ABSTRACT Effective debugging is necessary for producing high quality and reliable software. Fault localization plays a vital role in the debugging process. However, fault localization is the most tedious and expensive activity in program debugging, as such, effective fault localization techniques that can identify the exact location of faults is eminent. Despite various fault localization techniques proposed, their application in multiple-fault programs is limited. The presence of multiple faults in a program reduces the efficacy of the existing fault localization techniques to locate faults effectively due to fault interference. Moreover, most of these techniques are unable to localize faults simultaneously. This has led researchers to adopt alternative approaches such as one-fault-at-a-time debugging and parallel debugging. In this paper, we propose a novel fault localization technique based on complex network theory (FLCN) to improve localization effectiveness in programs with single-fault and multiple-fault and to aid developers to localize multiple faults simultaneously in a single diagnosis rank list. The proposed technique ranks faulty statements based on their behavioural abnormalities and distance between faulty statements in both passed and failed test executions. Two graph-based centrality measures are adopted for fault diagnosis, namely degree centrality and closeness centrality and a new ranking formula is proposed. FLCN is evaluated across 14 subjects with both single-fault and multiple-fault programs. Our experimental results show that FLCN is more effective at locating faults when compared with existing state-of-the-art fault localization techniques with improvement of fault localization effectiveness in both single-fault and multiple-fault programs.

INDEX TERMS Complex network theory, Fault interference, Program debugging, Program Spectra, Software fault localization.

I. INTRODUCTION

Program debugging has been the forefront activity in reducing software maintenance cost. With the increase in software complexity, faults are virtually inevitable in software programs. Studies have shown that companies spend about 50% to 80% of their budget in software maintenance [1, 2]. This has led to a high demand for automated debugging techniques. Program debugging is composed of three distinct activities, which are failure detection, fault localization, and fault repair. Fault localization is the main focus of our study. The growing complexity of software programs is directly linked to the increase of software faults. Therefore, fault localization techniques play an important role in maintaining software quality and cost reduction. Recently, there has been a transition of fault localization techniques from manual [3, 4] to automated techniques [2, 5, 6], because the traditional manual techniques are not scalable and unable to keep up with the increasing software complexity.

The most tedious and expensive activity in program debugging is fault localization [7], the activity where software developers are required to identify the exact locations of software faults in the source code at different levels of granularity (class, functions, or statements). With the growing need to improve the effectiveness of fault localization, many automated fault localization techniques have been proposed [6, 8-10]. One of the most popular techniques are the spectrum based fault localization techniques (SBFL), thanks to their low computational overhead with little or no knowledge requirement of program semantics to identify the location of
faults. An SBFL uses execution data during program testing (program spectra) and the execution result (passed/failed) for each test case to compute the statements suspiciousness, which is the likelihood of a statement to be faulty or not for each program component. The technique identifies parts of a software program that are mostly correlated with failure, therefore the likelihood of a program statement to be faulty is based on the suspicious score it carries. Program statements are then ranked according to their suspicious score for fault examination. The most broadly used technique is Tarantula [2, 8], which uses a large amount of data collected from program testing to understand the correlation between program statements and failure. Numerous techniques have been proposed such as Ochiai [11], Crosstab [9], Jaccard [11], and Zoltar-M [12, 13] which have shown to be effective in localizing different types of software faults.

However, most of the earlier fault localization techniques are mostly applied to programs with a single fault [14]. In reality, program failure can be caused by more than one independent fault. Therefore, a faulty program may contain multiple simultaneous faults (concurrent active faults). The studies in [15-18] have shown the profound effect of multiple simultaneous faults in a given faulty program. The authors found out that the effectiveness of the existing fault localization techniques decreases when a program contains multiple active faults due to interference between faults. The more faults in a program, the more difficult it is for a developer to localize faults due to a fault to failure complexity. This phenomenon (fault interference) is very crucial to the effectiveness of fault localization techniques and inevitable in the context of multiple faults [15]. Therefore, localizing multiple faults simultaneously using traditional localization techniques [2, 8, 19, 20] is difficult because of their reliance on failed test execution to identify faulty statements. Although, researchers have proposed various techniques and approaches [6, 13, 14, 21-24] to localize multiple faults, however, their experimental results showed that the effectiveness is still lackluster.

In this paper, we propose a novel fault localization technique based on complex network theory (FLCN) to localize faults in both single-fault and multiple-fault programs. Generally, complex network theory is vastly used in various research fields in science such as physics [25], biology [26], social networks [27], and neurology [28]. The theory is used to understand complex patterns, complex behaviors, error or disease propagation, social networks, software systems and much more. In recent years, complex network has been used in software engineering field of research to study object-oriented systems so as to capture its structural and behavioral characteristics [29, 30]. The study in [30] analysed software evolution process using complex network. The authors proposed a model of complex network to integrate and understand software dynamic behavior. Other studies tried to explore the idea of complex network application in software engineering [31]. The study in [20] proposed a fault localization technique using network centrality measures (SNCM). The authors adopted two centrality measures for fault diagnosis which are degree centrality and structural hole to understand program statements that are correlated with failure. However, based on our knowledge, there are limited studies on the application of complex network in fault localization. In Section 2, we elaborate more on the background of complex network theory in general. The main objective of this study is to improve localization effectiveness in programs with single-fault and multiple-fault and to aid developers to localize multiple active faults simultaneously in a single diagnosis rank list.

Our technique ranks faulty statements based on their behavioural abnormalities and distance between faulty statements in both passed and failed test executions. With fault interference in mind, the proposed technique identifies the locations of faulty statements regardless whether the statements are executed by failed or passed test cases. We evaluated FLCN across 4 different sets of programs (Siemens suite, Siemens-M, Gzip, and Sed) amounting to 14 different subject programs in totality. Siemens-M is a multiple-fault version based on the Siemens suite program. We created multiple-fault versions containing 2, 3, 4, and 5 faults across 5 Siemens suite programs to produce the Siemens-M case study (Section 5.1). The experimental results show that the proposed technique performs better than most of the existing fault localization techniques, where it is capable of locating 50% of all faults by checking only less than 20% of all our multiple-fault versions using simultaneous debugging approach. A detailed explanation of the proposed technique is discussed and illustrated in (Section 4).

This paper is structured into different sections. Section 2 gives an overview of the related work. Section 3 presents the motivational example. Section 4 presents the proposed technique. Section 5 contains the experimental setup, result and discussion. Section 6 discusses the threat to validity of the study, while Section 7 contains the conclusion and future work.

II. RELATED WORK

Fault localization is an active area of research for the past few decades. Various techniques have been proposed to localize faults effectively. In this section, we discuss several studies that are highly relevant to our work and related literature on complex network.

A. FAULT LOCALIZATION TECHNIQUES

Tarantula fault localization technique was proposed by Jones and Harrold in [8]. The technique uses coverage information of both passed and failed tests execution for fault localization. It ranks executable program statements based on their suspicious score. In Tarantula, a developer will start examining the program codes based on their suspicious score in descending order to locate program faults. Furthermore, Jones et al [14] proposed an approach named debugging in
parallel to assist in debugging programs with multiple faults. They attempted to cluster failed test cases into $K$ number of clusters, and combine each cluster with all passed test cases to form a fault-focused cluster. These fault-focused clusters are then given to selected developers to debug the faults in parallel.

In the work by [6], a modified form of Kulczynski similarity coefficient named DStar was proposed. The higher the DStar value, the more effective the technique in locating faults. The study in [32] proposed a fault localization method based on disparities of dynamic invariants. The method is called FDDI. FDDI selects a highly-suspected function and then apply invariant detection tools to this function separately. Variables that are not in a set of passed/failed test cases indicated by using these tools will be picked by FDDI for further analysis. The study in [20] proposed a fault localization technique based on software network centrality measure (SNCM) and adopts two centrality measures for fault diagnosis which are degree centrality and structural hole to understand program statements that are correlated with failure.

On the other hand, in the work by Liu et al [33], a statistical-based debugging algorithm named SOBER was proposed to rank suspicious predicates in a single run and also isolate faults in programs with multiple faults. SOBER classifies the effects of different faults and identifies predicates related to different faults. Therefore, these predicates will explain the conditions and frequencies of fault occurrences. This makes it easier to prioritize debugging effort. Wong et al. [9] proposed a crosstab based fault localization technique. This technique computes the suspiciousness of each program statement to measure the likelihood of it containing a fault. Moreover, two columns of variables (covered and not covered) of a given statement are created with two rows of variables (passed and failed). The suspiciousness of each statement will be computed based on the degree of association between the coverage of statements and their execution results.

Fault localization based on Simulink model was proposed in [34]. The approach used supervised learning technique named decision tree to cluster failed executions likely to have been caused by a single fault. The approach generates a rank list based on a statistical debugging technique whereby developers will check the rank list to find a fault, fix it, and retest the Simulink model to localize the remaining faults. The study in [35] proposed a fault localization framework that reduces the noise between faults in programs with multiple faults. The framework uses a chain of key basic blocks of a program and a noise reduction method to improve similarity coefficient fault localization techniques.

The study in [15] empirically examined how the quantity of faults affects SBFL technique effectiveness, and also, whether the quantity of faults contributes to the increase in fault interference. They concluded that even in the existence of multiple faults, at least one fault can be found with good effectiveness. In another study [36], a novel approach named the concept lattice of program spectrum for effective multiple fault localization (CLPS-MFL) was proposed. The approach uses formal concept analysis to convert a program spectrum into a concept lattice and uses three strategies to find failure root causes. Abreu et al [37] used a model-based approach to localize multiple-fault candidates, they used De Kleer’s intermittent fault model to explain software component behaviour. Moreover, in another study [12], the authors proposed a logical reasoning multiple-fault localization approach called Zoltar-M which was combined with Bayesian probability theory to rank multiple-fault candidates.

B. COMPLEX NETWORK THEORY

Complex network is capable of simulating complex data behaviour to understand and identify important components [26]. For the past few decades, researchers in diverse fields of studies have given a lot of attention to complex network study due to its robustness and adaptiveness in solving complex problems [25, 28, 38]. Particularly, physicists have shown a lot of interests in complex networks to provide a detailed explanation of various system topologies such as social networks, communication systems, World Wide Web, community structures, epidemic spreading and much more. Complex network has a lot of advantages over many models used in the study of complex data. It has the ability to learn highly complex data and structures of a system. The working principle of complex network is flexible and robust because the complex relationships of system components can be understood at a macro level. Most importantly, complex network has a strong mathematical background. These advantages have led complex network in becoming an important tool for understanding system complexity. It has been successfully applied in the scientific research areas such as neurology [28], biology [26], physics [25], social network [39], and software engineering [40, 41].

The study by [25] conducts a study to understand networks and the complexity of World Wide Web (WWW) from a physicist point of view. The authors try to understand the basic principles of networks structural organization and evolution. The study by [28], evaluate some of the most basic issues in networks in the area of neurobiology from the perspective of nonlinear dynamics. The outcome of the study shows that there are issues about the nonlinear dynamics of systems coupled according to small-world, scale-free or generalized random connective network. Furthermore, in order to capture the structural characteristics of an object-oriented software system, a study in [29] proposed an approach to represent the system using a weighted complex network. Based on the complexities of classes and their dependencies (methods), nodes and edges are modeled. Graph theory metrics were used on the modeled network. Their result shows that their proposed approach can help in identifying software components that violate software design principles.

In the work by [40], theories and methods were proposed to analyse the topological and structural complexity of software
systems. The study in [41] examined software collaboration graphs of several open-source systems, and the findings show that software graphs indicate small-world and scale-free network characteristics identical to those identified in other systems (sociological, biological, neurological etc.).

Complex network or sometimes called graph theory has two basic types of graphs which are, directed graph and undirected graph. In this paper, we model our graph as an undirected graph because we are not interested in knowing the micro-detailed relationship between software components. A graph is a group of nodes connected together via edges that may or may not be weighted. There are two types of node-to-node relationships, which are, symmetric and asymmetric. Nodes relationships are symmetric if the graph is modeled as an undirected graph, while the nodes relationships are asymmetric if the graph is modeled as a directed graph [42]. Basically, a graph provides an abstract representation of the modeled data, for instance, social network, complex data or software system components and their interactions, which is often regarded as a real-world network because it simulates real-world behaviour of a system.

Real-world networks show essential topological structures, patterns, and behaviours of a system. Existing studies have found that real-world networks possess some unique characteristics and behaviour such as the scale-free and small-world properties, which are not found in random networks [41, 43]. A directed graph is more suitable when applied in object-oriented software systems because it can capture the semantic relationships between software components. In this paper, we are not interested in detailed relationships between components as the case in object-oriented systems [40]. We modeled our graph as undirected graph. To build a software-based complex network, a developer needs to have a valid input regarding the structural behaviour of the software such as software components and their relationships. In our case, the input value will be program spectra and the execution results (passed/failed) of a program. The program statements will be modeled as nodes and the execution between them will be modeled as edges. Hence, we use the term program statement and node interchangeably.

In this paper, complex network is used for fault localization for the following reasons. Firstly, complex network theory is proven to be largely applicable theory and it has been effectively used to solve several problems in research areas such as physics [25], biology [26], social network [27], and software engineering [29]. Secondly, complex network has the ability to help researchers understand complex systems. For instance, it can be used to identify important nodes and their correlation with faults in software programs. Lastly, in the context of fault localization, complex network theory has the ability to aid in the identification and localization of faulty program statements and statements that are related to the failure.

III. MOTIVATIONAL EXAMPLE

In this section, a program with two bugs is illustrated as an example to show the effect of multiple faults on the effectiveness of SBFL techniques. The example shown in Figure 1 has two faults “bug 1” and “bug 2” and 4 test cases \{t_1, t_2, t_3, t_4\} in which t_1 and t_2 are passed tests while t_3 and t_4 are failed tests. If a statement execution is labeled as 1, that means the statement is executed by the test case in that test run. If a statement is labeled as 0, that means the statement is not executed by the test case in that test run. For the test result of each test input, 0 means the test case has passed while 1 means the test case has failed.

![Figure 1. Effect of multiple faults on fault localization techniques](image)

<table>
<thead>
<tr>
<th></th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>Susceptibility (Ochiai)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: if (b)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.70% 0.60% 0.60%</td>
</tr>
<tr>
<td>2: bug 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.50% 0.70% 0.70%</td>
</tr>
<tr>
<td>3: } else</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.50% 0.70% 0.70%</td>
</tr>
<tr>
<td>4: bug 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.70% 0.60% 0.60%</td>
</tr>
<tr>
<td>5:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Test Result

| 0  | 0  | 0  | 1  |

**Susceptibility (Ochiai)**

\[
S\text{us}p\text{i}c\text{i}o\text{n} = \frac{N_{cf}}{\sqrt{(N_{cf} + N_{nf}) \times (N_{cf} + N_{cs})}}
\]

\(N_{cf}\) denotes the number of failed test cases that cover a statement, and \(N_{nf}\) denotes the number of failed test cases that do not cover a statement, while \(N_{cs}\) denotes the number of passed test cases that cover a statement. When all test inputs are considered for suspicious score computation as shown in the fifth column, the suspicious score of \(s_2, s_3, s_4\) are the same (0.50%) while the non-faulty statements \(s_1\) and \(s_3\) have a very high suspicious score. As a result, this makes the Ochiai metric ineffective in localizing the two faults “bug 1” and “bug 2” if all tests are considered. Nonetheless, if \(t_3\) is removed from the test suite as shown in the sixth column, a developer can effectively localize “bug 2” which was given a high suspicious score (0.70%). On the other hand, if \(t_4\) is removed from the test suite (seventh column), “bug 1” can be effectively localized.
by the technique because “bug 1” has the highest suspicious score with (0.70%). Therefore, this scenario illustrates that with two faults in a faulty program, utilizing all the available test inputs might reduce the effectiveness of a fault localization algorithm. Hence, the example shows that by isolating test inputs a debugger can target and localize each fault separately. However, isolating faults into many separate groups can increase debugging cost and further increase software time-to-delivery. Fault localization gets even more difficult if more faults are present in a program. This example illustrates in detail the ineffectiveness of SBFL techniques in a program with multiple faults when utilizing all test inputs.

Our contributions in this paper are three-fold. First, we propose a novel fault localization technique based on complex network theory (FLCN) to improve localization effectiveness in programs containing both single and multiple active faults and to localize multiple active faults simultaneously in a single diagnosis rank list. Second, we propose a new metric named incremental developer expense (1DE) for estimating the efforts spent by a developer to localize multiple faults simultaneously in a single diagnosis rank list. Lastly, an empirical evaluation of the proposed technique using both single-fault and multiple-fault programs is conducted. The evaluation results successfully demonstrate the practicality and effectiveness of our proposed technique whereby the proposed technique is capable of locating 50% of all faults by checking only less than 20% of all our multiple-fault versions.

IV. PROPOSED TECHNIQUE

In this paper, we adopt two graph-based centrality measures for fault diagnosis, namely degree centrality and closeness centrality. Degree centrality measures the number of connections a node has to other nodes in a network, and closeness centrality of a node measures how close a node is to other nodes in the network. A new ranking formula is proposed to compute the suspiciousness of all program statements. This is based on the knowledge that a faulty statement might play a distinct role in the network.

**Definition 1.** (Degree centrality). Degree is a commonly used centrality measure in complex network and it can be utilized to statistically measure node importance in a given network [45, 46]. The node degree is measured by the total number of edges the node has. For an executable node $N_i$ in a sparse undirected network. If there is an edge from an executable node $N_i$ to node $N_j$, these means each of the nodes has a single edge. Hence, if node $N_i$ has an edge from another node $N_j$ that is executed by a separate test case, the node $N_i$ will record 2 edges because it is connected to two executable nodes ($N_i$ and $N_j$) executed by different test cases. The study in [47] deduces that a node with a higher degree centrality is likely to be stronger connected in a given network, hence being more likely to be the cause of failure or related to failure. The calculation of a node degree in a modeled program spectra is computed by Equation 1.

$$Dc(i) = \sum_{j=1}^{n} a_{ij} \quad (1)$$

For node $N_i$, the execution between the node and it neighboring executable node $N_j$ will be recorded. Where $j$ represents any other neighbor node of $N_i$, and $n$ represents the total number of executable nodes in the entire program. An adjacency metric is used to compute node to node connections where $a_{ij}$ is the connection between statement $i$ and statement $j$. Therefore, $a_{ij} = 1$ (adjacency metrics) when there is connection between two statements, while $a_{ij} = 0$ otherwise as in Equation (2).

$$a_{ij} = \begin{cases} \text{connection}, & 1 \\ \text{Otherwise}, & 0 \end{cases} \quad (2)$$

However, higher degree centrality does not always point to a faulty node. Our experiments on multiple fault subjects show nodes with much higher degree contains on average 30% to 40% of all the faults. Even though the degree of a node varies from 1 to 6 or more depending on the quality of a test suite, is still not enough. Therefore, to help in locating more faults, we adopt closeness centrality measure.

**Definition 2.** (Closeness centrality). Closeness centrality measures the length of the average shortest path between a node and all other nodes in a network, which means that all paths should lead to a node [43]. For faults that propagate through multiple program statements that are close to each other, this centrality measure will aid in locating them. As shown in Figure 2 and Figure 4, program statements $s_6$ and $s_8$ might have distinct degree centrality, but due to their close proximity, they will have a relatively closed closeness centrality value.
int mid (x, y, z, m) {
    1:     m = z;
    2:     if (y < z) {
            3:         if (x < y) {
                4:             m = y;
            5:         } else if (x < z) {
                6:             m = y;  // fault1
            7:         } else
                8:         if (x > y) {  // fault2
                9:             m = y;
            10:        }
    }
}

Figure 2. Closeness centrality example for program mid() 
To calculate the closeness centrality of a node \( N_i \) to all other nodes in the network, Equation 3 will be computed.

\[
C_c(j) = \frac{n - 1}{\sum_i d(i,j)} \quad (3)
\]

Where \( d(i,j) \) is the distance between node \( N_i \) and node \( N_j \), and \( n \) is the total number of executable statements in the network.

**Definition 3.** (Suspicious score). Our technique computes the suspicious value of each program statement using the degree centrality and closeness centrality value of a statement. For a given program node \( N_i \), the difference in the degree value of \( N_i \) and the closeness value of \( N_i \) will be computed using Equation 4 with respect to the entire program statements \( n \). Therefore, a suspicious score value will be assigned for each node \( n \) in the network. All program nodes will be generated in descending order of their suspicious score. Hence, a developer will start checking the program statements with the highest suspicious score until the faulty program nodes are identified.

**Suspicious score**

\[
\text{Suspicious score} = \sum_i (D_c - C_c) \quad (4)
\]

Equation 4 is used to compute all the statements suspicious score where \( D_c \) denotes the degree centrality of a node, \( C_c \) denotes the closeness centrality of a node, and \( n \) represents the total number of nodes in the network.

### A. COMPLEX NETWORK MODELLING

To model our complex network \( N \), we used the example given in Figure 4. The program has 12 statements \((n=12)\) with 11 executable statements and 3 active faults in \((s_1, s_6, \text{and } s_8)\). Out of the 3 faults, one fault \((\text{i.e. } s_8)\) is only executed by passed test cases \((t_3, t_4)\). Tarantula will rank this faulty statement at the bottom of the rank list because it is not executed by at least one failed test case. However, our proposed technique rank the statement at the top of the ranking list because it takes into consideration behavioural abnormalities and distance between faulty statements in both passed and failed test executions. We modeled our network using Cytoscape software platform (http://www.cytoscape.org/).

<table>
<thead>
<tr>
<th>Test cases</th>
<th>Execution trace</th>
<th>Test result</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>{1-2} {2-3} {3-5} {5-6} {6-12}</td>
<td>Passed</td>
</tr>
<tr>
<td>t2</td>
<td>{3-4}</td>
<td>Passed</td>
</tr>
<tr>
<td>t3</td>
<td>{2-7} {7-8} {8-9}</td>
<td>Passed</td>
</tr>
<tr>
<td>t4</td>
<td>{8-10} {10-12}</td>
<td>Passed</td>
</tr>
<tr>
<td>t5</td>
<td>{5-12}</td>
<td>Failed</td>
</tr>
<tr>
<td>t6</td>
<td></td>
<td>Failed</td>
</tr>
</tbody>
</table>

As shown in Table 1, a single network \( N \) is modeled to capture the entire program executable statements execution behaviour. Execution trace of \{t1, t2, t3, t4, t5, and t6\} is used to model network \( N \) irrespective of their execution results (passed/failed). From \( t_1 \), we know that there is an edge between \( s_1 \) and \( s_2 \), \( s_2 \) and \( s_3 \), \( s_3 \) and \( s_5 \), \( s_5 \) and \( s_6 \), \( s_6 \) and \( s_8 \), respectively. Any redundant connection between statements will not be considered, that is why test case \( t_6 \) is empty because it has the same execution trace as \( t_1 \). And \( t_2 \) shows an extra edge for \( s_3 \) with the connection between \( s_3 \) and \( s_4 \), which applies to the rest of the test cases as well.
B. FAULT LOCALIZATION

For a program $P$ with $n$ executable statements and three faults in $(s_8, s_3, s_6)$ as illustrated in Figure 4, $P$ is executed using $M$ number of test cases provided by all the faulty versions. Out of the $M$ test cases, some have successful results while some have failed results.

Figure 4 shows the effectiveness of our proposed fault localization technique in the program with multiple-fault. In $P$, $s_8$ (which is faulty) is only executed by two passed test cases, using a traditional fault localization technique, it will result in a low suspicious score. In our proposed technique, faulty statements (such as $s_8$) will be ranked by measuring the deviation in the behaviour of the program statement irrespective of the test case that executes it using our defined centrality measures.

Figure 3 shows the process of fault localization based on FLCN technique. We summarize the process as follows:

1) Execute the faulty program ($P$) with available test suite and identify passed and failed test executions.
2) Model the complex network of $P$ using statement execution profile as input data.
3) For all the executable program statements ($n$), compute degree centrality ($D_c$) and closeness centrality ($C_c$) based on Equation (1) and Equation (3).
4) Calculate the suspicious score value for each statement using Equation (4).
5) Rank program statements ($s_1, s_2, ..., s_n$) in descending order and examine the statements one by one from the top until the faults are located.

V. EXPERIMENTATION

The proposed fault localization technique is evaluated in this section, where we describe the experimental setup in Section 5.1. Evaluation metrics used in our study are discussed in Section 5.2. Subsequently, results and discussion of the empirical study are presented in Section 5.3.
A. SUBJECT PROGRAMS

Table 2 shows the details of the programs used in this study. The programs were used in our single-fault experiment, where each of these programs contains only a single fault. The programs were all downloaded from software infrastructure repository (SIR) (http://sir.unl.edu/portal/index.php). These programs have been used in several studies [6, 8, 9, 19, 33, 48-50].

In order to evaluate the proposed technique with multiple active faults, we adopted the multiple-fault versions of Siemens suite, as used in [13]. We used five out of seven of the Siemens suite programs in Table 2, which are Tcas, Print_tokens, Print_tokens2, Replace, and Schedule. Several faults from the former versions were combined and manually seeded into the associated programs to create programs versions with 2, 3, 4, and 5 faults each. This will help in the rigorous evaluation of our proposed technique. To further generalize our results, we used the Unix utility programs, Gzip and Sed. These programs contain both real and seeded faults, and they are relatively larger programs compared with Siemens suite programs. Sed program is used to parse textual input and apply a user-specified transformation to the input, which contains 12062 LOC with 360 test cases. Gzip is mostly used if the developer is localizing faults using the localization technique on multiple-fault programs, but it is considered to be more effective because less code is needed to be examined to locate the faults. Metrics such as wasted effort, Developer Expense (IDE). Jones et al [8] used a score to examine to find a fault. The study in [33] evaluates their technique using a metric called T-Score as follows:

\[ T - Score = \frac{\text{V}_{\text{examined}}}{V} \times 100\% \]

V is the size of program dependency graph (PDG), Vexamined is the number of statements examined before a faulty node is identified. Using this metric, the suspicious score is computed on all program statements (executable and non-executable) which will not give a realistic score to faulty statements in general. However, EXAM Score considers only executable statements which will be suitable for our single fault localization evaluation.

In this paper, we use the existing metric EXAM Score for our evaluation. It is defined as the percentage of code that a developer has to examine to find a fault.

\[ \text{EXAM Score} = \frac{\text{rank of fault}}{\text{executable statements}} \times 100\% \]

This metric will be specifically used on programs that contain only single fault in this study. Generally, for any fault localization technique, its effectiveness can be accessed and compared with EXAM Score, whereby if technique A has less EXAM Score than technique B, then technique A will be considered to be more effective because less code is needed to be examined to locate the faults. Metrics such as wasted effort has been used in studies such as [13, 37] to evaluate a fault localization technique on multiple-fault programs, but it is mostly used if the developer is localizing faults using the parallel approach to debugging. For multiple-fault programs, we propose a different metric called the Incremental Developer Expense (IDE) to aid in simultaneous localization of faults in a single diagnosis rank list. Therefore, for fault i, IDE is formulated as follows:

\[ \text{IDE} = \frac{\text{rank of fault}}{\text{executable statement}} \times 100\% - \text{previous fault expense} \]

The main objective of this metric is to allow the developer to continue searching for faults in a single diagnosis rank list until all the program faults are located. The fault localization process will not be interrupted even if the first fault is found. The process will halt when the maximum number of faults are located. For example, assume that we have a program with 3

<table>
<thead>
<tr>
<th>Program</th>
<th>Faulty version</th>
<th>Lines of code (LOC)</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print_tokens</td>
<td>7</td>
<td>565</td>
<td>4130</td>
</tr>
<tr>
<td>Print_tokens2</td>
<td>10</td>
<td>510</td>
<td>4115</td>
</tr>
<tr>
<td>Replace</td>
<td>32</td>
<td>412</td>
<td>2650</td>
</tr>
<tr>
<td>Schedule</td>
<td>9</td>
<td>307</td>
<td>2710</td>
</tr>
<tr>
<td>Schedule2</td>
<td>10</td>
<td>563</td>
<td>5542</td>
</tr>
<tr>
<td>Tcas</td>
<td>41</td>
<td>173</td>
<td>1608</td>
</tr>
<tr>
<td>Tot_info</td>
<td>23</td>
<td>406</td>
<td>1052</td>
</tr>
</tbody>
</table>

Table 3. Unix utility real-world programs
faults and 12 LOC (Figure 4) where \(s_3, s_6\), and \(s_8\) are faulty. If a single faulty diagnosis is obtained as \(D = \{8, 12, 3, 6, 5, 10 \ldots\}\), based on IDE, for faulty statement 8, this diagnosis is said to have an expense of \((1/12)*100\) which will be 8.3%. And faulty statement 3 will be \((3/12)*100 = 8.3\%\) in which its incremental developer expense will be summed up to 16.7%, and faulty statement 6 will be \((4/12)*100 = 16.7\%\) with a cumulative developer expense of 16.6%. Therefore, the total IDE to locate all faulty statements in a single diagnosis for the above example will be \(8.3\% + 16.7\% + 16.7 = 41.6\%\).

\[
Total\ IDE = \sum_{i=1}^{n} \frac{Expense(fault\ i)}{Faults/}\ i
\]

This metric calculates incremental developer expense based on how many program statements a developer has to check to find the next fault. Normally, developers do not know how many faults exist in a program when the program fails. As a stopping criterion, a developer is required to stop the fault localization process if he/she searches about 70% of the program executable codes. If the debugging effort has reached the assigned stipulated percentage (70%), then the process will be stopped. The program will be re-tested and if any test case failed, the debugging process will start all over again until the program is fault-free. Our results show that in most cases, the developer does not have to search that much to locate all faults in the faulty versions.

C. RESULTS AND DISCUSSION

Figure 5 depicts the percentage of located faults in terms of debugging efforts. Apart from evaluating the proposed FLCN technique, the following fault localization techniques which are generally known as some of the best techniques, namely Tarantula [8], Sober [51], Delta Debugging (DD) [52], Nearest Neighbour (NN) [48], SNCM [20], Intersection and Union [48], are also plotted for our comparative analysis for single-fault programs evaluation. For SNCM, we run it in our own environment. The values of the other techniques in Figure 5 are directly cited from their respective papers.

Based on the results shown in Figure 5, we can observe that FLCN performed relatively better with a very slight margin, followed by Tarantula and Sober. FLCN is capable of finding 90% of the faulty versions by examining less than 70% of the program executable codes and also finding about 99% of the faulty versions by examining less than 80% of the program source code. SNCM technique which uses centrality measures in complex network theory to locate faults is clearly outperformed by our proposed FLCN technique. However, the performance of the proposed FLCN technique is less ideal in the initial stage, whereby, by checking less than 10% of the program code we can only locate 40% of faults in all faulty versions. Tarantula and Sober, on the other hand, are capable of locating 46% and 47% of faults with the same effort respectively. We found out that this is mainly due to the sensitivity of our proposed FLCN technique with statements executed by passed test cases. Therefore, the effectiveness will probably improve if we model our network with failed test cases only instead of considering both test executions.

1) SIEMENS-M

In our multiple-fault experiment, we used 5 multiple-fault versions of Siemens suite programs with 2, 3, 4, and 5 faults across all faulty versions respectively. For the multiple-fault diagnosis, we use simultaneous approach with single developer to localize all the faults in a single diagnosis rank list. Unlike multiple fault diagnosis approaches such as one-fault-at-a-time [6] and parallelization [14] where multiple developers are needed to localize faults. Our proposed technique aims to localize all the faults in single diagnosis ranking list. In this case, a single developer is needed for the
fault localization process. This will in return reduce the maintenance cost and time-to-delivery of software systems.

Figure 6. 2-Fault versions of Siemens-M

Figure 7. 3-Fault versions of Siemens-M

Figure 8. 4-Fault versions of Siemens-M

Figure 9. 5-Fault versions of Siemens-M

Figure 6, depicts results for 2-fault versions of Siemens-M. Results show that FLCN can locate 10% of the faults by examining less than 10% of the program executable code. The overall incremental developer expense (IDE) of the 2-fault versions is 70%, meaning by checking less than 70% of all the faulty versions containing 2 faults, all the faulty versions can be found by the developer. However, Ochiai can at best locate all the faulty versions with 2 faults by checking less than 80% of the executable codes. With 3 faults, as shown in Figure 7, the developer can find 50% of the faults by examining less than 10% of the faulty versions using the FLCN technique. While by utilizing Ochiai, the developer can locate 30% of the faults by examining the same amount of faulty versions. Furthermore, using our proposed technique (FLCN) Figure 8 shows 50% of faults can be found by examining less than 20% of the faulty versions. The incremental developer expense reduces even further in Figure 9 with 5 faults active whereby examining less than
45% of program executable codes, 100% of the faults can be found in all the faulty versions. In contrast, SNCM achieves higher expense in all our multiple-fault experiments, while Ochiai performs relatively better in comparison with SNCM technique.

Our results show a trend where the more faults exist in a program, the more effective our technique is in locating faults. Moreover, even if passed test inputs execute faulty statements in a high proportion [16, 53], our proposed technique is still capable of localizing those faulty program statements simultaneously with relatively good effectiveness compared to existing similarity coefficient techniques. As a result, developer expense can be reduced even if the number of faults increases using our proposed technique because it takes into account passed test executions of a given program statement as well as failed test executions.

![Graph showing percentage of faults found vs degree centrality](image)

**Figure 10. Siemens-M (Degree centrality of program statement and its correlation with failure)**

With regards to our assertion that program statement with higher degree centrality is related to a fault. Figure 10, shows the percentage of faulty statements found with respect to their degree centrality ($Dc$) value in each faulty version of our Siemens-M programs with degree centrality of 3 and degree centrality of 2. Fig 10 illustrates that in Siemens-M programs with 2-fault, 20% of the faults were located in program statements with $Dc$ of 3 while 80% of faults are found in statements with $Dc$ of 2. Statements with $Dc$ lower than 2 have no faults across all faulty versions of our experiments.

In 3-fault versions, 33.33% of faults are found in statements with $Dc$ of 3 while 66.67% are found in statements with $Dc$ of 2. In 4-fault versions, 37.5% of faults are located in program statements with $Dc$ of 3 and 62.5% in statements with $Dc$ of 2; while in 5-fault versions, 40% of faults are found in statements with $Dc$ of 3 and 60% in statements with $Dc$ of 2. However, we observed that the higher the number of faults in a given multiple-fault program, the number of faulty statements with high $Dc$ value increases. This implies that faulty statements will be ranked at the top of the diagnosis rank list, and in return, help in simultaneous localization of multiple active faults.

2) UNIX UTILITY PROGRAMS

The Unix utility programs (Gzip and Sed) are composed of both real and seeded faults in each faulty version. To evaluate the effectiveness of our proposed technique, we compared it with Tarantula, Ochiai, and SNCM fault localization techniques. As discussed in Section 5.2, all faults are to be localized simultaneously in a single diagnosis rank list. In Figure 11, FLCN locates 45% of the faulty versions by examining less than 10% of program code and by examining less than 40% of the program code, a developer can locate 95% of all faults in the faulty versions in a single diagnosis rank list. However, Tarantula and Ochiai can locate 30% and 35% of the faulty versions by examining less than 10% respectively.

![Graph showing effectiveness comparison between FLCN, Tarantula, and SNCM](image)

**Figure 11. Effectiveness comparison between FLCN, Tarantula, and SNCM Technique for the Gzip program**

As shown in Figure 12, FLCN outperforms Tarantula, Ochiai, and SNCM where it is capable of identifying all the locations of faulty statements by examining less than 50% of the faulty versions. Using simultaneous approach to debugging caused the other techniques to lose their effectiveness. In general, the proposed FLCN technique surpasses Tarantula, Ochiai, and SNCM in locating all the faults simultaneously.
Figure 13. Gzip (Degree centrality of program statement and its correlation with failure)

Figure 14. Sed (Degree centrality of program statement and its correlation with failure)

Figure 13 and Figure 14 show Dc correlation with failure across all faulty versions of both Gzip and Sed respectively. For Gzip (Figure 13), program statements with Dc values of 4, 3, and 2 contain 10%, 5%, and 85% of the faults in all faulty versions respectively. While in Sed (Figure 14), 80% of the faults are found in program statements with Dc of 2, while 20% of the faults can be found in program statements with Dc of 3. The trend shows that on average, 60% - 70% of the faults are located in program statements with Dc of 2. The closeness centrality (Cc) plays a critical role in identifying faulty statements. We found out that faulty statements that have Dc value of 2 have relatively lower Cc in comparison to non-faulty statements. Therefore, using our proposed ranking metric, the difference between these two values are computed to rank the faulty statements.

VI. THREAT TO VALIDITY

Even though the empirical study presented in this paper indicates the usefulness of the simultaneous approach to localizing multiple faults, there are threats to the validity of the empirical results that readers should take into account when interpreting the results.

The threat to the external validity of an experiment limits generalization of the results. The use of small to medium-sized C programs is also a threat to external validity. Therefore, we cannot claim that the results presented are generalized to other larger programs and programs in different programming languages. Another threat to external validity is the way multiple-fault versions are created, seeded faults are used whereby the results might be different if real faults are used. However, we minimize this threat by using some real programs with both real and seeded faults.

The threat to internal validity occurs when there are unknown relationships between dependent and independent variables. In the simultaneous approach, a developer might reach the total expense threshold based on the evaluation stopping criterion without identifying all the faults. This would imply the use of sequential debugging approach such as one-fault-at-a-time approach. This might in turn increase the developer debugging expense. We also postulated that program statements with higher degree centrality are more suspicious, in some cases, non-faulty statements can have higher degree centrality.

Deploying a single developer to find and fix faults simultaneously can increase software time-to-delivery in some cases rather than multiple developers. Further studies are needed to investigate this issue.

VII. CONCLUSION

In this paper, we proposed a novel fault localization technique called Fault Localization based on Complex Network Theory (FLCN) to localize faults in both single-fault and multiple-fault programs. Program spectra are modeled as a complex network and two centrality measures were adopted for fault localization. In addition, a ranking formula is proposed for ranking suspicious program statements. The idea is that faulty statements behave differently in the network and these unique behaviors can be captured using graph theory metrics. FLCN models a complex network using program spectra of both passed and failed test executions with the same priority given to each executable statement regardless whether it is executed by passed test cases or failed test cases. This idea is poised to tackle issues of interference between faults that hinder the localization effectiveness of SBFL techniques.

Our experimental results show FLCN is more effective in both programs with single and multiple faults. However, in single-fault versions, FLCN performs less ideally compared to Tarantula and Sober in the first 10% of fault examination effort, basically due to its high sensitivity to program statements executed by passed test inputs. Nonetheless, in the overall debugging effort, the proposed FLCN technique outperforms all the other fault localization techniques. Moreover, in programs with multiple active faults, FLCN surpasses both Tarantula, Ochiai, and SNM where it is capable of identifying 50% of the faulty versions by examining only less than 20% of the programs executable.
codes in versions with 2, 3, 4, and 5 simultaneous active faults. Two metrics which are, Exam score (single-fault) and incremental developer expense (IDE) (multiple-faults) were used to evaluate our proposed FLCN technique. These metrics aid the developer to calculate his/her total expense during fault localization. IDE is designed to be generic and can be utilized to evaluate research studies using simultaneous debugging approach. Finally, we postulated that degree centrality (DC) indicate how important a node is in the fault localization process. While, closeness centrality (CC) aids in localizing faulty statements that are close to each other or failed due to different independent faults. We concluded that faulty program statements have higher degree centrality and lower closeness centrality value. However further study on different software programs can be done to fully understand this phenomenon.

We identified a number of research directions for our future work that results from this work. Firstly, single-fault localization can be improved by considering failed test executions alone to model our complex network because modeling the network with both passed and failed test executions has resulted in low effectiveness in single-fault programs. Moreover, to further improve localization accuracy in the presence of multiple faults, we determined the localization effectiveness in programs with multiple faults by proposing a modified modularity clustering algorithm [54] to further isolate independent faults into separate clusters and localize them independently by multiple developers. The perceived advantage of this approach is to reduce the amount of program executable codes each developer has to check to find the faults. Finally, we plan to perform further experiments on larger subject programs with a varying number of faults.

ACKNOWLEDGMENT
This work was carried out within the framework of the research project FP001-2016 under the Fundamental Research Grant Scheme provided by Ministry of Higher Education, Malaysia.

REFERENCES


