Array 3-4 (2019) 100008

Contents lists available at ScienceDirect

Array

journal homepage: www.elsevier.com/journals/array/2590-0056/open-access-journal

A single fault localization technique based on failed test input

Abubakar Zakari [a,*], Sai Peck Lee [b], Ibrahim Abaker Targio Hashem [c,**]

[a] Department of Computer Science, Kano University of Science and Technology, Wudil, P.M.B, 3244, Kano, Nigeria
[b] Department of Software Engineering, Faculty of Computer Science and Information Technology, University of Malaya, 50603, Kuala Lumpur, Malaysia
[c] School of Computing and IT, Taylor's University, Subang Jaya, Selangor, 47500, Malaysia

ARTICLE INFO

Keywords:
Complex network
Fault localization
Program debugging
Program spectra
Software testing

ABSTRACT

Testing and debugging are very important tasks in software development. Fault localization is a very critical activity in the debugging process and also is one of the most difficult and time-consuming activities. The demand for effective fault localization techniques that can aid developers to the location of faults is high. In this paper, a fault localization technique based on complex network theory named FLCN-S is proposed to improve localization effectiveness on single-fault subject programs. The proposed technique diagnoses and ranks faulty program statements based on their behavioral anomalies and distance between statements in failed tests execution by utilizing two network centrality measures (degree centrality and closeness centrality). The proposed technique is evaluated on a well-known standard benchmark (Siemens test suite) and four Unix real-life utility subject programs (*gzip, sed, flex, and grep*). Overall, the results show that FLCN-S is significantly more effective in locating faults in comparison with other techniques. Furthermore, we observed that both degree and closeness centrality play a vital role in the identification of faults.

1. Introduction

Despite much advancement in software development and testing in recent years, even software of the highest quality may still contain faults, and cause the software to fail [1]. On locating software faults, program debugging generally consumes a momentous amount of resources in a software project. Essentially, a three-step process which consists of first detecting program failure, secondly, determining the exact fault location and the nature of the fault that caused the failure, and lastly, fixing the identified fault itself [2]. Hence, while the entire debugging process is known to be extremely difficult, the second process (fault localization) is broadly reported to be the most tedious, time-consuming, and costly activity among all [3–5]. Even in an educational environment, where students were tasked to find the location of faults in a faulty program code, the report shows that identifying faults location in a program code is an extremely challenging task [6].

In the past decades, several fault localization techniques were proposed with the aim of improving the fault localization process such as spectrum-based techniques [7–9], statistical-based techniques [10,11], model-based techniques [12], and machine learning-based techniques [13,14]. Among these techniques, spectrum-based fault localization technique (SFL) has gained more popularity, thanks to its simplicity and minimal computational overhead. SFL requires little or no knowledge of software program semantics in order to localize faults. Generally, SFL uses test coverage data and their corresponding test results (passed/-failed) from dynamic execution to build a program spectrum. The program spectrum is used to define variables highlighting the distinction between tests execution results (passed/failed). Lastly, similarity coefficients based on SFL techniques are used to compute program statements suspiciousness (the likelihood of statements to be faulty). The statements are then ranked based on their suspicious scores in descending order for fault localization. Based on this insight, various similarity coefficients were proposed to calculate the suspicious score of program statements, such as Ochiai [15], Jaccard [16], DStar [9] etc.

This work builds on previous work [17] where we have proposed a new fault localization technique based on complex network theory (FLCN). Statements behavioral anomalies and the distance between program statements in both passed and failed tests execution are the two variables that the technique takes into account. Degree centrality and closeness centrality were adopted for fault diagnosis and a ranking formula was also proposed to aid in identifying fault location. The technique locates faulty statements irrespective of whether the statements were

* Corresponding author.
** Corresponding author.
E-mail addresses: abubakar.zakari@yahoo.com (A. Zakari), abubakar.zakari@yahoo.com (I.A.T. Hashem).

https://doi.org/10.1016/j.array.2019.100008
Received 6 February 2019; Received in revised form 30 August 2019; Accepted 2 October 2019
Available online 5 October 2019

executed by passed or failed test inputs. We observed that, on single-fault subject programs, the effectiveness is not convincing, whereby a developer can locate only 40% of faults by checking 10% of the program code. This is mainly due to the high sensitivity of the technique (FLCN) with program statements executed by passed test inputs. In this paper, we proposed a variant technique coined fault localization based on complex network theory for single-fault program (FLCN-S) to address this issue. The technique utilizes failed tests execution alone for fault localization (Section 3). Because for a program with a single fault, the faulty statement is more likely to be executed by failed test inputs due to less fault-to-failure complexity as shown in previous studies [18–22]. High fault-to-failure complexity occurs mainly when a program has many faults, whereby fault interfere with each other and deters fault localization effectiveness [21,23].

To demonstrate the effectiveness of utilizing the proposed technique, we perform an experiment on a well-known standard benchmark (Siemens test suite) and four Unix real-life utility programs (*gzip, sed, flex, and grep*). We compared the proposed technique with four different fault localization techniques, namely Ochiai, Tarantula, Jaccard, and SNCM. Overall, the results show a significant improvement on the single-fault subjects where 65% of all the faulty versions can be localized by checking less than 10% of the program code on Siemens test suite programs and is largely more effective on Unix real-life utility programs in comparison with other techniques. Furthermore, we observed that both degree centrality and closeness centrality plays an important role in the identification of faulty program statements. The major contributions of this paper are summarized as follows:

- A new fault localization technique named FLCN-S is proposed to improve localization effectiveness on single-fault subject programs.
- The impact of the adopted centrality measures on fault localization effectiveness was investigated.
- The effectiveness of the proposed technique is evaluated on a standard benchmark (Siemens test suite), and four Unix real-life utility subject programs. The experimental results highlight that FLCN-S is significantly more effective in localizing faults than the compared techniques.

The rest of the paper is organized as follows. Section 2 highlights some existing related work. Section 3 presents the proposed fault localization technique. Section 4 describes the experimental setup, results, and discussion. Lastly, the study is concluded in Section 5.

2. Related work

In the last two decades, many automated fault localization techniques were proposed to localize software faults effectively. In this section, we summarize some of these techniques that are mainly applied on a single-fault context.

Automated fault localization techniques help facilitate localization process. This facilitation helps in the increase in software quality and reduction in software delivery time [24]. SFL techniques exploits program dynamic coverage information of passed and failed test inputs to identify program locations that are more prone to errors [19]. The ratio of passed and failure of test inputs on a given program statement determines how suspicious the statement is. In case if there are more failed test inputs that execute the statement, the statement will have a high suspicious score, and less suspicious score if more passed test inputs executed it. Program statements will then be ranked in descending order of their suspicious score. According to the literature, SFL is one of the most utilized and effective fault localization techniques [24]. In the early days, researchers used failed tests execution alone in locating faults for SFL [25–27]. However, this practice was later shown to be ineffective in locating program faults, particularly on manual fault localization techniques [28]. Studies that use both passed and failed test cases have shown to achieve better results [15,18,29,30]. Renieres and Reiss proposed a technique named nearest neighbor, which produces a suspiciousness report of program statements by measuring the distance between a failed test input and a passed test input which is more similar to the failed one [31]. In SFL, the similarity between the test result vector and the coverage matrix of each statement is measured. Hence, this similarity is quantified and measured by a similarity coefficient-based fault localization technique. Various similarity coefficients and techniques were proposed such as Tarantula [18], Ochiai [15], Crosstab [29], Jaccard [15], and Zoltar-S [32,33]. These similarity coefficients/techniques have consistently shown to be useful in locating distinct sorts of software faults. Jones and Harrold proposed a fault localization technique called Tarantula [18]. Tarantula utilizes coverage information of tests execution (passed/failed) to localize faults. Executable statements are rank based on their suspiciousness score value. Hence, a developer will examine the program code based on their suspiciousness score value in descending order to identify faulty statements. Tarantula has shown to be one of the most effective software fault localization techniques. However, other coefficients have been found in recent years that surpass Tarantula in terms of effectiveness in locating program faults [29,34]. For example, the Ochiai similarity coefficient-based fault localization technique is regarded to be more effective than Tarantula [15]. Ochiai coefficient, S_S, is calculated as depicted in Equation (1).

$$S_s = \frac{Ncf}{\sqrt{(Ncf + Nnf) \times (Ncf + Ncs)}} \tag{1}$$

Ncf represents the total number of failed test inputs that cover a statement, and *Nnf* represents the total number of failed test inputs that do not cover a statement, while *Ncs* represents the total number of passed test inputs that cover a statement. Notations that are widely used in suspiciousness calculation are highlighted in Table 1. Naish et al. proposed two SFL techniques, which are O and O^P [35]. The former was built for programs with a single fault, while the latter is for programs containing multiple faults. The result of the study showed that O and O^P are more effective in localizing faults than Tarantula. Shu et al. proposed a fault localization method based on statement frequency to improve the diagnosis accuracy of SFL techniques [36]. The statement frequency information of each statement in the software program is used to localize faults. The study showed that the proposed approach outperforms Tarantula in terms of stability and effectiveness, respectively. An improved form of Kulczynski coefficient metric coined DStar was proposed in Ref. [9]. The greater the DStar value, the more efficient the technique is in locating program faults. DStar has proven to be very effective and surpasses various fault localization techniques in terms of effectiveness in locating faults. However, researchers in Refs. [19,37] conducted a comparison study of SFL techniques. They concluded that SFL techniques' performance varies based on the debugging scenarios they were applied to, whereby a coefficient can be more effective in a given scenario and less effective in other scenarios [38]. Therefore, there is no coefficient that can outperform all others under every scenario. Table 2 highlights some of the existing similarity coefficient-based fault localization techniques.

De Souza et al. proposed a technique to contextualize code inspection

Table 1
Some of the notations commonly utilized in suspiciousness calculation.

Notation	Description
N	The total number of test inputs
N_f	The total number of failed test inputs
N*s*	The total number of passed test inputs
N*cf*	The total number of failed test inputs that cover a statement
N*cs*	The total number of passed test inputs that cover a statement
Nc	The total number of test inputs that cover a statement
N*uf*	The total number of failed test inputs that cannot cover a statement
N*nf*	The total number of failed test inputs that do not cover a statement
N*us*	The total number of passed test inputs that cannot cover a statement
N*u*	The total number of test inputs that cannot cover a statement

Table 2
Similarity coefficient metrics.

S/N	Coefficient	Formula	S/N	Coefficient	Formula
1	Pearson	$\frac{n \times ((Ncf \times Nus) - (Ncs \times Nuf))2}{Nc \times Nu \times Ns \times Nf}$	9	Hamming	$Ncf + Nus$
2	Ochiai 1	$\frac{Ncf}{\sqrt{(Ncf + Nnf) \times (Ncf + Ncs)}}$	10	Euclid	$\sqrt{Ncf + Nus}$
3	Goodman	$\frac{2 \ Ncf - Nuf - Ncs}{2 \ Ncf + Nuf + Ncs}$	11	Dice	$\frac{2\text{Ncf}}{Ncf + Nuf + Ncs}$
4	Naish 1	$\begin{cases} -1, & ifNuf > 0 \\ Nus, & Otherwise \end{cases}$	12	Anderberg	$\frac{\text{Ncf}}{Ncf + 2(Nuf + Ncs)}$
5	Naish 2	$Ncf - \frac{Ncs}{Ncs + Nus + 1}$	13	Sorensen-Dice	$\frac{2\text{Ncf}}{2Ncf + Nuf + Ncs}$
6	Kulczynski	$\frac{Ncf}{Nuf + Ncs}$	14	Braun-Banquet	$\frac{\text{Ncf}}{\max(Ncf + Nus, +Ncf + Nuf}$
7	Jaccard	$\frac{Ncf}{Ncf + Nuf + Ncs}$	15	Zoltar	$\frac{Ncf}{Ncf + Nuf + Ncs + \frac{10000 \times Nuf \times Ncs}{Ncf}}$
8	Ample	$\frac{Ncf}{Ncf + Nuf} - \frac{Ncs}{Ncs + Nus}$	16	Goodman	$\frac{2 \ Ncf - Nuf - Ncs}{2 \ Ncf + Nuf + Ncs}$

to offer direction during fault localization and increase localization effectiveness of SFL techniques. This technique will help in localizing fault in the first generated suspiciousness list [39]. The result shows that the technique is useful in guiding developers to fault locations and improves localization effectiveness. Another study by Kim et al. proposed a technique to enhance the performance of existing SFL techniques [40]. The technique extracts variables that are suspicious and utilize the variables to generate a suspicious ranked list. The result shows that their proposed technique outperforms existing similarity coefficient-based techniques. Landsberg et al. improve the effectiveness of SFL technique by introducing a new method that generates a viable and efficient test suite for effective fault localization [41]. In another study by Ref. [42], a fault localization technique named FDDI was proposed. The technique (FDDI) chooses the most suspicious function and applies invariant detection tools to the function distinctly. Hence, for the variables which are not in a set of passed/failed test inputs specified by using these tools, FDDI will select those variables for further fault examination.

There are also works on statistical-based fault localization and machine learning-based techniques in locating program faults. In a previous study [43], an algorithm based on statistical-based debugging technique coined SOBER was proposed to rank and isolate suspicious program predicates on faulty programs. The technique categorizes the effects of distinct faults and finds predicates that are related to individual faults. The predicates clarify the circumstances and regularities of fault manifestations and make it simpler to manage and prioritize debugging effort. In addition, Wong et al. introduced a crosstab-based method for effective fault localization [29]. The technique calculates the suspiciousness score value of program statement to know its likelihood of containing faults. Hence, two columns of variables (covered/not covered) of a program statement are generated with two rows indicating tests results (passed/failed). The suspiciousness score value of each program statement is then calculated based on the degree of relationship between statements coverage and their execution results. Furthermore, Liblit et al. proposed a statistical debugging technique to separate faults in a software program with instrumented predicates [44]. Predicates are ranked based on their suspicious score. Predicates with high suspiciousness score are checked first, and if a fault is found and fixed, the faulty data is then removed. The process is then repeated to find the remaining faults until all predicates are examined.

Moreover, a statistical debugging approach was proposed to examine the behavior of continuously linked predicates in a given program execution [45]. For each test input, the approach constructs a weighted execution graph with predicates as vertices against change between sequential predicates as edges. Hence, for each edge in the graph, a suspicious score is computed to identify its fault relevant likelihood. Additionally, a novel probabilistic model for fault localization based on an important sampling of program statements was proposed [10]. By utilizing probability updates and sampling, the approach can help identify statements that have a high likelihood of being faulty. The approach was found to be more sensitive to failed test inputs than passed test inputs. In addition, Wong et al. proposed two machine learning-based techniques for fault localization, fault localization based on BP (back--propagation) neural network [34] and fault localization based on RBF (radial basis function) neural network [14] to localize faults effectively. The result shows that these techniques are effective in locating program faults. However, these techniques have problems of paralysis and local minima. In another study by Zheng and Wang, a fault localization based on Deep Neural Network (DNN) was proposed to tackle the problems of paralysis and local minima [13]. DNN was found to be very effective in comparison to other machine learning-based techniques.

Zhu et al. proposed a fault localization technique named SNCM [46]. The technique uses two centrality measures, namely degree centrality and structural hole, to measure statements correlation with failure. Recently, a fault localization technique based on complex network theory (FLCN) has been proposed to localize faults effectively [17]. Behavioral abnormalities of program statements and distance between them in both passed and failed tests execution are the two variables that the technique takes into account. Degree centrality and closeness centrality were adopted for fault diagnosis and a new ranking formula was also proposed. Most recent studies have shifted to fault localization on multiple faults [4,17,33,47–50] introducing various approaches and methods to localize faults efficiently.

3. Proposed technique

Before we start explaining our proposed technique, we need to revisit our initial benchmark technique (FLCN) and its observable limitation on single-fault context. We further need to highlight our current propositions on how to deal with the aforementioned limitation.

To improve the effectiveness of the former fault localization technique (FLCN) in the single-fault context, we proposed a technique named Fault Localization based on Complex Network theory for Single-Fault programs (FLCN-S), which is a variant of our initial technique in Ref. [17] coined FLCN. FLCN localizes faults with the utilization of both passed and failed test inputs. However, from our initial results, we observed that by utilizing both test inputs (passed/failed) in a single-fault context, the technique (FLCN) effectiveness reduces significantly. At best, FLCN aids developers in locating 40% of the faults by checking less than 10% of the program code in single-fault subjects. This is mainly due to the high sensitivity of FLCN with program statements executed by passed test inputs. Hence, the effectiveness is not convincing in comparison with similarity coefficient-based techniques such as Ochiai [19], Jaccard [15],

or Zoltar-S [33]. Hence, to address this issue, we proposed the use of failed test inputs alone in the fault localization process in modeling the program complex network. This is because for a program with a single fault, the faulty statement is more likely to be executed by failed test inputs due to less fault-to-failure complexity as shown in previous studies [18–22]. High fault-to-failure complexity occurs mainly when a program has many faults, whereby fault interfere with each other and deters fault localization effectiveness [21,23]. In this paper, we proposed a new variant technique called FLCN-S for single fault localization. This technique adopts the same centrality measures and suspicious score formula as utilized in our previous study [17]. The next section presents the proposed fault localization technique.

3.1. Single-fault localization based on complex network theory (FLCN-S)

Suppose we have a faulty program P with m executable statements and exactly a single fault. Suppose P is executed by t-test inputs, whereby some tests have passed while others have failed. The execution data is illustrated as presented in Fig. 1. The figure illustrates a coverage matrix and a test result vector. For the matrix and test result vector with the entry of (i, j), the value of the matrix is 1 if the test input i executes statement j, and the value is 0 otherwise. Moreover, an entry i in the test result vector is 1 if the result of the test input i is failed, and 0 if the result is passed. Hence, each row of the coverage matrix highlights the statements that are executed by available test inputs, and each column shows the coverage vector of the corresponding program statement.

The proposed technique (FLCN-S) uses t amount of test inputs that failed in a program test run as an input to construct the complex network N. The complex network N will be generated following the process detailed in Section 3.1.1. Two centrality measures will be used for fault diagnosis which are degree centrality and closeness centrality.

One of the most commonly utilized centrality measures in complex network research domain is degree centrality. Degree centrality can effectively be used to statistically quantify node significance in a given modeled network. Node degree is the aggregate measure of the edges a given node has in a network. Hence, in this paper context, for statement m_i in an undirected/unweighted complex network, if there is a connection from statement m_i to statement m_j, then each of the statements will have a single edge. Hence, if statement m_i has an edge from another statement m_f that is executed by a distinct test input, then, statement m_i will have two edges since it is connected to two executable statements (m_j and m_f) executed by separate test inputs. The study in Ref. [51] concludes that a node with a higher degree centrality is more likely to be stronger connected in a given network, therefore, the node will be more likely to be the cause of failure or related to failure. This centrality measure (degree centrality) is aimed to identify behavioral anomalies in statements executions and identify the most central statements in the program network N.

Therefore, to calculate the degree centrality of a statement in N, Equation (2) will be computed where m_i is the focal statement, m_j represents any other neighbor statement, n represents the total number of statements in N, and a is the adjacency matrix.

$$Dc(m_i) = \sum_{j=1}^{n} a_{(m_i,\ m_j)} \tag{2}$$

Overall, the adjacency metric will be used to compute the statements' connections, whereby if there is a connection between two statements (m_i, m_j), a is indicated as 1 and 0 otherwise as shown in Equation (3).

$$a_{(m_i,\ m_j)} = \begin{cases} 1, & connection, \\ 0, & Otherwise \end{cases} \tag{3}$$

Next, closeness centrality measures the inverse of the average shortest path between a statement and all other statements in N, which infers that all paths should lead to a statement [52]. This centrality measure tries to measure how long it will take to spread information, diseases, or failures from the node of interest to all other nodes sequentially. For fault that propagates through multiple program statements or faulty statements that are close to each other in the program, this centrality measure will aid in identifying them. Therefore, statements that are closer to the statements with abnormal behaviors (relatively high degree centrality) in N will be identified because the higher the closeness centrality value of a statement, the closer it is to all other statements. To compute the closeness centrality of statement m_i to all other statements in the network N, Equation (4) will be utilized.

$$Cc(m_i) = \frac{n-1}{\sum_{j\neq i}^{n} d(m_i, m_j)} \tag{4}$$

where d ($m_i m_j$) is the shortest path distance between statement m_i and m_j, and n is the total number of statements in N. Furthermore, to calculate the suspiciousness S of statement m_i in N, the technique (FLCN-S) computes the suspicious value of m_i using the degree centrality value Dc of m_i and it's closeness centrality Cc value in N. For a given program statement m_i, the difference between the two values will be computed using Equation (5). Computing the difference of these values will give a developer quantifiable value of how suspicious a statement is.

$$S(m_i) = Dc_i - Cc_i \tag{5}$$

For all statements in N where m_i can be represented as $m_i = \{m_1, m_2, m_3, \ldots, m_n\}$, the suspicious score value will be assigned to each statement in N. The program statements will be given in descending order of their suspicious score values. Henceforth, a developer will check the program statements with the highest suspicious score values until the first faulty program statement is found.

3.1.1. Constructing a complex network using failed test inputs

To model the complex network N, the single fault program in Table 3 is used for illustration. The program has twelve statements, six test inputs ($t = 6$), and exactly one fault in statement m_5. Therefore, if a statement

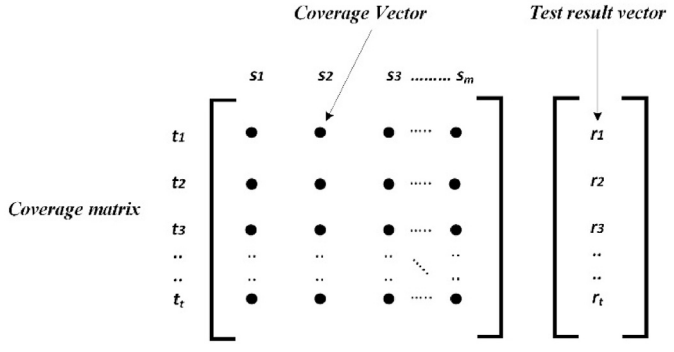

Fig. 1. Coverage data and execution result.

Table 3
A single fault program with tests execution.

m_i	mid ( ) { input x, y, z, m;	t_1	t_2	t_3	t_4	t_5	t_6
m_1	if (y < z)	1	1	1	1	1	1
m_2	if (x < y)	1	1	1	1	1	1
m_3	m = y;	1	1	0	0	1	1
m_4	else if (x < z)	0	1	0	0	0	0
m_5	m = z;//fault 1 m = x	1	0	0	0	1	1
m_6	else	1	0	0	0	0	1
m_7	if (x > z)	0	0	1	1	0	0
m_8	m = x;	1	0	1	1	0	1
m_9	else if (x > y)	0	0	1	0	0	0
m_{10}	m = y	0	0	0	1	0	0
m_{11}	print ("middle number is:", m);	0	0	0	0	0	0
m_{12}	}	0	0	0	1	1	0
Result	**Pass/Fail Status**	0	0	0	0	1	1

execution is labeled as 1, it indicates that the statement is executed by the test input in that test run, and 0 otherwise. For the test result of each test input, 0 means the test input has passed while 1 means the test input has failed. Out of the six available test inputs, two of which are failed test inputs $t_f = (t_5, t_6)$ while four are passed test inputs $t_p = (t_1, t_2, t_3, t_4)$. In order to model the complex network, failed test inputs $t_f = (t_5, t_6)$ is to be utilized. For the first failed test input t_5, there is an edge from m_1 to m_2, m_2 to m_3, m_3 to m_5, and m_5 to m_{12}. For the second failed test t_6, there is an extra edge from m_5 to m_6 and m_6 to m_8, respectively.

Therefore, all the failed test inputs in corresponding to the statements will be modeled as *N*. As a result, a single network *N* will be generated to capture the entire tests execution behavior with statements represented as nodes and execution between statements as edges. Cytoscape software platform (http://www.cytoscape.org/) is used for network construction and generation. We model the network as an undirected/unweighted complex network. The next section highlights the general framework of the proposed fault localization technique.

3.1.2. General framework

This section presents the detail steps of the proposed fault localization technique (FLCN-S) in locating a single fault.

- **Step 1:** In this step, the faulty program P will be executed by all the available test inputs in *T* and the execution data will be collected. The execution profile of statements with respect to each test input will be collected. The set of passed and failed test inputs will be identified.
- **Step 2:** A network *N* will be modeled using the execution profile of failed test inputs as input based on the process detailed in Section 3.1.1.
- **Step 3:** For all program statements *n* modeled as *N*, the *Dc* and *Cc* of each statement will be calculated based on Equation (2) and Equation (4). Furthermore, Equation (5) will be computed to calculate the suspicious score value *S* of each statement in *N*.
- **Step 4:** Based on the suspicious score value of program statements, rank the statements $m_1, m_2, m_3, \ldots, m_n$ based on $S_1, S_2, S_3, \ldots, S_n$ in descending order of their suspicious score values. A developer will be tasked to check the statements one by one from the top until the fault location is identified.

3.2. A running example

Let's use a sample program in Table 3 which takes three integers as input to demonstrate how FLCN-S can be used to locate fault. The program has 12 statements ($n = 12$) with 11 executable statements and a single fault in statements m_5. The faulty statement m_5 is executed by one passed test input $\{t_1\}$ and two failed test inputs $\{t_5$ and $t_6\}$. The proposed technique will rank the faulty statement at the top of the ranking list because it takes into consideration behavioral anomalies and distance between statements in failed tests execution.

At step 2, to model the network *N*, the process detailed in Section 3.1.1 will be used. The network in Fig. 2 is generated using the above-failed tests execution. For the first failed test input t_5, there is an edge from m_1 to m_2, m_2 to m_3, m_3 to m_5, and m_5 to m_{12}. For the second failed test input t_6, there is an extra edge from m_5 to m_6 and m_6 to m_8, respectively. The faulty statement m_5 is executed by all the failed test inputs. Thus, the network as shown in Fig. 2 is generated. Step 1 and step 2 is now completed.

Moving to step 3, the degree centrality *Dc* of a statement m_i and the closeness centrality *Cc* of a statement m_i to all other statements in the network *N* is calculated. Next, in step 4, the suspicious score *S* of the program statements will be calculated according to Equation (5). Rank the statements $m_1, m_2, m_3, \ldots, m_{12}$ based on $S_1, S_2, S_3, \ldots, S_{12}$ in descending order of their suspicious score values. The developer will check the statements one by one from the top until the location of the fault is identified. Lastly, the localization result is shown in Table 4. Looking at the result, the faulty statement m_5 is rank at the top of the ranking list with the highest suspicious score.

4. Experimentation

In this section, the subject programs, data collection process, and evaluation metrics and criteria for the experimental process are highlighted in Section 4.1. The results and discussion are also presented in Section 4.2.

4.1. Experimental setup

4.1.1. Subject programs

For the experiment, we utilized Siemens test suite subject programs [53] and four Unix real-life utility programs (*gzip, sed, flex, and grep*) [54] to evaluate the proposed technique. Generally, various studies have utilized these subject programs for fault localization [9,18,19]. Siemens test suite programs are utilized because the programs contains single fault each, while Unix real-life utility programs contains both real and seeded faults [33]. All of these subject programs are also written in C programming language.

Siemens test suite is composed of seven subject programs, namely *schedule, schedule2, print_tokens, print_tokens2, replace, tot_info,* and *tcas* where each of the subject programs has more than 1000 test inputs. For the Unix real-life utility program, *gzip* program is utilized for file compression and decompression. The program is normally utilized to decrease the size of name files. The input of *gzip* program comprises of 13

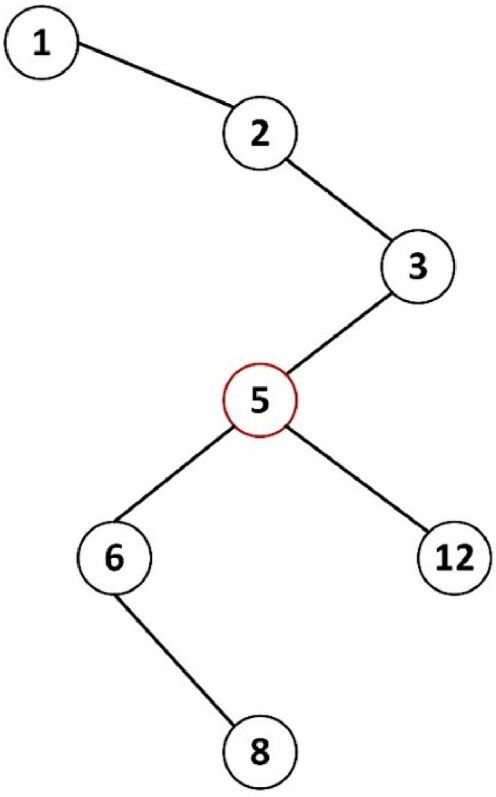

Fig. 2. Network for a single fault program mid ( ).

Table 4
Localization result of mid ( ) with single fault.

m_i	m_1	m_2	m_3	m_5	m_6	m_8	m_{12}
Dc_i	1	2	2	3	2	1	1
Cc_i	0.316	0.428	0.545	0.6	0.461	0.333	0.4
S_i	0.684	1.572	1.455	2.4	1.539	0.667	0.6
Rank	5	2	4	1	3	6	7

options with a list of files as well. The program has 6573 lines of code and 211 test inputs. The *sed* program performs simple changes in an input stream. It is basically utilized to parse textual input and also to apply a specified user changes to the input. The program has 12,062 lines of code and 360 test inputs.

The *flex* program is a lexical analyzer. It reads a file and produces a *C* source file called scanner. The input files contain sets of consistent expression and *C* code, called rules. The program has 13,892 lines of code and 525 test inputs. The *grep* program has two input parameters which are *patterns* and *files*. The program prints lines in each file that contains a match of any of the patterns. The program has 12,653 lines of code and 470 test inputs. Henceforth, the Unix real-life utility programs contain both real and seeded faults. Table 5 gives a brief characteristic of each program: the number of lines of code, the number of test inputs, the number of faulty versions, and brief descriptions of the programs. All the subject programs including the corresponding test inputs were downloaded from the software infrastructure repository (SIR) (http://sir.unl.edu/portal/index.php) [54].

4.1.2. Data collection

We executed each faulty version using all its available test inputs. Test inputs status (passed/failed) are determined by comparing tests execution output for each faulty version to its corresponding original fault-free version. If the output of a faulty version is different from the output of its corresponding fault-free version, the test input will be recorded as a failed test. However, if the output did not differ, the test input will be recorded as a passed test. Moreover, based on the documentation done on Siemens test suite programs and the experimental process in previous works, it is known that Siemens test suite programs contain single fault [42,44,55,56]. Hence, we removed versions whose fault cannot be revealed by any test input because the proposed technique requires failed tests execution for fault localization.

All our programs were executed on a PC with 2.13 GHz Intel Core 2 Duo CPU and an 8 GB physical memory. We use GCC compiler to compile the programs and we use Gcov to obtain the code coverage information for each test execution.

4.1.3. Evaluation metrics and criteria

To measure the effectiveness of a given fault localization technique, suitable metrics are essential for evaluation. In this paper, four metrics are utilized, namely the cumulative number of statements examined, *Exam score*, Wilcoxon signed-rank test, and Top-N.

Table 5
Experimental subject programs.

Program	Number of Faulty versions	Lines of code (LOC)	Number of test inputs	Description
print_tokens	7	565	4130	Lexical analyser
print_tokens2	10	510	4115	Lexical analyser
replace	32	412	2650	Pattern replacement
schedule	9	307	2710	Priority scheduler
schedule2	10	563	5542	Priority scheduler
tcas	41	173	1608	Altitude separation
tot_info	23	406	1052	Information measure
sed	7	12,062	360	Textual manipulator
gzip	5	6573	211	Data compression
flex	22	13,892	525	Lexical analyser
grep	7	12,653	470	Pattern searcher

4.1.3.1. Exam Score.

To access the overall effectiveness of a fault localization technique, a suitable metric must be used for evaluation. The work in Ref. [18] uses a metric named *Score* which is defined as the percentage of code that need not be examined to find a fault. However, in this paper, we will use a metric named *Exam score* which is a variant of the *Score* metric. Although these two metrics provide virtually the same information, *Exam score* is more straightforward and easy to understand. *Exam score* is defined as the percentage of code that needs to be examined until the first statement where the fault resides is reached. Many studies have used *Exam score* to access the effectiveness of a single fault localization technique [15,18,19,46]. The metric is stated in Equation (6).

$$Exam\ \ score = \frac{rank\ \ of\ \ fault}{number\ \ of\ \ executable\ \ statements} \times 100\% \quad (6)$$

Largely, for any given fault localization technique, the technique localization effectiveness can be known with *Exam score*. Therefore, if a given technique A has a lesser *Exam score* than another technique B, then technique A will be considered to be the most effective in comparison to technique B because technique A need to examined less code to locate the faults.

4.1.3.2. Cumulative Number of Statements Examined.

In addition, apart from utilizing *Exam score*, the cumulative (or total) number of statements that need to be examined with respect to subject programs to locate faults is also considered [9]. Therefore, for *n* faulty versions of a given program where X(i) and Y(i) are the number of statements that need to be examined to locate all the faults in the *ith* faulty version by two fault localization techniques X and Y, respectively. Therefore, technique X is more effective than technique Y if technique X requires a developer to examine less amount of statements than technique Y to find all faults in the faulty versions as shown in Equation (7).

$$\sum_{i=1}^{n} X(i) < \sum_{i=1}^{n} Y(i) \quad (7)$$

4.1.3.3. Wilcoxon Signed-Rank Test.

Wilcoxon signed-rank test is an alternative option to other existing hypothesis tests such as z-test and paired student's t-test particularly when a normal distribution of a given population sample cannot be assumed [9,57]. Wilcoxon signed-rank test is also utilized to give a comparison with a solid statistical basis between two or more techniques in terms of effectiveness. Since the aim is to demonstrate that the proposed technique is more effective than the compared techniques, after computing the total number of statements that a developer needs to check on all techniques, an evaluation will be conducted on the one-tailed alternative hypothesis that the other techniques used for cross-comparison require the examination of an equal or greater number of statements than the proposed technique. The null hypothesis is stated as follows:

H_0. The number of statements examined by other techniques $\leq$ the number of statements examined by the proposed technique.

Therefore, if H_0 is rejected, the alternative hypothesis is accepted. The alternative hypothesis implies that the proposed technique will require the examination of fewer statements than the compared techniques which indicates that the proposed technique is more effective.

4.1.3.4. Top-N.

Top-N symbolizes the percentage of faults a fault localization technique ranks for all faulty statements among the Top N (N = 1, 5, 10) positions in the ranked list. Hence, the smaller the value of N in Top-N, the stricter the metric. For instance, Top-5 metric demands that all faults are ranked within the top 5 positions in the ranked list.

4.2. Result and discussion

In this section, the results and discussion are presented. The result of

Table 6
Cumulative number of statements examined to locate faults for each program in Siemens test suite (best case).

	tcas	print_tokens	print_tokens2	schedule	schedule2	replace	tot_info
FLCN-S	195	251	408	311	499	340	200
Ochiai	205	324	423	363	550	370	270
Tarantula	243	391	443	350	555	381	299
Jaccard	259	404	451	388	561	401	304
SNCM	508	490	501	423	603	463	399

Table 7
Cumulative number of statements examined to locate faults for each program in Siemens test suite (worst case).

	tcas	print_tokens	print_tokens2	schedule	schedule2	replace	tot_info
FLCN-S	388	699	600	655	583	455	350
Ochiai	408	712	653	702	627	500	420
Tarantula	483	750	699	701	650	603	550
Jaccard	500	799	708	750	641	513	608
SNCM	812	825	800	801	713	670	800

our comparison with other fault localization techniques is also highlighted and discussed. For all our experiments in this paper, we presumed that the best case effectiveness entails that the faulty statement is identified at the top of the list of statements with the same suspiciousness score values, while for the worst case effectiveness, the faulty statement resides at the bottom of the list with the same suspicious score values. In our evaluation of FLCN-S technique, the result is mostly presented between these two levels of effectiveness for all the evaluation metrics (except for the result presented in Fig. 4). For all our subject programs use in this experiment, each faulty version has exactly one fault.

4.2.1. *Effectiveness of FLCN-S on siemens test suite programs*

Table 6 and Table 7 present the cumulative (total) number of statements examined by FLCN-S and other techniques in both the best and worst cases. For each program, in the best case scenarios, FLCN-S requires the examination of fewer statements than the compared techniques. The same applies to the worst case scenarios. For instance, we observed that for *print_tokens* program, FLCN-S can locate all the faulty versions by examining no more than 251 statements in the best case scenario, and 699 in the worst case scenario, respectively. Furthermore, with respect to the *print_tokens* program, the second best technique is Ochiai, which requires the examination of no more than 324 statements in the best case and 712 statements in the worst case scenarios.

Is worth knowing that these values represent the total number of statements that each technique requires to examine to locate the faults in each subject program. Looking at Tables 6 and 7, we observed that irrespective of which scenario is considered (best case or worst case), FLCN-S is consistently the most effective technique. Another important point worth noting is that, in some exceptional cases, Tarantula (which is the third best technique) is more effective than Ochiai (the second best technique). For example, with respect to *schedule* program, the best case of Tarantula (350) is better than the best case of Ochiai (363), the same applied to their worst cases as well. Nonetheless, the difference between the two techniques (Ochiai and Tarantula) is not that significant where Ochiai is more effective than Tarantula in most cases. Henceforth, is worth re-emphasizing that FLCN-S is the most effective technique with respect to both the best and worst case scenarios.

However, without arriving at any firm conclusion, we present the evaluation of FLCN-S with respect to *Exam score*. The single fault versions of *tcas, print_tokens,* and *print_tokens2* in the best and worst cases are highlighted in Fig. 3. The figure shows the effectiveness of FLCN-S in comparison with four other techniques, namely Ochiai, Tarantula, Jaccard, and SNCM. Therefore, the conclusions drawn with respect to these programs applied to the remaining programs in the Siemens test suite. The *y*-axis indicates the percentage of faults located in all the program faulty versions, while the *x*-axis indicates the effort wasted to locate the corresponding faults.

For instance, based on part (a) and (b) of Fig. 3, we observed that on the *tcas* program, by examining less than 10% of the program code, FLCN-S can locate 85% of the faults in the best case, and 45% in the worst case. Correspondingly, by examining the same amount of code (less than 10%), Ochiai (the second best) can only locate 85% (best case) and 40% (worst case).

In part (c) and (d), the effectiveness score of *print_tokens* is presented. We observed that, by examining less than 10% of the program code, FLCN-S can only locate 65% of the faulty versions in the best case and 35% in the worst case. Ochiai (the second best) can locate 60% of the faults in the best case, and 25% in the worst case. Moreover, the percentage for Tarantula (the third best) is 55% (best case), and 15% (worst case). However, looking at the curves in part (c) and (d), the two techniques (Jaccard and SNCM) are the least effective on *print_tokens* program faulty versions.

Furthermore, in part (e) and (f) of Fig. 3, with respect to *Exam score*, FLCN-S performs relatively better. The curves show that by examining less than 20% of the program code, FLCN-S can locate 55% of the faulty versions in the best case and 35% in the worst case. Ochiai (the second best) can only locate 50% in the best case and 25% in the worst case when examining the same amount of code. For Tarantula (the third best), by examining the same amount of code (less than 10%), is 45% (best case), and 20% (worst case), while for SNCM is 35% (best case) and 15% (worst case).

Fig. 4 gives the overall effectiveness score of FLCN-S and other fault localization techniques on Siemens test suite programs. Based on the result, FLCN-S can locate 65% of the fault in all faulty versions of the Siemens test suite by examining less than 10% of the programs' code. We observed that FLCN-S has yielded a drastic improvement from our benchmark technique (FLCN) [17] with a 25% increase in located faulty versions by checking less than 10% of the programs' code. Furthermore, FLCN-S has also outperformed Zoltar-S approach as concluded by Abreu et al. [33] where the latter can only locate 60% of the faults by checking less than 10% of the programs' code.

Based on the third evaluation metric, Table 8 and Table 9 gives the effectiveness comparisons of FLCN-S with other techniques using the Wilcoxon signed-rank test. The entries in the tables give the confidence of which the alternative hypothesis (which implies that FLCN-S requires the examination of fewer statements than the compared techniques to locate faults) can be accepted. For example, one can say with 97.88% confidence that FLCN-S is more effective than Ochiai on *schedule* program in both best and worst cases. Nevertheless, for *schedule2, replace,* and *tot_info* programs, the confidence to accept the alternative hypothesis is higher

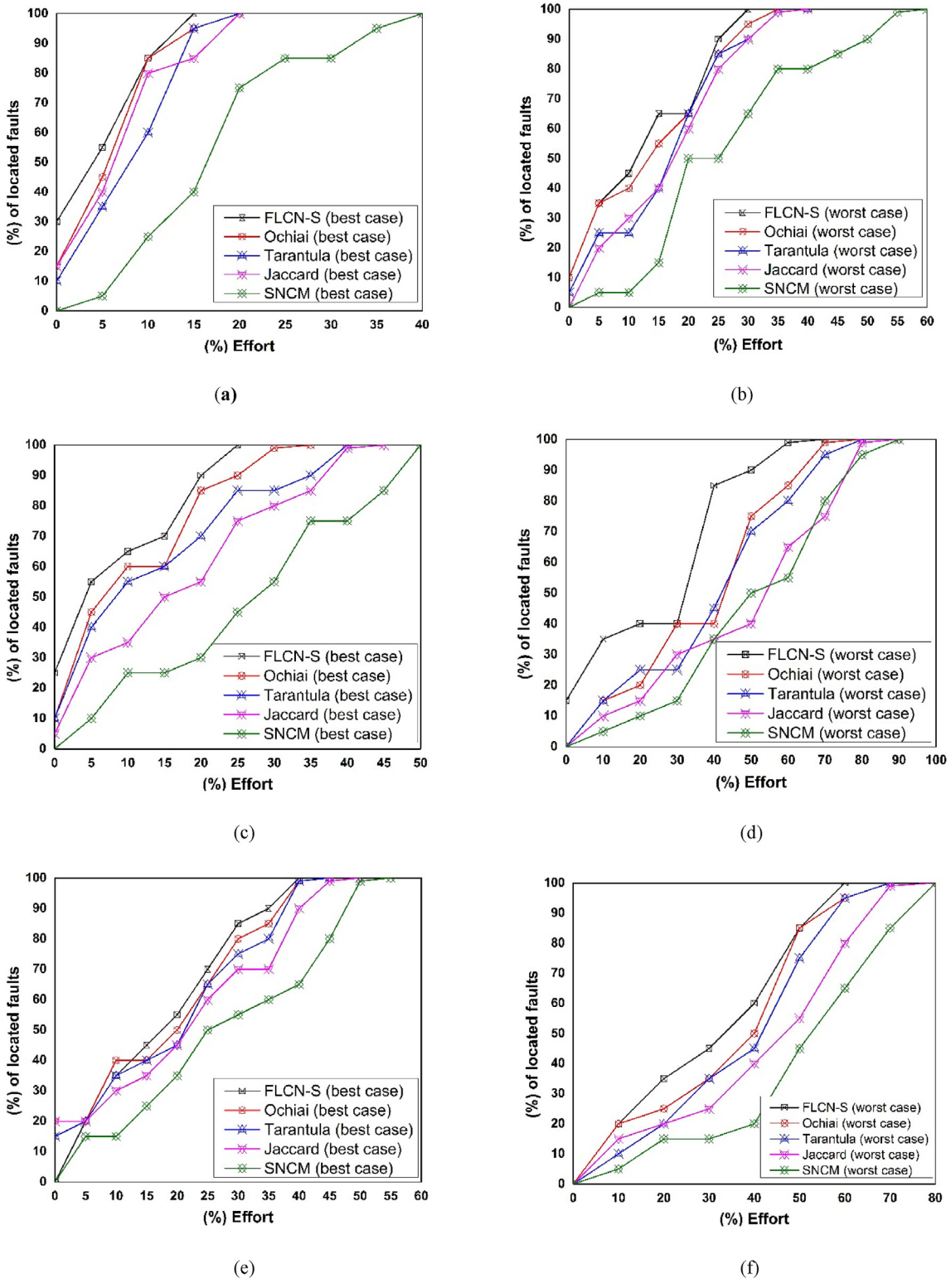

Fig. 3. Exam score-based comparison between FLCN-S and other techniques. (a and b) best case and worst case on tcas, (c and d) best case and worst case on print_tokens, (e and f) best and worst case on print_tokens2.

than 96% in all scenarios (best & worst cases).

Few scenarios have confidence level that is lesser than 95%, for instance, FLCN-S being more effective than Ochiai with 90.00% confidence for the best case of *tcas*, 93.33% confidence being better than Ochiai for the best case of *print_tokens2*, and with 92.31% confidence being better than Ochiai for the worst case of *print_tokens*. In summary, the results from the Wilcoxon signed-rank test clearly shows that FLCN-S is more effective than the compared techniques on Siemens test suite subject programs. The result is also in line with our former conclusion that FLCN-S performs better than the compared techniques in terms of the cumulative number of statements that need to be examined to find all the faults and the *Exam score*.

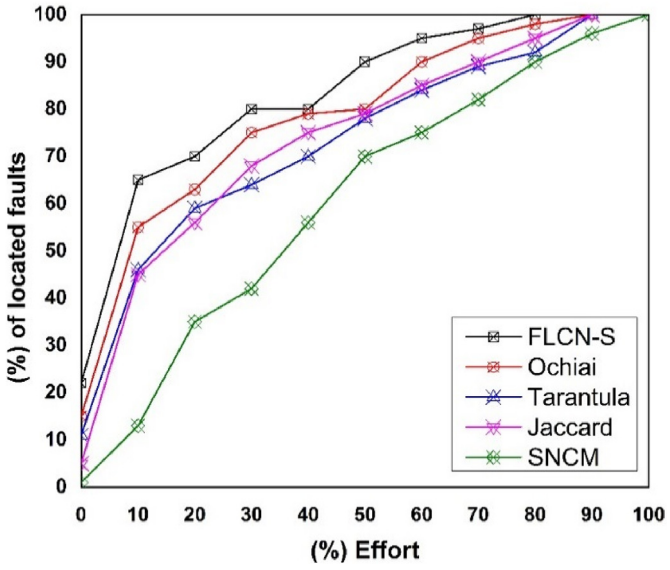

Fig. 4. Overall Effectiveness Comparison on Siemens test suite.

4.2.2. *Effectiveness of FLCN-S on unix real-life utility programs*

Table 10 gives the total number of statements examined by FLCN-S and Ochiai across four programs (*gzip, sed, grep,* and *flex*) to locate all faults in the programs faulty versions. Each faulty version under consideration has exactly one fault for this experiment. From Table 10, we observed that in all scenarios (best and worst cases), FLCN-S is always the most effective in comparison with Ochiai. In some cases, often the worst case of FLCN-S is better than the best case of Ochiai. For instance, with respect to *grep* program, the worst case of FLCN-S (2592) is still better than the best case of Ochiai (3092).

Next, we give the evaluation of FLCN-S with respect to *Exam score*. In Fig. 5, the best and worst case of *gzip* and *sed* programs were presented. However, the conclusions drawn on these two programs are also applicable to the remaining two programs (*flex* and *grep*). The figure shows the effectiveness on FLCN-S in comparison with Ochiai similarity coefficient-based technique. The black curve represents FLCN-S technique and the red curve represents Ochiai similarity coefficient-based technique. Looking at part (a) and (b) of Fig. 5, we find out that on *gzip* program, by examining less than 30% of the program code, FLCN-S can locate 99% of the faulty versions in the best case and 55% in the worst case. In contrast, by examining the same amount of program code, Ochiai can only locate 85% of faults in the best case and 50% in the worst case.

In part (c) and (d), the effectiveness score of *sed* program is presented. The curves show that by examining less than 20% of the program code, FLCN-S can locate 65% of faults in the best case and 40% in the worst case, respectively. Correspondingly, by examining the same amount of program code, Ochiai can locate 55% (best case) and 15% (worst case). The conclusion drawn from Fig. 5 with respect to *Exam score* of both techniques (FLCN-S and Ochiai) is that FLCN-S performs better than Ochiai. This result is consistent with the observations from Table 10 that FLCN-S is the most effective technique. Looking at our third evaluation metric, Table 11 highlights data comparing FLCN-S with Ochiai using Wilcoxon signed-rank test. The table highlights the confidence to which the alternative hypothesis can be accepted. For instance, one can say with 99.88% (best case) and 99.82% (worst case) that FLCN-S is more effective than Ochiai on the *sed* program.

Generally, for *gzip, sed, grep,* and *flex* programs, the confidence to accept the alternative hypothesis is higher than 99%. In total, the results from this test (Wilcoxon signed-rank test) show that FLCN-S is more effective than Ochiai similarity coefficient-based technique which is consistent with our former results using *Exam score* and the cumulative (total) number of statements examined metrics.

Table 12 gives results comparing FLCN-S with Ochiai in terms of Top-N. From the table, the percentage of faults that can be successfully located at Top-1, Top-5, and Top-10 by FLCN-S on *gzip* program is 25.92%, 35.62%, and 51.01%. On the other hand, Ochiai is 19.58%, 29.15%, and 44.27%, respectively. Looking at the table (Table 12), FLCN-S is more effective than Ochiai similarity coefficient-based technique in all scenarios.

4.2.3. *Centrality measures*

In our earlier work, we studied the impact of degree centrality and how statements degree relates to faults. We concluded that statement degree is vital in identifying faulty program statements, especially in a multiple-fault context. In this paper, we observed that on Siemens test suite programs, 23% of all faulty statements have degree centrality of 3 while 77% have a degree centrality of 2. Hence, we found out that in the single-fault context, closeness centrality plays a vital role in ranking and identifying program statements. Moreover, on both Siemens test suite and Unix real-life utility programs, both degree centrality and closeness centrality play a critical and vital role in the identification of faulty program statements.

Table 10
Cumulative number of statements examined by FLCN-S and Ochiai (best & worst cases).

	Best Case		Worst Case	
	FLCN-S Best	Ochiai Best	FLCN-S Worst	Ochiai Worst
gzip	1944	2692	2770	3992
sed	3201	3885	4100	4652
grep	1102	3092	2592	4825
flex	924	1141	1672	1853

Table 8
The confidence with which it can be claimed that FLCN-S is more effective than other techniques on Siemens test suite programs (best case).

	tcas	print_tokens	print_tokens2	schedule	schedule2	replace	tot_info
Ochiai	90.00%	98.64%	93.33%	97.88%	98.04%	96.67%	98.58%
Tarantula	97.92%	99.29%	97.15%	97.44%	98.22%	97.57%	98.99%
Jaccard	98.44%	99.35%	97.68%	98.71%	98.39%	98.37%	99.04%
SNCM	99.69%	99.59%	98.93%	99.11%	99.04%	99.19%	99.50%

Table 9
The confidence with which it can be claimed that FLCN-S is more effective than other techniques on Siemens test suite programs (worst case).

	tcas	print_tokens	print_tokens2	schedule	schedule2	replace	tot_info
Ochiai	95.00%	92.31%	98.08%	97.88%	97.73%	97.78%	98.58%
Tarantula	98.95%	98.04%	98.99%	97.83%	98.51%	99.33%	99.50%
Jaccard	99.11%	99.00%	99.08%	98.95%	98.28%	98.28%	99.62%
SNCM	99.77%	99.21%	99.50%	99.32%	99.24%	99.54%	99.78%

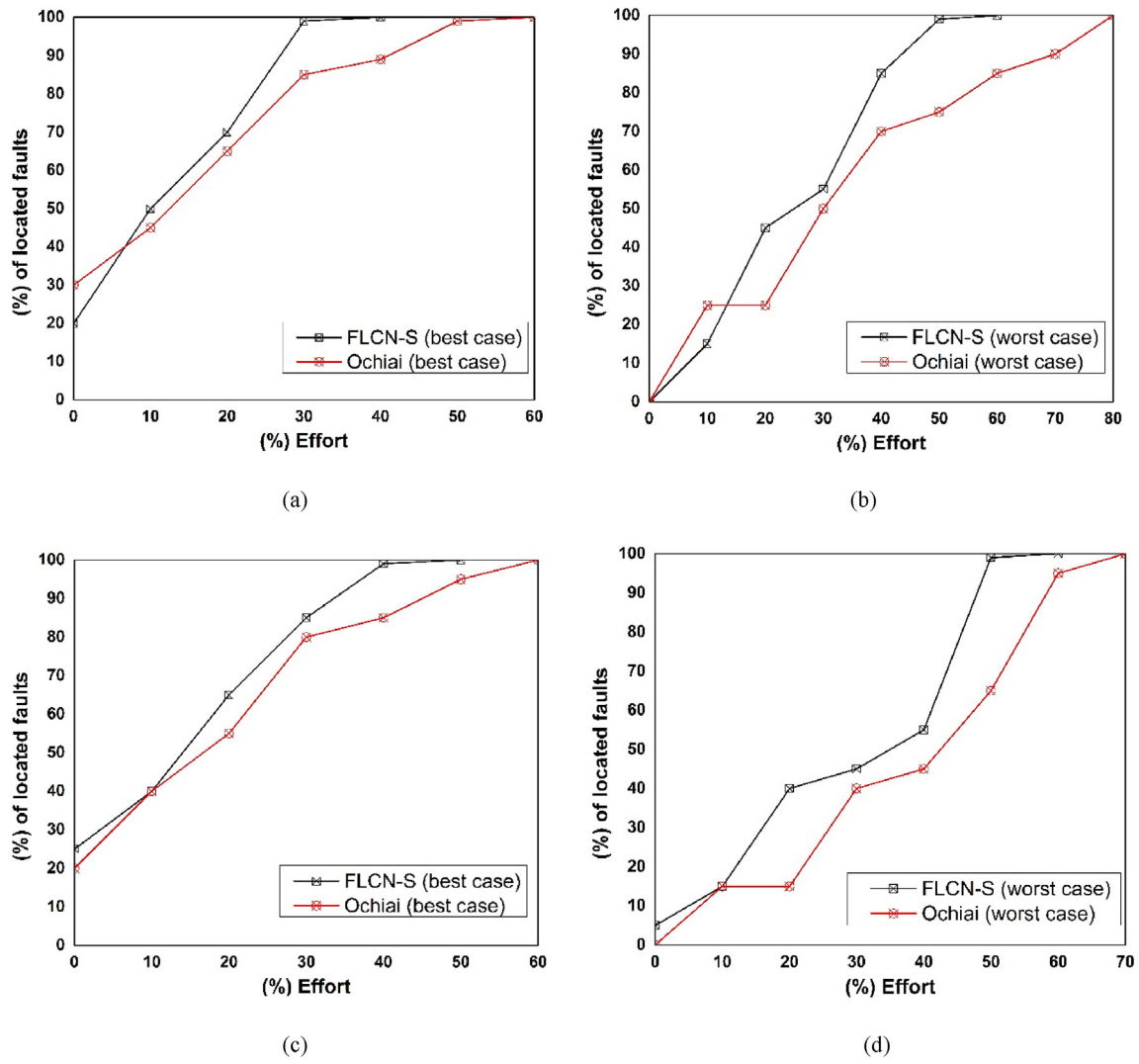

Fig. 5. Exam score-based comparison between FLCN-S and Ochiai similarity coefficient-based technique. (a and b) best case and worst case on gzip, (c and d) best case and worst case on sed.

Table 11
The confidence with which it can be claimed that FLCN-S is more effective than Ochiai (best & worst cases).

	Ochiai Best	Ochiai Worst
gzip	99.87%	99.92%
sed	99.86%	99.82%
grep	99.95%	99.96%
flex	99.54%	99.45%

Table 12
Percentage of faults successfully located at each Top-N metric by FLCN-S and Ochiai.

Programs	Techniques	Top-1	Top-5	Top-10
gzip	FLCN-S	25.92%	35.62%	51.01%
	Ochiai	19.58%	29.15%	44.27%
sed	FLCN-S	34.78%	41.01%	49.00%
	Ochiai	20.09%	32.82%	40.67%
flex	FLCN-S	39.46%	48.01%	58.44%
	Ochiai	20.08%	29.11%	39.12%

In Fig. 6, we observe that most of the faults are located on program statement with degree centrality of 2. However, a significant amount of faults were located on programs with a degree centrality of 3 and 4. Degree centrality is a single factor when localizing faults using our technique with closeness centrality playing a critical role in the fault localization process. From part (h) Fig. 6 (*gzip*), we can see that there are a little number of faults that is located on program statements with degree centrality of 4. Hence, almost 80% of all the faults are located on statements with the degree centrality of 2. This analysis is aimed at confirming the claims of previous studies in various research domains where researchers indicate the important role degree centrality plays in the identification of the most influential and faulty nodes in a network [46,58–61].

4.2.4. *Overall observations*

Generally, the effectiveness of a given technique is not always constant and can change depending on the subject program used. We observed that by utilizing failed test inputs alone, the effectiveness of the proposed fault localization technique (FLCN-S) has increased on single-fault programs. In our initial work, we utilized both test inputs (passed/failed) to localize single faults, the accuracy was not convincing, where we achieved 40% *Exam score* by checking less than 10% of the program faulty versions on Siemens test suite programs. This is mainly

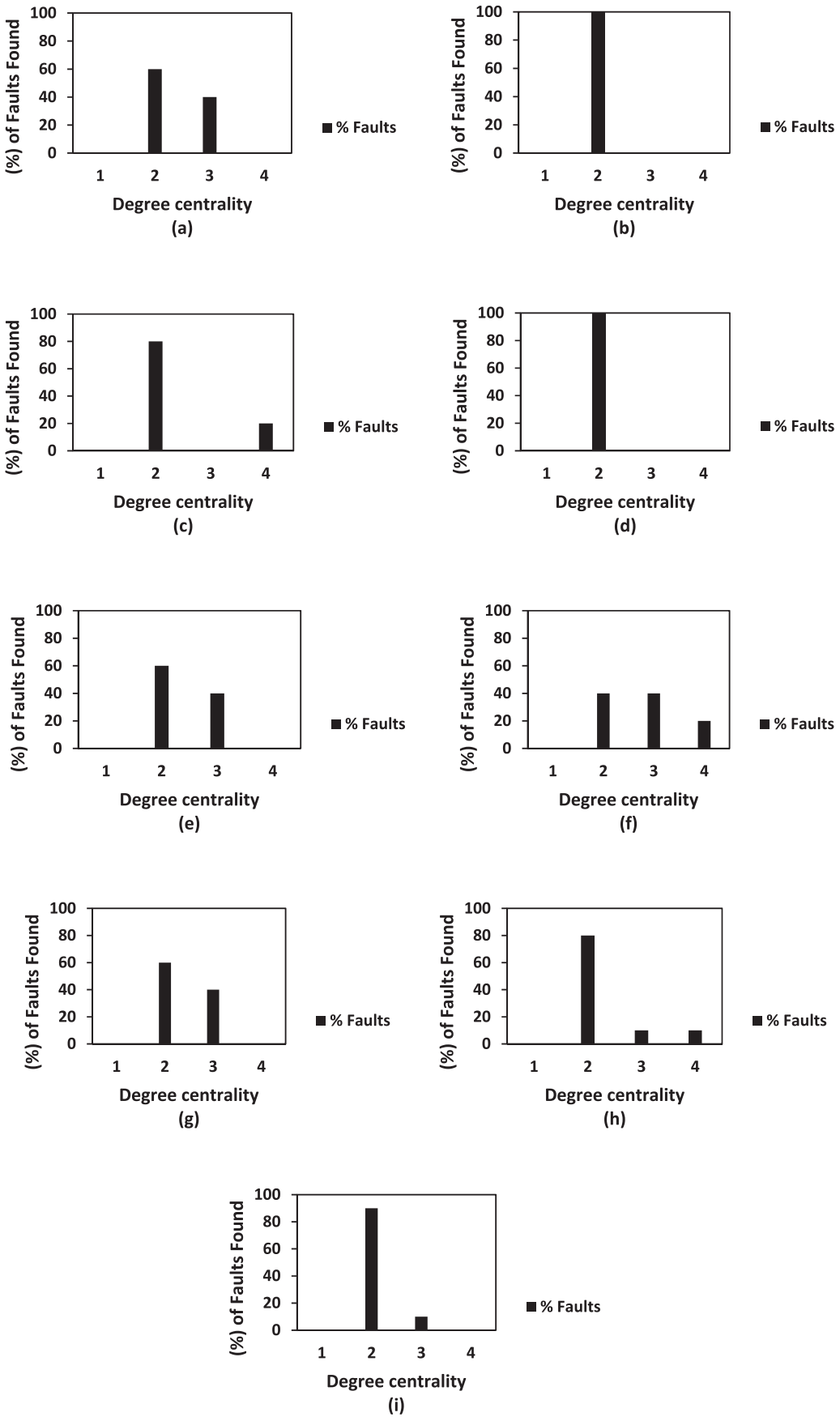

Fig. 6. Degree centrality correlation with failure for Siemens suite programs and two Unix real-life utility programs. (a) schedule, (b) schedule2, (c) print_tokens, (d) print_tokens2, (e) tcas, (f) replace, (g) tot_info, (h) gzip, and (i) sed.

due to the high sensitivity of the technique (FLCN) with program statements executed by passed test inputs. Therefore, we concluded that by utilizing failed test inputs alone, the accuracy of our proposed technique (FLCN-S) increases in the context of single fault due to the minimal fault-to-failure complexity that affects localization on multiple-fault programs. Our technique can effectively localize 65% of all faulty versions on Siemens test suite subjects by checking less than 10% of the program code and is largely more effective on Unix real-life utility program in comparison with other techniques in both best and worst case scenarios. Finally, we also observed that both degree centrality and closeness centrality plays a vital role in the identification of faulty program statements.

5. Conclusion

In this paper, we presented an automated debugging technique, coined FLCN-S to improve localization effectiveness in a single-fault context. The proposed technique is a variant inspired by our previous work [17] which uses both passed and failed tests executions. Our previous technique has proven to be less effective on single-fault subject programs where both test inputs (passed/failed) are taken into account. In contrast, FLCN-S diagnoses and rank program statements based on their behavioral anomalies and distance between statements in failed test inputs.

The proposed technique is evaluated on a well-known standard benchmark (Siemens test suite) and four Unix real-life utility programs (*gzip, sed, flex,* and *grep*). We compared our technique with four fault localization techniques, namely Ochiai, Tarantula, Jaccard, and SNCM. Overall, the results show a significant improvement on the single-fault subjects where 65% of all the faulty versions can be localized by checking less than 10% of the program code on Siemens test suite programs and is largely more effective on Unix real-life utility program in comparison with other techniques in both best and worst case scenarios. Furthermore, we observed that both degree centrality and closeness centrality play a vital role in the identification of faults.

For future work, we will like to explore other centrality measures for fault localization. And we plan to further explore the effectiveness of our technique on larger datasets to further substantiate our claims. Moreover, the proposed fault localization technique (FLCN-S) is considerably more effective in localizing faults in a single-fault context in comparison with our previous work and other techniques compared with.

Declaration of Competing Interest

The authors declare no conflict of interest.

References

[1] Debroy V, Wong WE. Combining mutation and fault localization for automated program debugging. J Syst Softw 2014;90:45–60.
[2] Myers GJ, Sandler C, Badgett T. The art of software testing. John Wiley & Sons; 2011.
[3] Cleve H, Zeller A. Locating causes of program failures. In: Software engineering, 2005. ICSE 2005. Proceedings. 27th international conference on. IEEE; 2005.
[4] Gao R, Wong WE. MSeer-an advanced technique for locating multiple bugs in parallel. IEEE Trans Softw Eng 2017;45(3):301–18.
[5] Zakari A, et al. software fault localization: a systematic mapping study. IET Software; 2018.
[6] Fitzgerald S, et al. Debugging from the student perspective. IEEE Trans Educ 2010; 53(3):390–6.
[7] Zhang M, et al. Boosting spectrum-based fault localization using PageRank. In: Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis. ACM; 2017.
[8] Tang CM, et al. Accuracy graphs of spectrum-based fault localization formulas. IEEE Trans Reliab 2017;66(2):403–24.
[9] Wong WE, et al. The DStar method for effective software fault localization. IEEE Trans Reliab 2014;63(1):290–308.
[10] Namin AS. Statistical fault localization based on importance sampling. In: Machine learning and applications (ICMLA), 2015 IEEE 14th international conference on. IEEE; 2015.
[11] Parsa S, Vahidi-Asl M, Asadi-Aghbolaghi M. Hierarchy-Debug: a scalable statistical technique for fault localization. Softw Qual J 2014;22(3):427–66.
[12] Wotawa F, Nica M, Moraru I. Automated debugging based on a constraint model of the program and a test case. J Log Algebr Program 2012;81(4):390–407.
[13] Zheng W, Hu DS, Wang J. fault localization analysis based on Deep neural network. Math Probl Eng 2016;2016:1820454. 2016.
[14] Wong WE, et al. Effective software fault localization using an RBF neural network. IEEE Trans Reliab 2012;61(1):149–69.
[15] Abreu R, Zoeteweij P. An evaluation of similarity coefficients for software fault localization. In: 2006 12th pacific rim international symposium on dependable computing (PRDC'06). IEEE; 2006.
[16] Chen MY, et al. Pinpoint: problem determination in large, dynamic internet services. In: Dependable systems and networks, 2002. DSN 2002. Proceedings. International conference on. IEEE; 2002.
[17] Zakari A, Lee SP, Chong CY. Simultaneous localization of software faults based on complex network theory. IEEE Access; 2018.
[18] Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. In: 20th IEEE/ACM international conference on automated software engineering, ASE 2005; 2005.
[19] Abreu R, Zoeteweij P, Van Gemund AJ. On the accuracy of spectrum-based fault localization. In: Testing: academic and industrial conference practice and research techniques-MUTATION, 2007. IEEE; 2007. TAICPART-MUTATION 2007.
[20] DiGiuseppe N, Jones JA. Fault density, fault types, and spectra-based fault localization. Empir Softw Eng 2015;20(4):928–67.
[21] Zakari A, Lee SP, Hashem IAT. A community-based fault isolation approach for effective simultaneous localization of faults. IEEE Access 2019;7:50012–30.
[22] Zakari A, Lee SP. Simultaneous isolation of software faults for effective fault localization. In: 2019 IEEE 15th international colloquium on signal processing & its applications (CSPA). IEEE; 2019.
[23] Zakari A, Lee SP. Parallel debugging: an investigative study. J Softw: Evolution and Process 2019:e2178.
[24] Wong WE, et al. A survey on software fault localization. IEEE Trans Softw Eng 2016; 42(8):707–40.
[25] Agrawal H, De Millo RA, Spafford EH. An execution-backtracking approach to debugging. IEEE Software 1991;8(3):21–6.
[26] Korel B. PELAS-program error-locating assistant system. IEEE Trans Softw Eng 1988;14(9):1253–60.
[27] Korel B, Laski J. STAD-A system for testing and debugging: user perspective. In: Software testing, verification, and analysis, 1988., proceedings of the second workshop on. IEEE; 1988.
[28] Agrawal H, et al. Fault localization using execution slices and dataflow tests. In: Software reliability engineering, 1995. Proceedings., sixth international symposium on. IEEE; 1995.
[29] Wong E, et al. A crosstab-based statistical method for effective fault localization. In: Software testing, verification, and validation, 2008 1st international conference on. IEEE; 2008.
[30] Neelofar N, et al. Improving spectral-based fault localization using static analysis. Software: Practice and Experience; 2017.
[31] Renieres M, Reiss SP. Fault localization with nearest neighbor queries. In: Automated software engineering, 2003. Proceedings. 18th IEEE international conference on. IEEE; 2003.
[32] Abreu R, Zoeteweij P, van Gemund AJ. Localizing software faults simultaneously. In: 2009 ninth international conference on quality software. IEEE; 2009.
[33] Abreu R, Zoeteweij P, Van Gemund AJ. Simultaneous debugging of software faults. J Syst Softw 2011;84(4):573–86.
[34] Wong WE, Qi Y. BP neural network-based effective fault localization. Int J Softw Eng Knowl Eng 2009;19(4):573–97.
[35] Naish L, Lee HJ, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Trans Software Eng Methodol 2011;20(3):11.
[36] Shu T, et al. Fault localization based on statement frequency. Inf Sci 2016;360:43–56.
[37] Le T-DB, Thung F, Lo D. Theory and practice, do they match? a case with spectrum-based fault localization. In: Software maintenance (ICSM), 2013 29th IEEE international conference on. IEEE; 2013.
[38] Yoo S, et al. No pot of gold at the end of program spectrum rainbow: greatest risk evaluation formula does not exist. RN 2014;14(14):14.
[39] de Souza HA, et al. Contextualizing spectrum-based fault localization. Inf Softw Technol 2018;94:245–61.
[40] Kim J, Kim J, Lee E. A novel variable-centric fault localization technique. In: Proceedings of the 40th international conference on software engineering: companion proceeedings. ACM; 2018.
[41] Landsberg D, Sun Y, Kroening D. Optimising spectrum based fault localisation for single fault programs using specifications. In: International conference on fundamental approaches to software engineering. Springer; 2018.
[42] Wang X, Liu Y. Fault localization using disparities of dynamic invariants. J Syst Softw 2016;122:144–54.
[43] Liu C, et al. Statistical debugging: a hypothesis testing-based approach. IEEE Trans Softw Eng 2006;32(10):831–48.
[44] Liblit B, et al. Scalable statistical bug isolation. ACM SIGPLAN Not 2005;40(6): 15–26.
[45] You Z, Qin Z, Zheng Z. Statistical fault localization using execution sequence. In: Machine learning and cybernetics (ICMLC), 2012 international conference on. IEEE; 2012.
[46] Zhu L-Z, Yin B-B, Cai K-Y. Software fault localization based on centrality measures. In: Computer software and applications conference workshops (COMPSACW), 2011 IEEE 35th annual. IEEE; 2011.
[47] Xiaobo Y, Bin L, Jianxing L. The failure behaviors of multi-faults programs: an empirical study. In: Software quality, reliability and security companion (QRS-C), 2017 IEEE international conference on. IEEE; 2017.

[48] Sun X, et al. IPSETFUL: an iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra. Front Comput Sci 2016;10(5):812–31.
[49] Liu B, et al. Localizing multiple faults in simulink models. In: Software analysis, evolution, and reengineering (SANER), 2016 IEEE 23rd international conference on. IEEE; 2016.
[50] Naish L, Ramamohanarao K. Multiple bug spectral fault localization using genetic programming. In: Software engineering conference (ASWEC), 2015 24th australasian. IEEE; 2015.
[51] Opsahl T, Panzarasa P. Clustering in weighted networks. Soc Netw 2009;31(2):155–63.
[52] Šubelj L, Bajec M. Software systems through complex networks science: review, analysis and applications. In: Proceedings of the first international workshop on software mining. ACM; 2012.
[53] Hutchins M, et al. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In: Proceedings of the 16th international conference on Software engineering. IEEE Computer Society Press; 1994.
[54] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir Softw Eng 2005;10(4):405–35.
[55] Zhang Z, et al. Fault localization through evaluation sequences. J Syst Softw 2010;83(2):174–87.
[56] Zhang Z, Chan WK, Tse T. Fault localization based only on failed runs. Computer 2012;45(6):64–71.
[57] Ott RL, Longnecker MT. An introduction to statistical methods and data analysis. Nelson Education; 2015.
[58] Girvan M, Newman ME. Community structure in social and biological networks. Proc Natl Acad Sci 2002;99(12):7821–6.
[59] Borgatti SP. Centrality and network flow. Soc Netw 2005;27(1):55–71.
[60] Li D, Han Y, Hu J. Complex network thinking in software engineering. In: Computer science and software engineering, 2008 international conference on. IEEE; 2008.
[61] Cai K-Y, Yin B-B. Software execution processes as an evolving complex network. Inf Sci 2009;179(12):1903–28.