, H.J. Hamilton, and C.J. Butz, A Foundational Approach to Mining Itemset Utilities from Databases, *Proc. of SIAM International Conf. on Data Mining*, pp.482-486, 2004.
[13] H. Yao and H.J. Hamilton, Mining Itemset Utilities from Transaction Databases, *Data & Knowledge Engineering*, vol.59, no.3, pp.603-626, 2006.
[14] R. Agrawal and R. Srikant, Fast Algorithms for Mining Association Rules in Large Databases, *Proc. of the 20th International Conf. on Very Large Data Bases*, pp.487-499, 1994.
[15] Y. Liu, W. Liao, A. Choudhary, A fast high utility itemsets mining algorithm, *Proc. of the 1st International workshop on Utility-based data mining*, pp.90-95, 2005.
[16] A. Erwin, R. P. Gopalan, N. R. Achuthan, Efficient mining of high utility itemsets from large datasets, *Proc. of the 12th Pacific-Asia Conf. on Advances in knowledge discovery and data mining*, pp.554-561, 2008.
[17] M. Liu and J. Qu, Mining High Utility Itemsets without Candidate Generation, *Proc. of the 21st ACM International Conf. on Information and knowledge management*, pp.55-64, 2012.
[18] J. Liu, K. Wang, and B. C. M. Fung, Direct Discovery of High Utility Itemsets without Candidate Generation, *In Proc. of the IEEE International Conf. on Data Mining*, pp.984-989, 2012.
[19] Cheng-Wei Wu, Yu-Feng Lin, Philip S. Yu , Vincent S. Tseng, Mining high utility episodes in complex event sequences, *Proc. of the 19th international Conf. on Knowledge discovery and data mining*, Chicago, USA, pp.536-544, 2013.
[20] L. Page, S. Brin, R. Motwani, and T. Winograd, *The pagerank citation ranking: bringing order to the web*, Technical Report of Stanford Digital Library Technologies Project, 1999.