Effective Lightweight Software Fault Localization Combined with Fault Context and Test Suite Optimization

Amol Saxena, Roheet Bhatnagar, *Member, IAENG*, and Devesh Kumar Srivastava, *Member, IAENG*

***Abstract*—Locating software faults during debugging is crucial yet challenging. Automated techniques, such as spectrum-based fault localization (SBFL), aid developers in efficiently localizing faults by analyzing program execution information and utilizing statistical approaches to rank program entities according to their suspiciousness. SBFL is also known as lightweight fault localization because of its scalability and minimal computational overhead. While essential, there has been limited research on how test suites affect fault localization. In this paper, we show how test suites impact fault localization and how they can be optimized for better results. SBFL techniques have some inherent limitations, especially in diagnosing faults within loop bodies or iteration statements. Additionally, identical suspiciousness levels can result in ties. While SBFL techniques effectively rank faulty program entities among the Top-N suspicious entities, they might not consistently position the faulty entity within the initial few positions. To address these research gaps, this paper proposes a hybrid approach that combines test suite optimization, statement execution frequency, and fault context concepts to enhance the performance of existing SBFL techniques in single fault scenarios. We evaluate our approach using three popular SBFL methods (Ochiai, Jaccard, and DStar) on Siemens benchmarks and four large real-world programs (flex, grep, sed, and space) with their test suites. The results demonstrate a significant enhancement in fault localization performance when applying our proposed approach to existing SBFL methods. For example, when applied to Ochiai, it reduces examined statements by 62.76% and 65.23% on average for the two test suites, respectively. Furthermore, it identifies 52% of faults by examining only 1% or less of the code and locates 60% of faults by analyzing only 0.1% or less of the code in Siemens and four large real-world programs, respectively. Similar improvements are observed when our approach is applied to Jaccard and DStar methods on the same test suites. We also show that our results are statistically significant, validating that our approach substantially improves the performance of existing SBFL techniques.**

***Index Terms*—fault context, program spectrum, spectrum-based fault localization, statement execution frequency, suspiciousness, testing and debugging, test suite optimization.**

Manuscript received May 16, 2023; revised February 21, 2024.

Amol Saxena is a PhD candidate of Computer Sc. & Engineering Department, SCSE, Manipal University Jaipur, Jaipur, Rajasthan, India-303007 (email: amolsaxena2015@gmail.com).

Dr. Roheet Bhatnagar is a Professor of Computer Sc. & Engineering Department, SCSE, Manipal University Jaipur, Jaipur, Rajasthan, India-303007 (Phone: +91-8003897115; email: roheet.bhatnagar@jaipur.manipal.edu).

Dr. Devesh Kumar Srivastava is a Professor of Information Technology Department, SIT, Manipal University Jaipur, Jaipur, Rajasthan, India-303007 (email: devesh988@yahoo.com).

I. Introduction

Software is becoming increasingly large and complex as a result of its widespread use and adoption. Software plays a critical role in various safety-critical systems in industries such as healthcare, defense, aviation, nuclear energy and so on. However, faults in software are inevitable, and the rapid growth and intricate nature of software systems have resulted in more faults leading to software failures, resulting in substantial losses [1], [2]. Testing and debugging are crucial activities in the software development process but are generally very costly. They can account for up to 75% of the total software development costs [3]. The process of testing and debugging involves three steps: first, identifying the scenarios in which a program fails, second, locating the faults responsible for program failures, and third, fixing the faults. The second process is the most difficult, tedious, and costly in terms of the developer's time and effort [4], [5]. In software engineering literature this second process is popularly known as software fault localization. Fault localization can be defined as detecting and identifying the locations of faults in software systems. In this process, the programs are executed with predefined test cases, and execution information is utilized to locate faults. Earlier the process of fault localization was carried out manually and was known to be a very tedious, time consuming and prohibitively expensive, especially in case of large scale and complex software systems. The manual fault localization also depends on the expertise, experience and judgment of the developer who is performing the debugging task. Automated fault localization techniques have emerged as a solution to the drawbacks of manual fault localization methods, as they require minimal or no human intervention to locate faults.

Over the past few decades, researchers have introduced numerous automated software fault localization techniques. These techniques include spectrum-based methods, statistics-based approaches [6], [7], model-based methods [8], machine learning-based techniques [9], [10], slice-based methods, and program state-based approaches. These automated techniques improve software quality, reliability, and reduce delivery time. Spectrum-based fault localization (SBFL), according to recent studies, is the most widely used and effective technique due to its superior scalability and low computational overhead [11] characteristics. Due to these reasons SBFL is also referred to as lightweight fault localization in the software engineering literature. Moreover, SBFL can be used to locate faults with little or no

knowledge of the semantics of the program being debugged. The objective of this paper is to introduce a framework that enhances the precision and efficiency of lightweight software fault localization (i.e. SBFL), specifically in a single fault scenario. Typically, a program consists of various entities, including statements, statement blocks, predicates, and methods. However, this study concentrates solely on isolating faults in program statements when only one fault is present. Therefore, whenever the term program entity is used in this paper, it should be interpreted as referring to program statements, unless stated otherwise.

Test suites are essential, as they drive program execution and enable the collection of program spectrum information for initiating the fault localization process. However, prior studies have not given enough emphasis on the adequacy of test suites in this context. Our study improves fault localization performance by using optimized test suites. Current SBFL techniques have limitations in accurately diagnosing faults, especially in loops and iterations, as they only consider whether a statement is executed or not. Additionally, identical suspiciousness levels can result in ties. To address these issues, we incorporate statement execution frequency information into the SBFL. While SBFL techniques effectively rank faulty program entities among the Top-N suspicious entities, they might not consistently position the faulty entity within the initial few positions. To enhance the prioritization of faulty program entities, this paper proposes incorporating the concept of fault context into SBFL.

To address the above mentioned limitations of traditional SBFL techniques, we propose a hybrid approach that combines the concepts of test suite optimization, statement execution frequency, and fault context concepts to improve the effectiveness of existing SBFL techniques in single fault scenarios. In order to evaluate our proposed approach, we conducted a thorough empirical study on two benchmark test suites: Siemens and large programs (flex, grep, sed, and space), comparing its performance with the existing classic SBFL techniques. We employ four metrics to assess the efficacy of the proposed approach in comparison to the traditional SBFL methods. The four metrics utilized are Exam Score, Cumulative Number of Statements Examined, Top-N, and Wilcoxon Signed-Rank Test. Our results demonstrate that the proposed approach outperforms existing SBFL methods in most of the cases. Our approach on average reduces examined statements by 62.76% for Siemens programs and 65.23% for large real-world programs when applied to classic Ochiai. It detects 52% of the faults in Siemens and 60% of the faults in large real-world programs by analyzing less than or equal to 1% and 0.1% of the code, respectively, outperforming existing classic Ochiai method. Furthermore, the proposed approach achieves better Top-N results as it locates 20% of the faults at top-1, 60% of the faults at top-5 positions for Siemens programs, similarly 7% of the faults at top-1 and 27% of the faults at top-5 positions for large real-world programs. We also test our proposed approach on two other SBFL techniques, Jaccard and DStar, and find that it significantly improves fault localization effectiveness for both the techniques. The main contribution of this paper can be summarized as follows.

1. This paper investigates how test suites impact fault localization and how they can be optimized for improved software fault localization performance.
2. To address the inherent limitations of existing spectrum-based fault localization (SBFL) techniques, such as ties in the ranking of statements with the same suspiciousness scores and inaccurate diagnoses of faults occurring within loop bodies or iteration statements, we propose incorporating the concept of statement execution frequency into SBFL.
3. To demonstrate how the combination of fault context with spectrum-based fault localization can enhance the accuracy of identifying faulty program entities, resulting in an improved absolute rank of such entities.
4. We propose a hybrid approach that combines test suite optimization, statement execution frequency, and fault context concepts to enhance the effectiveness of spectrum-based software fault localization in single fault scenarios. We evaluate its effectiveness on the Siemens benchmark and four large real-world programs (flex, sed, grep, and space). The results demonstrate that the proposed approach significantly improves the performance of existing SBFL techniques.

The subsequent sections of the paper are structured as follows. Section II highlights the background information necessary for a comprehensive understanding of the research context. Section II-A introduces the lightweight (or spectrum-based) fault localization techniques. Section II-B explains test suite optimization process and how optimized test suites can be used to improve the performance of lightweight fault localization. Section II-C describes how statement execution frequency information can be incorporated into SBFL formulas to address some of the inherent issues associated with SBFL. Section II-D explains the concept of fault context, which can be combined with SBFL techniques to further improve the absolute rank of faulty program entities. Motivational examples are provided to illustrate each of the concepts discussed in subsections II-B, II-C, and II-D. Section III presents our proposed approach/framework with the help of a motivational example. Section IV presents the empirical study, covering research questions, experimental setup, evaluation metrics, results, and discussions. The related literature review is provided in Section V, and threats to validity is summarized in Section VI. Section VII concludes the paper highlighting the scope of the future work.

II. Background and Motivation

The aim of this paper is to improve the effectiveness and precision of conventional spectrum-based fault localization (SBFL) methods. This section presents a fundamental introduction to SBFL, also referred to as lightweight software fault localization (LFL). To gain a better understanding of the methodology proposed in this paper, we elaborate on how the ideas of test suite optimization, statement execution frequency, and fault context can be utilized to enhance the accuracy and effectiveness of lightweight software fault localization from a single fault perspective. Motivational working examples have been used to illustrate all these concepts.

A. Lightweight Software Fault Localization

In the domain of fault localization research, spectrum-based fault localization (SBFL) is frequently termed as lightweight software fault localization due to its advantageous features of minimal computational overhead and strong scalability. In this paper, the terms SBFL and lightweight software fault localization (LFL) are utilized interchangeably.

SBFL employs program spectrum information, which is dynamically collected when the program under debugging is executed with predefined test cases. The program spectrum is built from the dynamic coverage information acquired from the test execution results of the faulty program and the corresponding test results (passed/ failed). A program is said to be passed when it gives the expected results, and is considered as failed otherwise. A program statement is said to be hit when it gets executed in the testing process. This statement hit information or statement coverage information collected from the execution of a program with several test cases is called the statement hit spectra. The correlation between the test results (passed/ failed) and statement hit spectra is utilized by the SBFL in order to locate the fault.

SBFL techniques employ formulas based on similarity coefficients to determine the suspiciousness of program statements, which indicates the likelihood of those statements being faulty. These methods operate on the fundamental concept that if the execution pattern of a statement is similar to that of failed test cases, the statement is more likely to be faulty and thus more suspicious. Conversely, if the execution pattern of a statement differs from the failed test cases' execution pattern, the statement is considered less suspicious. These techniques use similarity coefficient-based methods to measure the degree of similarity between the statement's execution pattern and the failed test cases, which is then used to determine the statement's suspiciousness.

A debugging report that contains statements ranked in descending order according to their suspiciousness scores is generated to perform fault localization. In response to this perception, researchers have proposed a number of metrics based on similarity coefficients that calculate program statement suspiciousness scores. Examples include Tarantula [12], Ochiai [11], Jaccard [13], DStar [14], Zoltar-S [15], [16], Crosstab [17] etc. These metrics, which are based on similarity coefficients, are also known as ranking metrics or ranking heuristics. In essence, these are statistical formulas that compute the suspiciousness of statements in the faulty program using program spectrum information.

		M Test Cases				
	Program	T_1	T_2	…	T_M	Suspiciousness
N Statements	S_1	E_{11}	E_{12}	…	E_{1M}	P_1
	S_2	E_{21}	E_{22}	…	E_{2M}	P_2
	…	…	…	…	…	
	S_N	E_{N1}	E_{N2}	…	E_{NM}	P_N
	Result (Pass/ Fail)	R_1	R_2	…	R_M	

Fig. 1. Input to SBFL (Program Spectrum)

We now formally define the process of spectrum-based fault localization. Consider a faulty program P, and let $S=\{S_1, S_2, \ldots, S_N\}$ represent its statements and $T=\{T_1, T_2, \ldots, T_M\}$is the test suite, which has M test cases. Fig. 1 shows a two dimensional matrix of size $(N+1) \times M$, which represents the program spectrum. This program spectrum is the input to the SBFL process. An element E_{ij} has a value of 1 if test case T_i covers the statement S_j, otherwise it has a value of 0. The last row of the matrix is the result vector R, which represents the execution result in terms of pass (P) or fail (F) of the program P when run with test case T_i, where i=1 to M. When test T_i fails, result R_i is 1 (or F), but if T_i passes (i.e. gives the expected output), result R_i is 0 (or P) (or vice versa). The value of each statement S_i's suspiciousness is shown in the table's last column. The higher the suspiciousness value, the more likely it is that the statement is faulty.

SBFL utilizes the comparison of the statement vector and result vector in the matrix shown in Fig. 1 to determine the suspiciousness of a given statement. To simplify similarity calculations, SBFL defines the statistical variables outlined as follows. N_{CF} stands for the test cases count that cover a statement and also result in failure, whereas N_{UF} represents the count of test cases that fail without covering a statement. On the other hand, N_{CS} stands for the test cases count that both pass and cover a statement, while N_{US} represents the test cases count that pass but do not cover a statement. Furthermore, the counts of test cases that cover and do not cover a statement are denoted by N_C and N_U, respectively. N_S and N_F, on the other hand, refer to the total number of passing and failing test cases, respectively. These statistical variables have been used to propose several similarity coefficient based metrics/formulas by researchers in past years. Some of them are given below in Table I.

TABLE I
SPECTRUM-BASED FAULT LOCALIZATION TECHNIQUES

Sr. No.	Coefficient	Formula (Algebraic Form)
1	Tarantula	$\frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F}+\frac{N_{CS}}{N_S}}$
2	Jaccard	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
3	Ochiai	$\frac{N_{CF}}{\sqrt{N_F \times (N_{CF}+N_{CS})}}$
4	DStar†	$\frac{(N_{CF})^*}{N_{CS}+N_{UF}}$
5	Kulczynski	$\frac{N_{CF}}{N_{UF}+N_{CS}}$
6	Dice	$\frac{2N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
7	Ample	$\frac{N_{CF}}{N_{CF}+N_{UF}}-\frac{N_{CS}}{N_{CS}+N_{US}}$
8	Anderberg	$\frac{N_{CF}}{N_{CF}+2(N_{UF}+N_{CS})}$
9	Zoltar	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}+\frac{10000 \times N_{UF} \times N_{CS}}{N_{CF}}}$

† In our experiments, we consider the value of * = 2, as that value is thoroughly investigated in the fault localization literature [14].

B. Test Suite Optimization

Because the program spectrum information is dynamically collected by running the faulty program with the necessary input supplied through individual test cases of a test suite, the accuracy and efficiency of fault localization largely depend on the test suite being used in the fault localization process. As part of the fault localization process, this dynamic program spectrum data is analyzed (in our case SBFL). The majority of fault localization studies do not much emphasize on test suite sufficiency in terms of its capability to increase fault localization effectiveness. Investigating the relation between test suites and fault localization is therefore important. Past researches show that in order to enhance the fault localization efficiency the concept of optimized test suites was used by very few findings, although many studies have suggested ways to enhance fault localization performance by other means [18]–[22]. The effectiveness of fault localization was examined by Yan Lei et al. [19] in relation to the positive or negative impact of various test suite components. In this case, a positive impact indicates an improvement in the fault localization effectiveness, whereas a negative impact indicates a reduction in the fault localization accuracy.

As per their investigation, the accuracy of identifying faults in a program is negatively impacted by faulty statements that are executed by passing test cases within a test suite. Contrarily, faulty program entities (i.e., statements in our study) that are not executed by passing test cases have a positive effect on the suspiciousness rank of those program entities, which in turn enhances fault localization performance. The performance of fault localization, however, always increases when a test suite includes failing test cases. The passing test cases of a test suite are largely responsible for the variations in SBFL performance, as they occasionally execute the faulty statement (S_f) and occasionally do not. The final ranking of S_f is negatively affected if passing test cases exercise (execute) the S_f because in that case the suspiciousness of S_f is reduced. In their work, Yan Lei et al. [19] introduced a measure known as Passing Test Discrimination (PTD) to evaluate the comparative efficiency of two test suites in enhancing the suspiciousness of faulty statements. PTD is calculated as the ratio of the number of test cases that pass in a test suite but do not execute the faulty statement to the overall count of passing test cases. As a result, PTD can be utilized as an indicator to assess how well a test suite can detect faults in a program. It is recommended to strive for a relatively high PTD during the creation of a test suite to enhance fault localization accuracy.

The PTD of a test suite T can be increased by the use of a straightforward heuristic explained as follows [23]. Consider S as a set of statements that includes all those statements executed by all failing test cases of a test suite T. Suppose t is a passing test case, and if t executes most of the statements in S, then there is a high probability that t will execute the faulty statement. So, from T, we may remove a test case t if there is a high similarity between the set of statements executed by t and S. Therefore, in this way, with the application of PTD metric, an existing test suite can be optimized, or a new optimized test suite can be created with a sufficiently high PTD score and effectively used in the fault localization process.

In the following paragraph we formally explain the above heuristic. A passing test case can be removed from the test suite if it is likely to execute a statement that is faulty. To identify such passing test cases, dynamic statement coverage information (program spectrum) is utilized from test cases that give failing results. The fundamental idea is that when the statement execution coverage of a successful test case closely matches that of the unsuccessful test cases (or failing) in the test suite, there is a greater likelihood that the successful test case will run (or cover) a faulty statement. A heuristic as explained below can be employed to determine the degree of similarity between passing and failing test cases. The minimum suspicious set (MSS) is the collection of statements covered by all the failed test cases, and the heuristic employs this concept. Due to the fact that the faulty statement is responsible for the failure of a test case, it makes sense that the MSS contains the faulty statement. Now suppose t is a passing test case and S is the set of statements executed by t, and if there is a much similarity between S and MSS that is $|S \cap MSS|/|MSS| > \alpha$, where

TABLE II
ILLUSTRATION OF FAULT LOCALIZATION WITH RANDOM TEST SUITE (TESTSUITE-1)

Stmt. No.	Program	T1	T2	T3	T4	T5	T6	N_{CF}	N_{CS}	Susp. (Ochiai)	Susp. Rank
S1	void main(int argc,char *argv[])	1	1	1	1	1	1	3	3	0.71	3
S2	{ char strch[100];	0	0	0	0	0	0	0	0	0.00	13
S3	int alpha, digit, ch, i;	0	0	0	0	0	0	0	0	0.00	13
S4	alpha = digit = ch = i = 0;	1	1	1	1	1	1	3	3	0.71	3
S5	strcpy(strch,argv[1]);	1	1	1	1	1	1	3	3	0.71	3
S6	while(strch[i]!='\0')	1	1	1	1	1	1	3	3	0.71	3
S7	{ if((strch[i]>='a' && strch[i]<='z') || (strch[i]>='A' && strch[i]<='Z'))	1	1	1	1	1	1	3	3	0.71	3
S8	alpha++;	0	1	1	1	1	1	3	2	0.77	2
S9	else if(strch[i]>'0' && strch[i]<='9') //correct strch[i]>='0'	1	1	1	1	1	1	3	3	0.71	3
S10	digit++;	0	0	0	1	1	0	2	0	0.82	1
S11	else	0	0	0	0	0	0	0	0	0.00	13
S12	ch++;	1	1	1	1	1	1	3	3	0.71	3
S13	i++;}	1	1	1	1	1	1	3	3	0.71	3
S14	printf("Alphabets =%d Digits=%d Special characters = %d", alpha,digit,ch);	1	1	1	1	1	1	3	3	0.71	3
S15	}	1	1	1	1	1	1	3	3	0.71	3
	Result (Pass=P, Fail=F)	P	P	P	F	F	F				

TABLE III
ILLUSTRATION OF FAULT LOCALIZATION WITH OPTIMIZED TEST SUITE (TESTSUITE-2)

Stmt. No.	Program	T1	T2	T3	T4	T5	T6	N_{CF}	N_{CS}	Susp. (Ochiai)	Susp. Rank
S1	void main(int argc,char *argv[])	1	1	1	1	1	1	3	3	0.71	4
S2	{ char strch[100];	0	0	0	0	0	0	0	0	0.00	13
S3	int alpha, digit, ch, i;	0	0	0	0	0	0	0	0	0.00	13
S4	alpha = digit = ch = i = 0;	1	1	1	1	1	1	3	3	0.71	4
S5	strcpy(strch,argv[1]);	1	1	1	1	1	1	3	3	0.71	4
S6	while(strch [i]!='\0')	1	1	1	1	1	1	3	3	0.71	4
S7	{ if((strch [i]>='a' && strch [i]<='z') || (strch [i]>='A' && strch[i]<='Z'))	1	1	1	1	1	1	3	3	0.71	4
S8	alpha++;	1	1	1	1	1	1	3	3	0.71	4
S9	else if(strch[i]>'0' && strch[i]<='9') //correct strch[i]>='0'	0	0	1	1	1	1	3	1	0.87	1
S10	digit++;	0	0	0	1	1	0	2	0	0.82	3
S11	else	0	0	0	0	0	0	0	0	0.00	13
S12	ch++;	0	0	1	1	1	1	3	1	0.87	1
S13	i++;}	1	1	1	1	1	1	3	3	0.71	4
S14	printf("Alphabets =%d Digits=%d Special characters = %d", alpha,digit,ch);	1	1	1	1	1	1	3	3	0.71	4
S15	}	1	1	1	1	1	1	3	3	0.71	4
	Result (Pass=P, Fail=F)	P	P	P	F	F	F				

α is a threshold; then t can be removed as we believe that there is a certain probability that t will execute the faulty statement. As there is no perfect practical value for this threshold, conducting experimental studies are necessary to determine the appropriate value of α for a particular practical settings.

To illustrate the process of test suite optimization, let us consider a simple working example. Table II presents a program that counts the number of alphabets, digits, and special characters in its input, with an operator mutation fault seeded at statement S9. The correct statement is also provided as a comment at the same place. Six test cases are executed, with three (T1, T2, and T3) producing correct results (passed test cases), and the remaining three (T4, T5, and T6) failing to produce the desired outputs (i.e., failed test cases). The statement hit spectra corresponding to the execution of these test cases are given from column three to column eight. If an entry contains the digit "1", it implies that the statement is covered by the corresponding test case, whereas a "0" denotes that the statement is not covered by the test case. For each statement, the values of N_{CF} and N_{CS} are given in columns nine and ten, respectively. According to the definition of Ochiai similarity coefficient metric (listed in Table I) the suspiciousness scores are computed and shown in column eleven. Each statement is ranked in descending order of its suspiciousness score in the last column. That is, the statement with the highest chance of being faulty is ranked first.

In order to improve the accuracy of fault localization, we can utilize test suite optimization. To illustrate this concept, we will use a sample program. As mentioned previously in this section, the localization of faults can be negatively impacted by passing test cases that execute statements containing faults.

The example program is run with two different sets of test suites, testsuite-1 and testsuite-2, as shown in Table II and Table III, respectively. The first test suite (testsuite-1) used in the example shown in Table II is a random one with a Passing Test Discrimination (PTD) measure of 0 (i.e. 0/3), because in this case all three successful test cases (T1, T2, and T3) have executed the faulty statement. The example program executed with the second test suite (testsuite-2) in Table III, on the other hand, is an optimized test suite with a PTD measure of 66.66% (i.e. 2/3) because two of the three passing test cases do not execute the faulty statement S9. As stated at the beginning of this section, increasing the value of PTD makes a test suite more suitable for fault localization.

We now present the idea of test suite optimization in a different way, in which we measure the similarity between the set of statements executed by passing test cases and failing test cases with the help of the minimum suspicious set (MSS).

From the analysis of Table II, it is evident that the passing test cases (T1, T2, and T3) cover 83.33%, 91.66%, and 91.66% of the Minimum Suspicious Set (MSS) of testsuite-1, respectively. This significant overlap between the statements covered by the passing test cases and the MSS raises the probability that the faulty statements are executed by passing test cases, thereby decreasing the precision of fault localization. As a consequence, testsuite-1 (Table II) can be regarded as a random test suite (i.e. inefficient for fault localization).

Successful (passing) test cases T1, T2, and T3 in testsuite-2 (Table III) share 75%, 75%, and 91.66% of statements with MSS of testsuite-2, respectively. Because T1 and T2 (successful or passing test cases) share a smaller portion of MSS in test suite-2, they are less likely to execute the faulty statement. As a result, testsuite-2 is an optimized test suite that will improve fault localization performance. Upon examining the last columns of Table II and Table III, it is evident that Test Suite 1 necessitates eight searches to locate the faulty statement (S9), whereas Test Suite 2 (an optimized version) requires only one search.

The motivational example given above thus clarifies how test suite optimization can enhance fault localization performance.

C. Incorporating statement execution frequency information into lightweight fault localization

The existing SBFL techniques (some of them are listed in Table I) have limited diagnostic capabilities as they represent statement coverage using binary information (1 or 0). That means if a statement is covered or not covered by a

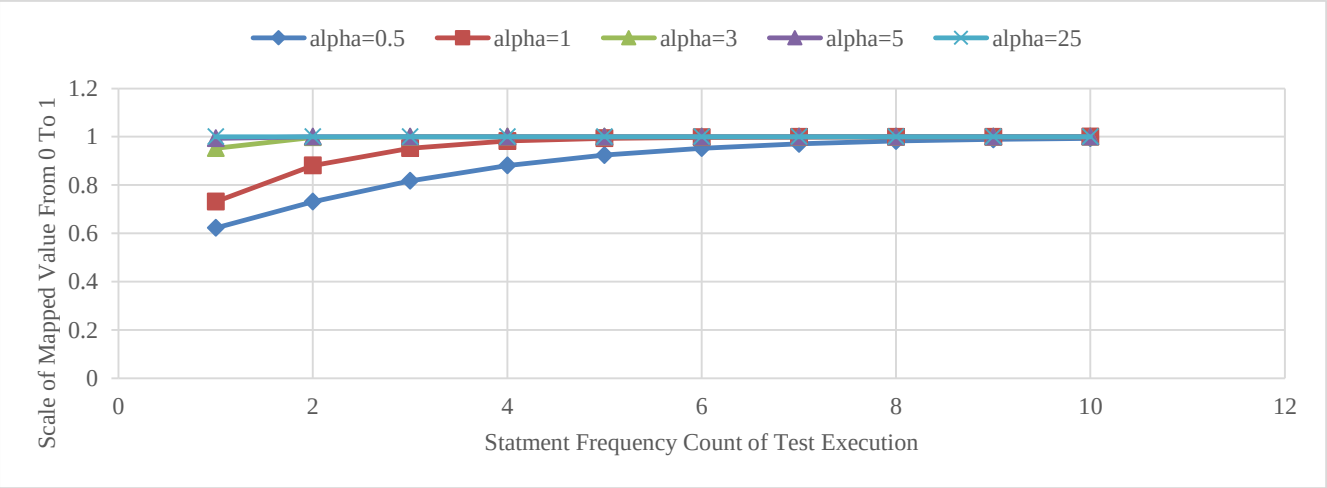


Fig. 2. Comparison between the statement execution frequency count of test executions and their corresponding mapped values

particular test case is represented by 1 or 0, respectively. These methods ignore the information that how many times a statement is executed by a test case. Consequently, the fault localization accuracies of SBFL methods are limited when faults are there in the loop bodies or iteration statements. Moreover, identical suspiciousness levels can result in ties.

To improve fault localization effectiveness we will use the concept of statement execution frequencies in program spectrum information instead of binary execution count. The program spectrum information will be collected from the execution of respective test cases. This concept is also known as spectral frequencies in fault localization studies. Statement frequency would impact suspiciousness of a statement as it indicates how many times that statement is executed by a corresponding test case. We will use the sigmoid function to map or normalize this additional information of statement execution frequency count to a value in the range of [0, 1), means greater than or equal to 0 and less than 1. Sigmoid function is also known as logistic function and has been used in various domains such as economics, biology and machine learning. In fault localization domain we adapt the definition of the sigmoid function given in (1).

$$K(n_{st}) = \begin{cases} \frac{1}{e^{-\alpha * n_{st}+1}} & if\ n_{st} > 0 \\ 0 & otherwise \end{cases} \quad (1)$$

Here, n is the frequency of statement s when executed with test case t, α is a constant value. The above function K is evaluated for mapping the nonzero frequency counts to the range of [0, 1). The function K returns 0, if the statement was not executed, that means frequency count is zero. The above function can also be referred as statement frequency weighting function.

Fig. 2 shows a simple mapping of the frequency count of statements ranging from 0 to 10 for different alpha values. We show this mapping for five α values (α = 0.5, 1, 3, 5 and 25). We can see that with the increase of α value the frequency response curve (weighting function) gets sharper. The mapping is equivalent to binary function 0 or 1 when the alpha (α) value gets very large (i.e. α = 25), and this is similar to binary program spectrum information. We now show how statement frequency information of test coverage when incorporated into SBFL methods can perform better in terms of improving the accuracy of pinpointing faults within a program as compared to using traditional binary execution information. To substantiate this fact we present a working example as shown in Table IV, Table V and Table VI. The example program counts the number of alphabets, digits and special characters in its input. An operator mutation fault has been seeded in statement S9. The correct statement should be “else if (strch[i]>='0' && strch[i] <='9')”. The program is exercised with six test cases T1, T2, T3, T4, T5, and T6, out of which the first three are passing and the last three are failing test cases. In other words, first three test cases (T1, T2 & T3) are giving the correct output, but T4, T5, and T6 are not giving the expected output. Table IV shows the statement hit spectra in binary form. That means, if a statement is executed at least once by a test case then it is shown as ‘1’ and otherwise it is shown as ‘0’. Table V shows the number of times each statement is executed by a corresponding test case. In this case the statement hit spectra consists of frequency count of statements executed by a corresponding test case. The computation of statement suspiciousness in our example program using the frequency count information is demonstrated in Table VI. The statement frequency weighting function (adapted sigmoid function) as given in (1) is used to map the frequency counts to the values between 0 and 1 and the mapped values are shown in Table VI for each test case T1 to T6. The last rows of Table IV and Table VI show program execution results as pass (P) or fail (F) of the respective test cases. To compute the suspiciousness score we use Ochiai similarity coefficient metric as defined in Table I. The second last and last columns of Table IV and Table VI show the suspiciousness score and rank in descending order of suspiciousness scores of each program statement, respectively. That means the statements with higher suspiciousness scores are ranked first as these statements are more likely to be faulty. We can observe from Table V that the statement S6 executes 5 and 6 times when program is executed with test cases T1 and T2, respectively. The corresponding mapped values according to the statement weighting function as given in (1), are 0.92 and 0.95, respectively as shown in Table VI.

Table IV shows that statements {S1, S4, S5, S6, S7, S9, S12, S13, S14, and S15} share identical suspiciousness values, making it difficult to pinpoint the likely faulty statement. Specifically, it requires eight searches to identify the faulty statement S9, as depicted in Table IV. By

integrating statement execution count information into the SBFL formula (Table VI), the faulty statement can be identified with a 50% improvement, requiring only four searches. The reason is that, we are able to further differentiate the probability of a statement being faulty because differing values are assigned to N_{CF} and N_{CS} according to the statement execution counts. We can see that the faulty statement S9 has greater N_{CF} value (i.e. 2.32) than the N_{CS} value (i.e. 2.19), and therefore has higher suspiciousness rank. Therefore, the given working example suggests that the concept of statement execution frequency information can improve the performance of existing SBFL techniques.

TABLE IV
SUSPICIOUSNESS CALCULATION USING BINARY INFORMATION FROM TEST EXECUTIONS

Stmt. No.	Program	T1	T2	T3	T4	T5	T6	N_{CF}	N_{CS}	Susp. (Ochiai)	Rank
S1	void main(int argc,char *argv[])	1	1	1	1	1	1	3	3	0.71	3
S2	{ char strch[100];	0	0	0	0	0	0	0	0	0.00	13
S3	int alp, digit, ch, i;	0	0	0	0	0	0	0	0	0.00	14
S4	alp = digit = ch = i = 0;	1	1	1	1	1	1	3	3	0.71	4
S5	strcpy(strch,argv[1]);	1	1	1	1	1	1	3	3	0.71	5
S6	while(strch[i]!='\0')	1	1	1	1	1	1	3	3	0.71	6
S7	{ if((strch[i]>='a' && strch[i]<='z') || (strch[i]>='A' && strch[i]<='Z'))	1	1	1	1	1	1	3	3	0.71	7
S8	alp++;	0	1	1	1	1	1	3	2	0.77	2
S9	else if(strch[i]>'0' && strch[i]<='9') //correct strch[i]>='0'	1	1	1	1	1	1	3	3	0.71	8
S10	digit++;	0	0	0	1	1	0	2	0	0.82	1
S11	else	0	0	0	0	0	0	0	0	0.00	15
S12	ch++;	1	1	1	1	1	1	3	3	0.71	9
S13	i++;}	1	1	1	1	1	1	3	3	0.71	10
S14	printf("Alphabets =%d Digits=%d Special characters = %d", alp,digit,ch);	1	1	1	1	1	1	3	3	0.71	11
S15	}	1	1	1	1	1	1	3	3	0.71	12
	Result (Pass=P, Fail=F)	P	P	P	F	F	F				

TABLE V
STATEMENT EXECUTION FREQUENCY COUNT INFORMATION FROM TEST EXECUTIONS

Stmt. No.	Program	T1	T2	T3	T4	T5	T6
S1	void main(int argc,char *argv[])	1	1	1	1	1	1
S2	{ char strch[100];	0	0	0	0	0	0
S3	int alp, digit, ch, i;	0	0	0	0	0	0
S4	alp = digit = ch = i = 0;	1	1	1	1	1	1
S5	strcpy(strch,argv[1]);	1	1	1	1	1	1
S6	while(strch[i]!='\0')	5	6	5	6	8	4
S7	{ if((strch[i]>='a' && strch[i]<='z') || (strch[i]>='A' && strch[i]<='Z'))	4	5	4	5	7	3
S8	alp++;	2	3	2	2	3	2
S9	else if(strch[i]>'0' && strch[i]<='9') //correct strch[i]>='0'	2	2	2	3	4	1
S10	digit++;	0	2	0	2	2	0
S11	else	0	0	0	0	0	0
S12	ch++;	2	0	2	1	2	1
S13	i++;}	4	5	4	5	7	3
S14	printf("Alphabets =%d Digits=%d Special characters = %d", alp,digit,ch);	1	1	1	1	1	1
S15	}	1	1	1	1	1	1

TABLE VI
SUSPICIOUSNESS CALCULATION USING FREQUENCY COUNT INFORMATION FROM TEST EXECUTIONS

Stmt. No.	Program	T1	T2	T3	T4	T5	T6	N_{CF}	N_{CS}	Susp. (Ochiai)	Rank
S1	void main(int argc,char *argv[])	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	12
S2	{ char strch[100];	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	13
S3	int alp, digit, ch, i;	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	14
S4	alp = digit = ch = i = 0;	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	8
S5	strcpy(strch,argv[1]);	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	9
S6	while(strch[i]!='\0')	0.92	0.95	0.92	0.95	0.98	0.88	2.82	2.80	0.69	1
S7	{ if((strch[i]>='a' && strch[i]<='z') || (strch[i]>='A' && strch[i]<='Z'))	0.88	0.92	0.88	0.92	0.97	0.82	2.71	2.69	0.67	2
S8	alp++;	0.73	0.82	0.73	0.73	0.82	0.73	2.28	2.28	0.62	5
S9	else if(strch[i]>'0' && strch[i]<='9') //correct strch[i]>='0'	0.73	0.73	0.73	0.82	0.88	0.62	2.32	2.19	0.63	4
S10	digit++;	0.00	0.73	0.00	0.73	0.73	0.00	1.46	0.73	0.57	7
S11	else	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	15
S12	ch++;	0.73	0.00	0.73	0.62	0.73	0.62	1.98	1.46	0.62	6
S13	i++;}	0.88	0.92	0.88	0.92	0.97	0.82	2.71	2.69	0.67	3
S14	printf("Alphabets =%d Digits=%d Special characters = %d", alp,digit,ch);	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	10
S15	}	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	11
	Result (Pass=P, Fail=F)	P	P	P	F	F	F				

D.Improving absolute suspiciousness rank of faulty statements using fault context information

Improving accuracy and effectiveness of lightweight software fault localization techniques (or SBFL techniques) is very important for debugging process. The fault localization methods can only be of practical use for developers if they have a certain level of accuracy. In order to be accurate these methods should identify the faults as early as possible while searching the descending order suspiciousness rank list.

According to the literature, SBFL methods in general perform well but in some cases SBFL methods are not able to locate faults early in the suspiciousness rank list, which means sometimes these methods are not so effective in locating faults [24]. In their study, Wang Y. et al. [25] introduced the notion of fault context to enhance the performance of traditional SBFL (or LFL) methods. The primary goal was to enhance the ranking (absolute) of faulty program entities in the rank list. This was achieved by utilizing a technique that calculates the suspiciousness of a program entity based on both its individual suspiciousness and that of its fault context. The suspiciousness of the program entity can be computed using any of the statistical lightweight software fault localization methods (examples of which are provided in Table I). Following this, the suspiciousness of the corresponding program entity's fault context is determined and then combined with the program entity's suspiciousness to arrive at the final suspiciousness score of the program entity.

The following is a definition of the fault context for a program entity. All program entities executed by a particular failed test case, excluding the entity itself, are included in the fault context of that program entity. In essence, the more suspicious a program entity is, the lower the suspiciousness of its fault context, resulting in a higher ranking for the entity in question.

We now present a working example as shown in Table VII that demonstrates the idea that how traditional spectrum-based fault localization techniques can yield superior results when combined with the fault context approach. We are using a popular SBFL technique Ochiai for the calculation of suspiciousness scores of program entities (i.e. statements) in this illustration.

The sample program finds the occurrences of vowels, consonants, digits and white spaces in its input. The program traverses through each character in the inputted string and determines the frequencies of the desired characters.

The conditional statement at statement number S10 contains a seeded fault because it is written incorrectly as *"if (line[i] = 'p')"*. The correct form of the conditional statement is *"if (line[i] <= 'z')"*.

As we can see in Table VII out of the total six test cases T1, T2, T3, T4, T5, and T6, three execute successfully and rest of the three have execution results as fail that means do not give the expected output. The bottom row shows the execution result as pass (P) or fail (F). The statement hit spectra is shown from columns three to eight. The entry with a '1' means the corresponding statement has been executed by the test case and a '0' means statement did not execute. In the following table, the N_{CF} and N_{CS} columns represent the number of failed and passed test cases, respectively, that cover a statement. These notations were previously defined in Section II-A. The suspiciousness score for each statement is displayed in the eleventh column, with the suspiciousness rank shown in descending order in the twelfth column. The rank indicates the likelihood of a statement being faulty. The statement with the highest suspiciousness score, indicating a greater probability of being faulty, is listed first in the rank list arranged in descending order. If multiple statements have the same suspiciousness score, they will share the same rank. This rank list serves as a debugging report for the faulty program and aids the developer in the debugging process. By examining each statement in the rank list in descending order, the developer can identify the faulty statement. If two statements have the same rank, the developer will examine them in a top-down manner. An effective fault localization technique should necessitate as few searches as possible from the developer in order to identify the faulty statement. Our objective is to enhance the fault localization performance by minimizing the developer's effort to identify and locate the faulty statement. The working example demonstrates how the fault context method can be used to improve fault localization performance.

By observing Table VII we can see that statement S16 has the highest suspiciousness score (0.82) and therefore has the highest rank (i.e. 1) but S16 is not the root fault. The root fault is actually statement S10, which has a suspiciousness score of 0.77 and a rank of 2. This scenario exemplifies the case where traditional SBFL techniques (like Ochiai) sometimes fail to give correct results. We now illustrate how fault context method improves the accuracy of SBFL techniques in such situations.

After examining the execution trace of failed test case T4, as shown in Table VII, we discover that the statement S10 is present in the trace. In this scenario, the fault context of S10 is {S1, S4, S5, S6, S7, S8, S9, S12, S15, S16, S17, S18}. The suspiciousness scores of all the statements in the fault context of S10 can be added together to determine the suspiciousness score of S10's fault context. Similarly, the fault context of S10 in failed executions of test cases T5 and T6 are {S1, S4, S5, S6, S7, S8, S9, S11, S12, S13, S14, S15, S16, S17, S18} and {S1, S4, S5, S6, S7, S8, S9, S17, S18}, respectively. Therefore, there exist three suspiciousness scores for S10's fault context.

As previously discussed in this section, a program entity's probability of being faulty increases with its higher suspiciousness rank and lower fault context suspiciousness. To determine the suspiciousness score of S10's fault context, the minimum suspiciousness score among the three scores is selected. The following formula is employed to compute the score of suspiciousness of S10's fault context.

$$\gamma_c(S10)=[(\text{ minimum }((\gamma_e(S1)+\gamma_e(S4)+\gamma_e(S5)+\gamma_e(S6)+\gamma_e(S7)+\gamma_e(S8)+\gamma_e(S9)+\gamma_e(S12)+\gamma_e(S15)+\gamma_e(S16)+\gamma_e(S17)+\gamma_e(S18)),(\gamma_e(S1)+\gamma_e(S4)+\gamma_e(S5)+\gamma_e(S6)+\gamma_e((S7)+\gamma_e(S8)+\gamma_e(S9)+\gamma_e(S11)+\gamma_e(S12)+\gamma_e(S13)+\gamma_e(S14)+\gamma_e(S15)+\gamma_e(S16)+\gamma_e(S17)+\gamma_e(S18)),(\gamma_e(S1)+\gamma_e(S4)+\gamma_e(S5)+\gamma_e(S6)+\gamma_e(S7)+\gamma_e(S8)+\gamma_e(S9)+\gamma_e(S17)+\gamma_e(S18))]$$

TABLE VII
SUSPICIOUSNESS CALCULATION USING TRADITIONAL OCHIAI METHOD (STEP-1 OF FAULT CONTEXT METHOD TO FAULT LOCALIZATION)

Stmt. No.	Program	T 1	T 2	T 3	T 4	T 5	T 6	N_{CF}	N_{CS}	Susp. (Ochiai)	Rank
S1	int main(int argc, char *argv[]) {	1	1	1	1	1	1	3	3	0.71	4
S2	char line[150];	0	0	0	0	0	0	0	0	0.00	17
S3	int vowels, consonant, digit, space;	0	0	0	0	0	0	0	0	0.00	17
S4	vowels = consonant = digit = space = 0;	1	1	1	1	1	1	3	3	0.71	4
S5	strcpy(line,argv[1]);	1	1	1	1	1	1	3	3	0.71	4
S6	for (int i = 0; line[i] != '\0'; ++i) {	1	1	1	1	1	1	3	3	0.71	4
S7	if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' ||line[i] == 'o' || line[i] == 'u')	1	1	1	1	1	1	3	3	0.71	4
S8	++ vowels;	1	0	1	1	1	1	3	2	0.77	2
S9	else if (line[i] >= 'a'){	1	1	1	1	1	1	3	3	0.71	4
S10	if(line[i] <= 'p') //correct if(line[i] <= 'z')	1	1	0	1	1	1	3	2	0.77	2
S11	++consonant;}	1	1	0	0	1	0	1	2	0.33	14
S12	else if (line[i] >= '0'){	1	1	1	1	1	0	2	3	0.52	13
S13	if(line[i] <= '9')	1	1	0	0	1	0	1	2	0.33	14
S14	++digit;}	1	1	0	0	1	0	1	2	0.33	14
S15	else if (line[i] == ' ')	0	1	1	1	1	0	2	2	0.58	12
S16	++space;}	0	0	0	1	1	0	2	0	0.82	1
S17	printf("Vowels: %d\nConsonants: %d\nDigits: %d\nWhite spaces: %d", vowels, consonant, digit, space);	1	1	1	1	1	1	3	3	0.71	4
S18	return 0;	1	1	1	1	1	1	3	3	0.71	4
S19	}	0	0	0	0	0	0	0	0	0.00	17
	Result (Pass=P, Fail=F)	P	P	P	F	F	F				

γ_c (S10) = minimum (8.34, 9.34, 6.43)
γ_c (S10) = 6.43

Where, γ_c(S10) denotes the suspiciousness score of the fault context of the statement S10, and γ_e (S1) represents the suspiciousness score of the statement S1.

Similarly, the statement S16 is in execution trace of program execution in unsuccessful test cases (i.e. failed) T4 and T5, and the fault contexts of S16 in these two executions are {S1, S4, S5, S6, S7, S8, S9, S10, S12, S15, S17, S18} and {S1, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S17, S18}, respectively. It is important to note that because S16 does not get executed in the execution coverage of failed test case T6, therefore, T6's execution coverage will not be considered in the computation of S16's fault context.

TABLE VIII
SUSPICIOUSNESS OF STATEMENTS, THEIR FAULT CONTEXTS AND IMPROVED SUSPICIOUSNESS RANK (STEP-2 & 3 OF FAULT CONTEXT METHOD TO FAULT LOCALIZATION)

Statement No.	Ochiai		Fault Context		Ochiai incorporating Fault Context	
	γ_e	R_e	γ_c	R_c	$R_e + R_c$	Rank (R)
S1	0.707	4	0.650	3	7	3
S2	0.000	17	10.000	17	34	17
S3	0.000	17	10.000	17	34	17
S4	0.707	4	0.650	3	7	3
S5	0.707	4	0.650	3	7	3
S6	0.707	4	0.650	3	7	3
S7	0.707	4	0.650	3	7	3
S8	0.775	2	0.643	1	3	1
S9	0.707	4	0.650	3	7	3
S10	0.775	2	0.643	1	3	1
S11	0.333	14	0.978	14	28	14
S12	0.516	13	0.860	13	26	13
S13	0.333	14	0.978	14	28	14
S14	0.333	14	0.978	14	28	14
S15	0.577	12	0.854	12	24	12
S16	0.816	1	0.830	11	12	11
S17	0.707	4	0.650	3	7	3
S18	0.707	4	0.650	3	7	3
S19	0.000	17	10.000	17	34	17

The following calculation can be used to determine the suspiciousness score of the fault context for S16.

γ_c (S16) = minimum [(γ_e (S1) + γ_e (S4) + γ_e (S5) + γ_e (S6) + γ_e (S7) + γ_e (S8) + γ_e (S9) + γ_e (S10) + γ_e (S12) + γ_e (S15) + γ_e (S17) + γ_e (S18)), (γ_e (S1) + γ_e (S4) + γ_e (S5) + γ_e (S6) + γ_e (S7) + γ_e (S8) + γ_e (S9) + γ_e (S10) + γ_e (S11) + γ_e (S12) + γ_e (S13) + γ_e (S14) + γ_e (S15) + γ_e (S17) + γ_e (S18))]
γ_c (S16) = minimum (8.30, 9.30)
γ_c (S16) = 8.30

The statement S16 is ranked higher than the root fault S10 in our example. The statement S16 was influenced by S10, and S10 is contained in its fault context. Therefore, it is possible that the fault context of S16 has a higher suspiciousness score than that of S10.

Table VIII lists the scores of suspiciousness of all program entities (in our case statements), the normalized suspiciousness scores of all statement's fault contexts, and the final overall rank of each statement based on the two suspiciousness ranks.

Steps to compute suspiciousness using fault context method

In this section, we formally explain how fault context information can be combined with spectrum-based fault localization to improve the absolute suspiciousness rank of faulty program entities.

Consider a faulty program P being debugged and a test suite T being represented as follows:

P = {e_1, e_2, e_3, …,e_n}, each e_i is a program entity (i.e. statement in this study). T= {t_1, t_2, t_3, …, t_m}, each t_j is a test case of test suite T. The test suite T is divided into passed and failed test cases, which are denoted as T_P and T_F, respectively. In order to collect the program spectra, program P is executed using input from both test cases T_P and T_F. Following this, the suspiciousness score of each program entity (using any of the SBFL similarity coefficient techniques), along with the suspiciousness score of its corresponding fault context, is calculated.

The final suspiciousness rank for each program entity is generated based on the ranks of the two suspiciousness scores. The resulting rank list reflects the likelihood of each program entity being faulty, where the most suspicious entity is assigned a rank of 1, followed by the next suspicious entity with a rank of 2, and so on. A programmer can examine each program entity (i.e., program statement) one by one according to its suspiciousness rank in order to locate the faulty statement.

The fault context approach to fault localization has the following major steps.

i. Suspiciousness computation for the program entities

In this step suspiciousness is computed using a spectrum-based similarity coefficient metric as defined in Table I. The program is executed with the given test suite T and run time program spectra is collected for the passing (T_P) and failing (T_F) test cases. The program spectra has two components, the coverage data for each program entity (i.e. whether a statement executed or not) and the result vector, which indicates whether the program passed or failed according to the given input. In this example we use Ochiai similarity coefficient metric as an SBFL tool to compute the suspiciousness of statements. The metric is defined below in (2) and the notations N_{CF}, N_{CS} and N_F are explained in Section II-A.

$$\gamma_e{}^{Ochiai}(j) = \frac{N_{CF}}{\sqrt{N_F \, x \, (N_{CF} + N_{CS})}} \quad (2)$$

Where $\gamma_e(j)$ stands for the suspiciousness score of a program entity j using Ochiai metric.

ii. Suspiciousness computation for fault contexts

This step involves providing a formal definition of the fault context. Let $ec_i = \{e_1, \ldots, e_j, \ldots, e_k\}$ denotes the set of covered entities for a failed test case execution t_i. For a given entity e_j, the fault context is defined as the collection of all statements that are included in the failed test execution t_i, excluding the entity e_j itself. The following expression given in (3) can be used to denote the fault context of e_j.

$$F_c(e_j, t_i) = ec_i \,/\, e_j \quad (3)$$

To determine the suspiciousness score of the fault context of statement e_j, we need to add up the suspiciousness scores of all statements ec_i covered during the execution of test case t_i, while excluding the statement e_j itself. Formula given in (4) provides the formal expression for calculating the suspiciousness score of the fault context of entity e_j.

$$\gamma_c(e_j, t_i) = \sum \gamma_e(F_c(e_j, t_i)(e_k)) \quad (4)$$

Where, $F_c(e_j, t_i)$ is the fault context of entity e_j in test execution t_i. The fault context for entity e_j in test execution t_i is defined as the collection of all entities that are covered by the failed test execution t_i, excluding the entity e_j itself. The notation e_k represents the k^{th} entity in this set. The value of k will range from 1 to n, where n is the total number of entities in the fault context. The function γ_e returns the suspiciousness score of the entity passed to it as per the SBFL metric defined in (2). Hence, the suspiciousness score of the fault context of entity e_j will be calculated by summing up suspiciousness scores of all entities present in the fault context of program entity e_j.

We must determine which fault context for the program entity e_j in T_F is the smallest, if e_j has multiple fault contexts. The definition of e_j's fault context and its suspiciousness are provided below in (5) and (6), respectively.

$$F_c(e_j) = \{ec_i / e_j \mid i \in \ failed\ test\ executions\ set\} \quad (5)$$

$$\gamma_c(e_j) = \text{minimum}\{\gamma_c(e_j, t_i) \mid i \in \text{failed test executions set}\} \quad (6)$$

iii. Generation of new fault ranking list

The first step computes the suspiciousness scores for program entities, and the second step calculates the suspiciousness score for each program entity's fault context.

In the third step, a new improved fault rank list is generated as per the following explanation. Firstly, two fault rank lists are created, R_e and R_c, where, R_e ranks the program entities in descending order based on their suspiciousness scores (i.e. γ_e), while R_c ranks the program entities in ascending order based on the suspiciousness scores of their fault contexts (i.e. γ_c). Finally, a new improved rank list, R, is created by combining the rank lists R_e and R_c as follows. Consider that e_i and e_j are two entities that are probably suspicious, with e_i having ranks R_e^i and R_c^i and e_j having ranks R_e^j and R_c^j. If $R_e^i + R_c^i <= R_e^j + R_c^j$, then e_i will be given a higher rank than e_j in the new fault rank list. Assuming e is a program entity, its rank in the newly generated ranking list will be determined by its suspiciousness score γ_e (e) and the suspiciousness score of its fault context γ_c (e), where a higher γ_e (e) and lower γ_c (e) will result in a higher overall suspiciousness rank for the entity e. It is important to emphasize that, a program entity is more likely to have a higher potential suspiciousness rank (root cause of fault) if its own suspiciousness score is greater and the suspiciousness score of its fault context is lower. Conversely, if the suspiciousness score of a program entity's fault context is higher, then lower will be the suspiciousness score of that entity, and therefore, that entity cannot be the root fault.

Table VII indicates that the conventional approach for fault localization requires three searches to identify the faulty statement S10. In contrast, the new fault context-based approach enables developers to locate the faulty statement by searching only two statements, as shown in Table VIII. Consequently, the developer's effort is reduced by 33.33% due to the improvement in the absolute rank of faulty statement. Here, we are measuring the effectiveness of fault context based approach in terms of improvement relative to the traditional SBFL metric Ochiai.

Before moving to the next section, we now summarize the background information given in Section-II. Section-II outlined three techniques aimed at enhancing the efficiency and efficacy of spectrum-based fault localization (SBFL). Test suites have a pivotal role in testing and debugging since they are responsible for driving program execution. In Section II-B, we see how optimized test suites can be selected or generated using Passing Tests Discrimination based method. The working example given in Section II-B illustrates that the use of optimized test suites improves the performance of existing SBFL methods. Section II-C explains how the concept of statement frequency is incorporated in suspiciousness formula instead of binary coverage information to improve the effectiveness of software fault localization. Section II-D elaborates on the notion of fault context, which can be employed to enhance

the absolute suspiciousness ranking of faulty statements (i.e. program entities) by comprehending the underlying cause of failure.

The next section presents our proposed hybrid approach (a fault localization framework for lightweight fault localization or SBFL) that combines the concepts of test suite optimization, statement execution frequency and fault context, to further improve the accuracy and performance of existing SBFL techniques in a single fault scenario. In this study, we employ the proposed approach to enhance the performance of existing classical SBFL techniques. It is important to emphasize that our approach can be applied to any SBFL technique to improve its performance.

III. Proposed Approach

A. Framework

This section presents a formal description of our proposed framework that integrates test suite optimization, statement execution frequency, and fault context to enhance the accuracy and performance of spectrum-based software fault localization in a single fault perspective. The framework, illustrated in Fig. 3, comprises several steps, which are presented in algorithmic form in Table IX.

B. Motivational Example

The following section provides a practical demonstration of the working of our proposed framework, showing how program entities (i.e., statements in this study) can be ranked efficiently based on their suspiciousness in descending order to identify the faulty statement. Fig. 3 depicts the process flow of our proposed framework/ approach. Through this example, we aim to demonstrate the effectiveness of our proposed approach in improving the performance (improved suspiciousness ranks of faulty program entities) of existing SBFL techniques. We illustrate that how our proposed approach, when applied to the classic Ochiai method, improves its fault localization performance.

For illustration purpose we consider the same program that has been used earlier in Section II-D, which finds the occurrences of vowels, consonants, digits and white spaces in its input. Throughout in this study we focus on fault localization in a single fault context, therefore our example uses the program that has a single seeded fault.

We first perform fault localization using the traditional Ochiai technique as shown in Table X. The statement S10 has an artificially seeded fault and its correct form is also given in the comments.

Table X presents the program's execution outcomes using a test suite that comprises three passing (T1, T2, and T3) and three failing test cases (T4, T5, and T6). The third through eighth columns show the execution result or statement coverage information. The N_{CF} and N_{CS} values are presented in columns nine and ten, respectively, where N_{CF} denotes the number of failing test cases that cover a statement and N_{CS} represents the number of passing test cases that cover a statement. In accordance with the definition of the Ochiai similarity coefficient-based metric, as described in subsection II-A, the suspiciousness value of each statement is calculated and displayed in the eleventh column. In the last column, each statement is ranked in decreasing order according to how suspicious it is. As can be observed, using the conventional Ochiai method, it would require eight searches to identify the faulty statement (i.e., S10).

In order to illustrate the performance of our proposed framework, we now perform fault localization on the same example program using the proposed framework. Table XI and Table XII show the step by step execution of different phases of the proposed fault localization approach.

As per the step-1 of the proposed framework, we use optimized test suite for the fault localization. As explained in Section II-B a test suite's effectiveness depends on its Passing Test Discrimination (PTD) measure. PTD is defined as ratio of total count of passing test cases not executing the faulty statement over the total number of passing test cases in a test suite.

TABLE IX
Major Steps (Algorithm) of the Proposed Lightweight Fault Localization Approach

(i) Step-1: Test Suite Optimization
This step creates or selects effective and reduced set of test cases (test suite) using Passing Test Discrimination-based method.

(ii) Step-2: Computation of statement execution frequency and normalization
Execution count of each statement is calculated with respect to each test case. A frequency weighting function (i.e.an adapted sigmoid function) is used to map the statement execution frequency into a normalized real value in the range of 0 and 1 [0, 1), as specified in (1).

(iii) Step-3: Improving absolute suspiciousness rank of faulty statements using fault context information
A new fault ranking list is produced in this stage, which considers both the suspiciousness score of a program entity and the suspiciousness score its fault context.

(a) Suspiciousness computation for program entities and their fault contexts

i. This step utilizes the program spectrum information obtained from the Step-2, to compute the suspiciousness of each statement using one of the SBFL similarity coefficient metric (e.g. Ochiai, Jaccard, DStar, etc.) as per the normalized frequency count instead of a binary coverage information (0 or 1).

ii. Fault contexts for each program entity is generated in each failed execution.

iii. The suspiciousness score of each program entity's fault context is then calculated according to the definition of suspiciousness of a fault context as explained in Section II-D. If a program entity (i.e., a program statement) has multiple fault contexts, then the minimum value among the different suspiciousness scores of the entity's fault contexts is selected.

(b) Upon completing step iii (a), we obtain the suspiciousness scores for each program entity and suspiciousness scores of their corresponding fault contexts. Utilizing these two suspiciousness scores, we generate two distinct lists of fault rankings, denoted as R_e and R_c. R_e ranks the program entities in descending order based on their suspiciousness scores (i.e. γ_e), while R_c ranks the program entities in ascending order based on the suspiciousness scores of their fault contexts (i.e. γ_c).

(iv) Step-4: Finally, a new improved rank list, R, is created by combining the rank lists R_e and R_c as follows. Consider that e_i and e_j are two entities that are probably suspicious, with e_i having ranks R_e^i and R_c^i and e_j having ranks R_e^j and R_c^j. If $R_e^i + R_c^i <= R_e^j + R_c^j$, then e_i is ranked higher than e_j in the new fault ranking list. A developer can then start examining the statements one by one as per the new fault rank list to locate the faulty statement.

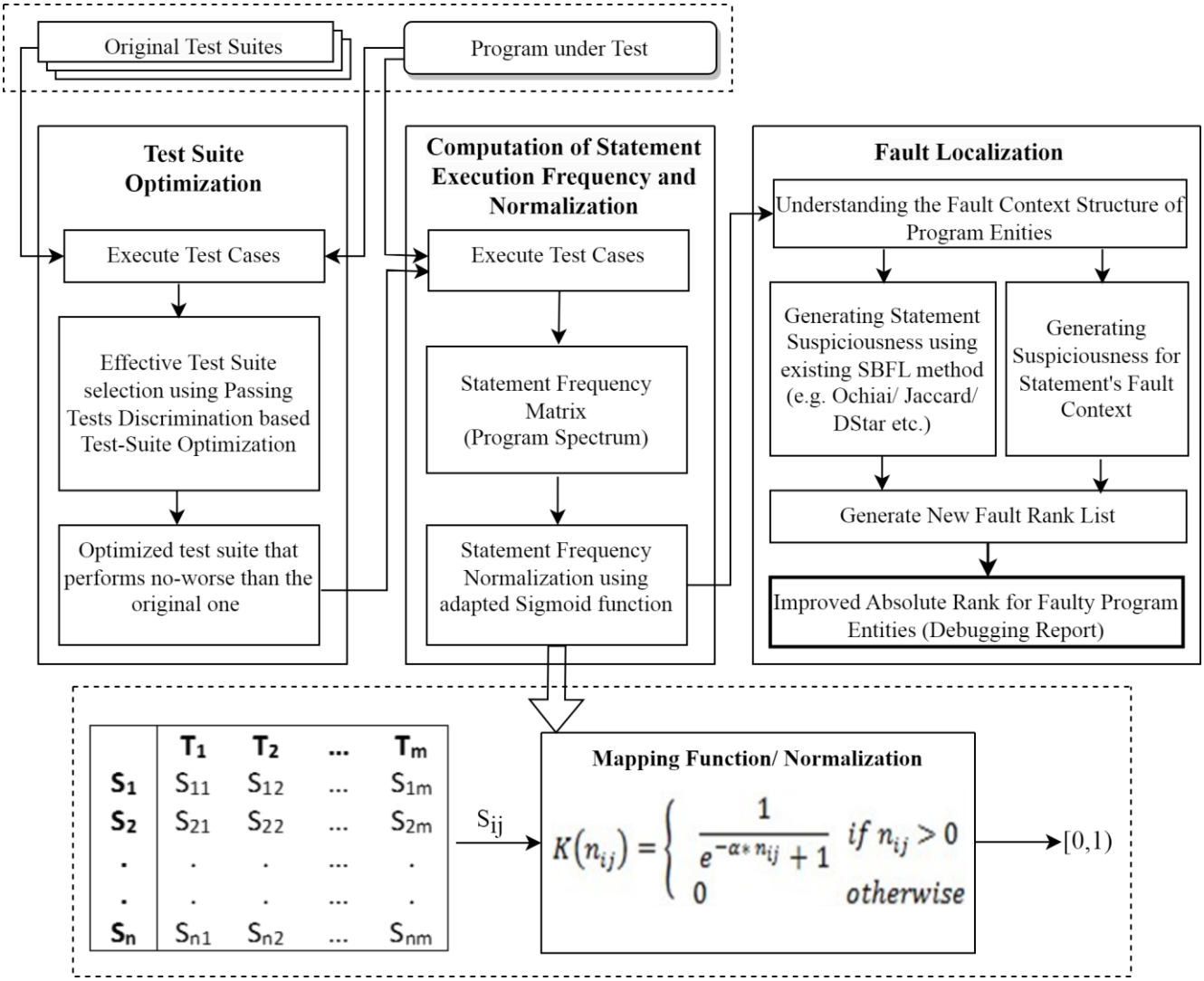


Fig. 3. Process flow of the proposed approach/ framework.

TABLE X
FAULT LOCALIZATION USING TRADITIONAL OCHIAI METHOD

Stmt. No.	Program	T 1	T 2	T 3	T 4	T 5	T 6	N_{CF}	N_{CS}	Suspiciousness (Ochiai)	Suspiciousness Rank
S1	int main(int argc, char *argv[]) {	1	1	1	1	1	1	3	3	0.71	2
S2	char line[150];	0	0	0	0	0	0	0	0	0.00	16
S3	int vowels, consonant, digit, space;	0	0	0	0	0	0	0	0	0.00	16
S4	vowels = consonant = digit = space = 0;	1	1	1	1	1	1	3	3	0.71	2
S5	strcpy(line,argv[1]);	1	1	1	1	1	1	3	3	0.71	2
S6	for (int i = 0; line[i] != '\0'; ++i) {	1	1	1	1	1	1	3	3	0.71	2
S7	if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' ||line[i] == 'o' || line[i] == 'u')	1	1	1	1	1	1	3	3	0.71	2
S8	++ vowels;	1	0	1	0	0	1	1	2	0.33	13
S9	else if (line[i] >= 'a'){	1	1	1	1	1	1	3	3	0.71	2
S10	if(line[i] <= 'p') //correct if(line[i] <= 'z')	1	1	1	1	1	1	3	3	0.71	2
S11	++consonant;}	1	1	1	0	0	0	0	3	0.00	16
S12	else if (line[i] >= '0'){	1	1	1	1	0	1	2	3	0.52	12
S13	if(line[i] <= '9')	1	1	0	1	0	0	1	2	0.33	13
S14	++digit;}	1	1	0	1	0	0	1	2	0.33	13
S15	else if (line[i] == ' ')	0	1	1	1	0	1	2	2	0.58	11
S16	++space;}	0	0	0	1	0	1	2	0	0.82	1
S17	printf("Vowels: %d\nConsonants: %d\nDigits: %d\nWhite spaces: %d", vowels, consonant, digit, space);	1	1	1	1	1	1	3	3	0.71	2
S18	return 0;	1	1	1	1	1	1	3	3	0.71	2
S19	}	0	0	0	0	0	0	0	0	0.00	16
	Result (Pass=P, Fail=F)	P	P	P	F	F	F				

TABLE XI
LIGHTWEIGHT SOFTWARE FAULT LOCALIZATION USING THE PROPOSED APPROACH (STEPS 1 & 2 OF THE PROPOSED APPROACH AS SHOWN IN TABLE IX)

Stmt. No.	Program	T1	T2	T3	T4	T5	T6	N_{CF}	N_{CS}	Susp. (Ochiai)	Susp. Rank
S1	int main(int argc, char *argv[]) {	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	11
S2	char line[150];	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0.00	17
S3	int vowels, consonant, digit, space;	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0.00	17
S4	vowels = consonant = digit = space = 0;	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	11
S5	strcpy(line,argv[1]);	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	11
S6	for (int i = 0; line[i] != '\0'; ++i) {	0.92	0.88	0.88	1.00	0.99	0.82	2.8	2.69	0.69	3
S7	if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' ||line[i] == 'o' || line[i] == 'u')	0.88	0.82	0.82	0.99	0.98	0.73	2.71	2.52	0.68	5
S8	++ vowels;	0.73	0.73	0.62	0.73	0.62	0.62	1.98	2.08	0.57	10
S9	else if (line[i] >= 'a'){	0.73	0.62	0.73	0.98	0.97	0.62	2.58	2.08	0.69	4
S10	if(line[i] <= 'p') //correct if(line[i] <= 'z')	0.62	0.62	0.00	0.88	0.73	0.62	2.23	1.24	0.69	2
S11	++consonant;}	0.62	0.62	0.00	0.62	0.62	0.00	1.24	1.24	0.46	16
S12	else if (line[i] >= '0'){	0.62	0.00	0.73	0.88	0.92	0.00	1.8	1.35	0.59	8
S13	if(line[i] <= '9')	0.62	0.00	0.00	0.73	0.82	0.00	1.55	0.62	0.61	6
S14	++digit;}	0.62	0.00	0.00	0.73	0.82	0.00	1.55	0.62	0.61	6
S15	else if (line[i] == ' ')	0.00	0.00	0.73	0.73	0.73	0.00	1.46	0.73	0.57	9
S16	++space;}	0.00	0.00	0.00	0.73	0.73	0.00	1.46	0	0.70	1
S17	printf("Vowels: %d\nConsonants: %d\nDigits: %d\nWhite spaces: %d", vowels, consonant, digit, space);	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	11
S18	return 0;	0.62	0.62	0.62	0.62	0.62	0.62	1.87	1.87	0.56	11
S19	}	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0.00	17
	Result (Pass=P, Fail=F)	P	P	P	F	F	F				

In Table X, the program being debugged is executed by a random test suite, which has PTD score of 0, as all passing test cases execute the faulty statement S10. Whereas, the test suite used in Table XI is an optimized test suite with a PTD score of 33.33% (i.e. 1/3). We can see that passing test case T3 does not execute the faulty statement S10.

As per step-2 of our proposed framework, we refrain from using binary information (0 or 1) to indicate whether a program statement is executed by a test case or not. Instead, we use statement coverage count (execution frequency) for each statement covered by a test case, as depicted in Table XI. We further normalize the statement execution frequencies within the [0, 1) range by employing the sigmoid function (with α =0.5) as described in (1) of Section II-C. As illustrated in Table XI, the test suite comprises six test cases, with three passing and the remaining three failing.

Now, we compute the suspiciousness score of each program statement according to the step-3 of the proposed framework (see Table IX). First, we compute the suspiciousness score of each program statement using Ochiai SBFL technique as defined in Table I of Section II-A. The suspiciousness score of each statement is given in column number eleven of Table XI. Then fault context of each statement is generated in each failed execution. By observing Table XI we can see that statement S16 has the highest suspiciousness score (0.70) and therefore has the highest rank (i.e. 1), but S16 is not the root fault. The root fault is actually statement S10, which has a suspiciousness score of 0.69 and has a rank of 2. In this scenario, even if we have used a statement execution frequency count instead of binary information to represent the statement coverage information, the faulty statement is still not ranked at the highest position in the rank list.

We now demonstrate how fault context method further improves the absolute suspiciousness rank of a faulty statement in such situations.

By observing execution trace of failed test case T4 as shown in Table XI, we find that the statement S10 is in the execution trace of T4, and in this situation the fault context of S10 is {S1, S4, S5, S6, S7, S8, S9, S11, S12, S13, S14, S15, S16, S17, S18}. The suspiciousness score of S10's fault context can be expressed as the sum of the suspiciousness scores of all statements in the S10's fault context i.e. {S1, S4, S5, S6, S7, S8, S9, S11, S12, S13, S14, S15, S16, S17, S18}. Similarly, the fault contexts of S10 in failed executions of test cases T5 and T6 are {S1, S4, S5, S6, S7, S8, S9, S11, S12, S13, S14, S15, S16, S17, S18} and {S1, S4, S5, S6, S7, S8, S9, S17, S18}, respectively. We can note that the fault context in test executions T4 and T5 are same. We can define the suspiciousness score for S10's fault context as the total of suspiciousness scores of every statement present within S10's fault context. Therefore, we can say that for statement S10, there exist three suspiciousness scores in its fault context for each failed test case T4, T5 and T6 where S10 executes. As already explained in this section, a program entity is likely to be faulty if its rank, which is based on the descending order of its suspiciousness, is higher and its fault context's suspiciousness is lower. Hence, we choose the least among the three suspiciousness scores associated with the fault context of S10. The below given formula finds the suspiciousness score of S10's fault context.

γ_c (S10) = minimum [(γ_e (S1) + γ_e (S4) + γ_e (S5) + γ_e (S6) + γ_e (S7) + γ_e (S8) + γ_e (S9) + γ_e (S11) + γ_e (S12) + γ_e (S13) + γ_e (S14) + γ_e (S15) + γ_e (S16) + γ_e (S17) + γ_e (S18)), (γ_e (S1) + γ_e (S4) + γ_e (S5) + γ_e (S6) + γ_e (S7) + γ_e (S8) + γ_e (S9) + γ_e (S11) + γ_e (S12) + γ_e (S13) + γ_e (S14) + γ_e (S15) + γ_e (S16) + γ_e (S17) + γ_e (S18)), (γ_e (S1) + γ_e (S4) + γ_e (S5) + γ_e (S6) + γ_e ((S7) + γ_e (S8) + γ_e (S9) + γ_e (S17) + γ_e (S18))]

γ_c (S10) = minimum (8.94, 8.94, 5.42)

γ_c (S10) = 5.42

Where, γ_c(S10) denotes the suspiciousness score of the fault context of the statement S10 and γ_e (S1) represents the suspiciousness score of the statement S1 (computed using Ochiai method). Here, it should be noted that the suspiciousness scores of S10's fault context are same in failed executions of test cases T4 and T5 (i.e. 8.94).

Likewise, statement S16 is present in the execution trace of failed test cases T4 and T5, both having the same fault context {S1, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S17, S18}. However, since S16 is not covered in the execution of failed test case T6, its execution trace is excluded when calculating the fault context of S16. We can compute the suspiciousness score of S16's fault context using the following formula.

γ_c (S16) = minimum [(γ_e (S1) + γ_e (S4) + γ_e (S5) + γ_e (S6) + γ_e (S7) + γ_e (S8) + γ_e (S9) + γ_e (S10) + γ_e (S11) + γ_e (S12) + γ_e (S13) + γ_e (S14) + γ_e (S15) + γ_e (S17) + γ_e (S18), (γ_e (S1) + γ_e (S4) + γ_e (S5) + γ_e (S6) + γ_e (S7) + γ_e (S8) + γ_e (S9) + γ_e (S10) + γ_e (S11) + γ_e (S12) + γ_e (S13) + γ_e (S14) + γ_e (S15) + γ_e (S17) + γ_e (S18))]

γ_c (S16) = minimum (8.94, 8.94)

γ_c (S16) = 8.94

The statement S16 is ranked higher than the root fault S10 in our example. The statement S16 was influenced by S10, and S10 is contained in its fault context. Therefore, the score of suspiciousness of S16's fault context may be higher than that of S10's.

Table XII presents a summary of the suspiciousness scores of all program statements, the normalized suspiciousness scores of all statements' fault contexts, and the final overall rank of each statement based on the two suspiciousness ranks. We have normalized the suspiciousness scores of all statement's fault contexts as shown in column four (γ_c) of Table XII.

The final ranks of statements' suspiciousness is shown in column 7 of Table XII. We can see that the faulty statement (S10) can now be directly identified as it has the top rank of 1. If we compare our proposed approach with traditional SBFL method (Ochiai, Jaccard and Dstar in this study), we find that the traditional approach (see Table X) took eight searches to locate the faulty statement whereas the proposed approach directly identified the faulty statement in one search. Therefore, in this example, applying our proposed approach to the traditional SBFL method, Ochiai, resulted in an 87.5% improvement in fault identification accuracy.

TABLE XII
SUSPICIOUSNESS OF STATEMENTS, THEIR FAULT CONTEXTS AND IMPROVED SUSPICIOUSNESS RANK (STEP 3 & 4 OF THE PROPOSED APPROACH AS SHOWN IN TABLE IX)

Stmt. No.	Ochiai		Fault Context		Ochiai Incorporating Fault Context	
	γ_e	R_e	γ_c	R_c	$R_e + R_c$	Rank (R)
S1	0.56	11	0.56	6	17	7
S2	0.00	17	10.00	17	34	17
S3	0.00	17	10.00	17	34	17
S4	0.56	11	0.56	6	17	7
S5	0.56	11	0.56	6	17	7
S6	0.69	3	0.54	2	5	2
S7	0.68	5	0.54	4	9	4
S8	0.57	10	0.55	5	15	6
S9	0.69	4	0.54	3	7	3
S10	0.69	2	0.54	1	3	1
S11	0.46	16	0.92	16	32	16
S12	0.59	8	0.90	14	22	14
S13	0.61	6	0.90	12	18	12
S14	0.61	6	0.90	12	18	12
S15	0.57	9	0.91	15	24	15
S16	0.70	1	0.89	11	12	5
S17	0.56	11	0.56	6	17	7
S18	0.56	11	0.56	6	17	7
S19	0.00	17	10.00	17	34	17

IV. EMPIRICAL STUDY

This section highlights the research questions of the study, programs used as subjects for the experiments, the process of data collection, evaluation metrics, evaluation criteria for the experimental process, and results and analysis.

The objective of this study is to improve the performance and efficacy of spectrum-based (or lightweight) fault localization techniques in a single-fault scenario. To substantiate this claim, we conducted a thorough experimental study using the standard Siemens benchmark suite and some large real-world subject programs taken from the software-artifact infrastructure repository (SIR) (http://sir.unl.edu/portal/index.php) [26]. We perform an empirical evaluation to investigate how the proposed approach enhances the performance of existing SBFL methods. In this research study, we apply the proposed approach to three existing SBFL methods, namely Ochiai, Jaccard, and DStar, and analyze the resulting performance improvements.

A. Research Questions

The research questions that we aim to answer through our empirical study are as follows:

RQ1: Can test suites be optimized to improve the performance of existing spectrum-based fault localization techniques?

RQ2: How does the concept of statement execution frequency information improve fault localization performance when it is incorporated into the suspiciousness formula of SBFL techniques instead of binary information of execution count?

RQ3: Is it possible to improve the absolute suspiciousness ranking of faulty program entities by understanding the root cause of failure using fault context?

RQ4: Is the proposed framework/approach, which combines the concepts of statement execution frequency, test suite optimization, and fault context, effective in further enhancing the performance of spectrum-based software fault localization in a single fault context?

The concepts related to first three research questions have already been explained in Section II that presents the background and motivation of our proposed approach. However, in the following paragraphs, we provide explicit answers to all four research questions by referring to each related concepts. The answers to the first and the fourth research questions are given in greater detail in Results and Analysis section (i.e. Section IV-C).

We have evaluated our proposed approach on subject programs as listed in Table XIII. We have downloaded all subject programs along with standard test suites from the software-artifact infrastructure repository (SIR) (http://sir.unl.edu/portal/index.php) [26]. For our empirical study, we have used a total of 40 different faulty versions,

out of which 25 faulty versions we have taken from the Siemens test suites, and 15 faulty versions we have taken from large real-world programs (see Table XIII).

In response to the first research question (*RQ1*), we optimized the test suites using Passing Test Discrimination (PTD) method, which has been explained in detail in the subsection II-B. The motivational example shown in Table II uses a random test suite (testsuite-1) for testing, and in this case, it requires eight searches to locate the faulty statement. Whereas, the example shown in Table III uses an optimized test suite (testsuite-2), which requires only one search to locate the fault.

The primary aim of the PTD metric is to enhance the efficiency of a test suite by preventing the execution of faulty statements by passing test cases. Such execution can adversely impact the accuracy of localization of faults. Therefore, a test suite with a high PTD value is deemed optimal, indicating that a greater proportion of passing test cases do not execute faulty statements.

When comparing the PTD scores of the two test suites in the motivational examples provided in Tables II and III of Section II-B, we observe that the optimized test suite (testsuite-2) achieves a PTD score of 66.66%. This is because two out of three passing test cases do not execute the faulty statement (i.e., 2/3). On the other hand, the original test suite (testsuite-1) has a PTD score of 0% since all test cases that produce the expected output (i.e., passing) execute the faulty statement S9. As a result, testsuite-2 is considered to be an optimized test suite that provides improved fault localization results.

In order to optimize an existing test suite, we eliminate those passing test cases that are likely to execute the faulty statement. The minimum suspicious set (MSS) notion is utilized to recognize the potential test cases for removal from the original test suite. To increase a test suite's PTD score, we can add new passing test cases whose execution covers a smaller percentage of statements in MSS, reducing the possibility of executing the faulty statement. In this way we select or create the optimized test suites. We have provided a detailed explanation of test suite optimization in Section II-B. In the proposed fault localization approach we use the concept of optimized test suites as shown in Fig. 3.

With regard to the second research question (*RQ2*), we use the concept of statement execution frequency information (instead of binary information) in suspiciousness calculation formula of spectrum-based fault localization. As explained in Section II-C, existing SBFL techniques have limited diagnostic capabilities particularly when faults occur in loop bodies or iteration statements. There is one more limitation with SBFL techniques that statements with same suspiciousness scores result in ties in the ranking [1]. These limitations can be addressed if we incorporate statement execution frequency information instead of binary coverage information (0 or 1) in the suspiciousness calculation formula of existing SBFL techniques. The motivational examples given in Section II-C (Table IV, Table V, and Table VI) clearly show that, when we incorporate statement execution frequency information in the suspiciousness formula, the fault localization performance improves by 50% (from rank 8 to rank 4). Therefore, our proposed approach solves the inherent problems that exist with SBFL as discussed above, by incorporating the concept of statement execution frequency information in the suspiciousness calculation formula of existing SBFL techniques as shown in Fig. 3.

The following paragraph answers the third research question (*RQ3*). As outlined in Section II-D, SBFL techniques generally perform well, but there are instances where they are unable to identify faults early in the suspiciousness rank list. To address this limitation and further enhance the absolute rank of faulty program entities, we incorporate the concept of fault context into existing SBFL techniques. This approach involves computing the suspiciousness of a program entity by combining its own suspiciousness score with that of its fault context, as explained in Section II-D. The practical example provided in the section demonstrates how utilizing fault context can lead to further improvement in the performance of existing SBFL techniques. In Section II-D, we provided a motivational example that demonstrates how the fault context-based method can enhance the accuracy of fault localization compared to traditional SBFL techniques such as Ochiai, Jaccard etc. Specifically, as explained in the example given in Section II-D, the fault context-based method improved the fault localization accuracy by 33.33%. To enhance the absolute rank of program entities responsible for the failure of the program, our proposed framework (depicted in Fig. 3) make use of the concept of fault context.

In response to the fourth research question (*RQ4*), we propose a hybrid framework that effectively enhances the performance of existing SBFL techniques in a single fault perspective. We have conducted rigorous experimentation to measure the efficiency of our proposed approach. We have evaluated the proposed approach using four different measures, namely, Exam score, Cumulative Number of Statements Examined, Top-N, and Wilcoxon signed-rank test. Section IV-C contains a detailed presentation of the experimental results. Therefore, by analyzing these results, we are able to answer the fourth research question.

B. Experimental Setup

In order to apply our proposed approach, we have created a prototype tool. The suspiciousness score and other results are computed by this automated tool, which we have developed in Python 3.7.3, for use in our experimentation.

i. Subject Programs

To evaluate our proposed approach, we have performed extensive experimentation work in which we have utilized the popular seven Siemens test suite subject programs and four large real-world programs, as described in Table XIII. Out of four large real-world programs, three are UNIX utilities, namely flex, sed, grep, and the fourth one is ‘space’ program, which is an interpreter for array definition language (ADL). All subject programs were obtained from the software-artifact infrastructure repository (SIR) (http://sir.unl.edu/portal/index.php) [26]. As per the software engineering literature, these subject programs have been widely utilized in fault localization experimentation [12], [14], [27]. In contrast to UNIX utilities, which contain both real-world and seeded errors, Siemens applications contain only a single seeded fault. In our experimentation study, we have utilized only single fault versions of all the subject programs.

ii. Data Collection Process

In our experimentation work, various faulty versions of subject programs are used, as shown in Table XIII. With all of the available test inputs, we executed each faulty version. For the purpose of determining whether the outcome was successful or unsuccessful, we then compared the test execution output of each faulty version to its corresponding fault-free original version. The test input is marked as failing if the output of a version that has a fault differs from the output of its fault-free counterpart. But, if the output of a faulty version is identical to that of its faulty-free counterpart, the test input is deemed successful.

On a Windows 10 computer with an Intel® Core i5 processor running at 2.7 GHz and 8 GB of RAM, a Linux environment was installed and used for all experiments. The programs were compiled using GCC 11.3.0, and the code coverage data for each test execution was collected using GCOV 11.3.0 [https://gcc.gnu.org/onlinedocs/gcc/Gcov.html].

TABLE XIII
SUBJECT PROGRAMS USED IN EMPIRICAL RESEARCH

Program	Line of Code	Faulty versions used in the Experimentation	Brief Description
Siemens programs			
print_tokens	565	v5, v7	Lexical analyzer
print_tokens2	510	v4, v5, v6, v7	Lexical analyzer
tcas	173	v1, v2, v3, v4, v5	Altitude separation
tot_info	406	v2, v4, v5, v7	Information measure
replace	562	v1, v3, v4, v6, v7	Pattern recognition
schedule	412	v2, v3, v4	Priority scheduler
schedule2	307	v6, v7	Priority scheduler
Large real-world programs			
flex	13,892	v1, v2, v3, v4	Lexical analyzer generator
sed	12,062	v2, v3	Textual manipulator
grep	12,653	v1, v2	Pattern searcher
space	9,126	v5, v14, v15, v18, v20, v21, v23	ADL interpreter

iii. Evaluation metrics and criteria

In order to evaluate the effectiveness of a specific fault localization technique, it is necessary to utilize appropriate metrics. Within this section, we will examine the various evaluation metrics employed to measure the efficacy of our proposed approach in comparison to traditional SBFL methods. This study employs four metrics to evaluate the proposed approach, specifically the Exam score, Cumulative Number of Statements Examined, Top-N, and Wilcoxon Signed-Rank Test, in order to assess its performance against the classic SBFL methods.

a) Exam Score

Choosing an appropriate metric is essential when evaluating the efficacy of a single fault localization technique. Among the most frequently used metrics in the fault localization literature [11], [12], [27], [29], is the Exam Score. This metric is a subset of the original Score metric [12], which represents the proportion of code that does not require inspection to locate a fault. In contrast, the Exam Score is a more straightforward metric that refers to the percentage of code that needs to be inspected to identify the initial faulty statement in a program being debugged. The EXAM score metric has two variants: the relative and the absolute variant. The relative variant takes into account the size of the program being debugged, i.e., the total number of statements. In contrast, the absolute variant, which is also a form of the EXAM score metric, calculates the number of statements that need to be inspected to detect the first faulty instruction. The formula for the EXAM score metric is presented below in (7).

$$\text{Exam Score} = \frac{\text{Rank of Fault}}{\text{Overall Statement Count of the Program}} \times 100\% \quad (7)$$

A higher level of performance of the fault localization method is indicated by a lower Exam Score because it necessitates the inspection of a smaller fraction of the code to identify the faulty statements. In this study, we utilize the Exam Score metric to compare the proposed approach with traditional SBFL methods. The objective is to evaluate the effectiveness of the proposed approach compared to the existing SBFL techniques. The proposed approach enhances the absolute suspiciousness ranking of the faulty program entities in the fault rank list. The improvement in the proposed approach can be defined using the formula given in (8).

$$\text{Improvement}\,(A, \mathrm{B}) = \frac{A-B}{A} \times 100\% \quad (8)$$

In the above formula, A and B represent the definitive rank (i.e. absolute rank) produced by the traditional SBFL method and the proposed approach, respectively.

b) Cumulative Number of Statements Examined

The cumulative number of statements examined (CSE) is a valuable evaluation tool used to estimate the effectiveness of a fault localization technique. It reflects the total number of program statements scrutinized by the technique in question during the fault localization process. The basic idea behind CSE is that a good fault localization technique should examine as few statements as possible to locate the fault accurately. Therefore, the lower the CSE value, the more efficient the technique is considered to be.

CSE is a cumulative metric, which means that it counts all the statements examined by the technique up to a certain point in the debugging process, and not just the statements examined in a single iteration. This allows for a fair comparison of different fault localization techniques, even if they use different strategies for examining program statements.

As the fault localization technique is applied, the number of program statements examined is counted. A program statement is considered examined if it is executed or evaluated in some way during the fault localization process. The cumulative number of statements examined is calculated as the sum of the statements examined up to the point where the fault was successfully localized across all faulty versions considered in the experimentation. This value is used as a measure of the efficiency of the fault localization technique.

Suppose there is a program with N faulty versions, and two fault localization techniques, S and T. Let S(i) and T(i) represent the number of statements that need to be inspected by techniques S and T, respectively, to locate all the faults in the i^{th} faulty version. If it is observed that technique S requires fewer statements to be examined than technique T, as depicted in (9), then it can be concluded that technique S is more effective than technique T in identifying all the faults in the faulty versions.

$$\sum_{i=1}^{N} S(i) < \sum_{i=1}^{N} T(i) \quad (9)$$

c) Top-N

The metric known as Top-N refers to the percentage of faults detected by a fault localization technique within the top-n positions of the ranked list [28]. Here, N in Top-N denotes the position like N =1, 5, 10. The smaller value of N is considered to be more stringent. For example, N=5 requires that all faults (in different versions of a subject program) should be ranked in top 5 positions in the ranked list. The fault localization literature frequently makes use of the Top-N metric. In our study, we use the top-N metric to compare the performance of the conventional SBFL techniques and our proposed approach.

d) Wilcoxon Signed-Rank Test

The Wilcoxon signed-rank test is a statistical technique that is frequently employed to compare two paired samples and evaluate the difference between them. It is a non-parametric test, which implies that it does not assume any particular distribution of the data. This makes it a useful alternative to parametric tests, like the paired t-test, when normality assumptions are not met. The test works by comparing the ranks of the differences between paired observations, rather than the actual values themselves. This approach makes it particularly useful when dealing with data that is not normally distributed or contains outliers [14].

The Wilcoxon signed-rank test can be utilized in fault localization to compare the efficacy of various techniques used to locate faults on a single set of faults. By comparing the performance of various fault localization techniques on the same set of faults, their effectiveness can be assessed. To do this, the techniques are applied to the set of faults, and the number of statements that each technique examines to locate the first faulty statement is recorded. These numbers can then be used to rank the performance of the techniques.

The Wilcoxon signed-rank test can then be applied to determine whether any differences in the rankings are statistically significant or just due to chance. This allows for a more robust evaluation of the effectiveness of different fault localization techniques and can help identify which techniques are most effective for a given software system.

In this research work, we conducted experiments to demonstrate that our proposed approach significantly improves the performance of existing SBFL techniques. We began by calculating the total number of statements that a programmer would need to analyze to locate the first faulty statement, and found that our proposed approach was more efficient than Ochiai and other conventional SBFL approaches. We then evaluated the two-sided alternative hypothesis that the existing SBFL methods like Ochiai (or Jaccard, DStar etc.) must examine an equal or greater number of statements than our proposed approach to achieve similar results.

The null hypothesis used in this study is as follows. H_0: The existing SBFL technique requires examining the same or fewer statements than the proposed approach.

In the event that the null hypothesis is rejected, the alternative hypothesis will be accepted. According to the alternative hypothesis, the proposed approach necessitates examining fewer statements than the compared classical SBFL technique, indicating its superior efficiency.

C. Results and Analysis

This section presents a comprehensive analysis of the results of the experimental study to validate our proposed approach/ framework. We use four evaluation metrics namely EXAM score, cumulative number of statements examined (CSE), Top-N, and Wilcoxon Signed-Rank Test, as explained above in Section IV-B (iii), to evaluate the performance and efficiency of the proposed approach as against the classic SBFL techniques.

i. Improving fault localization performance using test suite optimization

In response to the first research question (*RQ1*), this section further illustrates that as per the proposed approach, how we have used the concept of test suite optimization (see Section II-B for details) to improve the effectiveness of locating faults within a program in our experimentation work. In Section II-B, it is explained how the PTD score of a test suite can influence the accuracy of identifying faults in a program in both positive and negative ways. To be precise, a test suite with a higher PTD score can enhance the efficiency of fault localization by refining the placement of faulty program units (e.g., statements) in the suspicious ranking list. The said ranking list is arranged in descending order based on the suspiciousness score of program entities, and the enhancement signifies that the faulty statement is more likely to be positioned at a higher rank in the list, thus facilitating its localization.

To increase the PTD score of a test suite T, we utilize a heuristic discussed in Section II-B. This heuristic involves eliminating from T those passing test cases, denoted as t, which exhibit a greater likelihood of executing a faulty statement. The heuristic is explained as follows. Here, S is the set of statements executed by t, MSS is the minimum suspiciousness set and α is the threshold value.

$|S \cap MSS|/|MSS| > \alpha$

The test case t can be removed from the test suite T, if α is greater than a certain threshold. There is no standard value for α in practice as it is a matter of investigation on different experimental settings. It is understood that when passing test cases execute a significant number of statements within the MSS, there is a higher likelihood that they will execute the faulty statement, resulting in poorer accuracy in identifying faults during the debugging process of a program. Conversely, if the passing test cases execute only a small number of statements within the MSS, the test suite is considered optimized, and this approach can lead to improvement in the accuracy of identifying faults in a program.

As previously stated, our experiments involved six test cases, three of which are passing test cases while the other three are failing that means do not give the expected output. The passing test cases executed specific percentage of statements within the MSS, and this information is presented in Table XIV for every subject program being utilized in our experiments. We can observe that certain passing test cases cover only a small number of statements in the MSS, making it less likely for those specific passing test cases to execute the statement that is a faulty one. In this way we optimize test suites for our proposed fault localization process (see Section II-B for details). Table XIV shows the

portion of MSS covered by the execution of each passing test case (T1, T2 & T3) in percentage. We can note that the passing test cases (T1, T2 & T3) of the optimized test suite for version V5 of 'print_tokens' subject program are

TABLE XIV
PASSING TEST CASES (T1, T2 & T3) OF OPTIMIZED TEST SUITES COVERING A SPECIFIC PERCENTAGE OF STATEMENTS IN MSS

Subject Program	Version	T1	T2	T3
Siemens Programs				
print_tokens	V5	55.09	53.89	86.83
print_tokens	V7	56.59	87.60	62.02
print_tokens2	V4	46.49	76.76	47.03
print_tokens2	V5	67.40	60.77	77.35
print_tokens2	V6	48.86	67.05	86.36
print_tokens2	V7	69.82	50.89	66.86
replace	V1	41.26	87.38	41.26
replace	V3	75.61	33.17	92.68
replace	V4	61.17	30.10	82.52
replace	V6	40.19	35.41	29.67
replace	V7	50.00	71.79	71.79
schedule	V2	29.80	29.80	82.78
schedule	V3	86.75	86.09	57.62
schedule	V4	67.55	86.09	85.43
schedule2	V6	43.61	80.45	69.92
schedule2	V7	43.38	79.41	60.29
tcas	V1	96.23	86.79	56.60
tcas	V2	56.60	56.60	96.23
tcas	V3	3.92	3.92	86.27
tcas	V4	56.60	94.34	94.34
tcas	V5	3.51	52.63	52.63
tot_info	V2	73.08	47.12	47.12
tot_info	V4	76.11	65.49	38.94
tot_info	V5	55.56	84.26	34.26
tot_info	V7	77.19	40.35	42.98
Large real-world programs				
flex	V1	62.22	47.40	43.32
flex	V2	58.27	44.45	41.78
flex	V3	56.12	42.68	36.47
flex	V4	59.46	40.77	38.92
grep	V1	68.90	63.89	64.23
grep	V2	70.85	80.15	77.30
sed	V2	95.09	94.57	87.17
sed	V3	93.94	62.66	39.03
space	V14	45.86	38.73	70.63
space	V15	68.21	66.58	40.72
space	V18	81.69	76.42	48.39
space	V20	76.24	58.71	78.25
space	V21	80.60	38.36	76.41
space	V23	76.67	28.46	29.00
space	V5	73.61	73.61	81.22

covering 55.09%, 53.89% and 86.83% of statements in MSS, respectively. It is worth mentioning that in this scenario, T1 and T2 provide coverage to a limited portion of the MSS. Consequently, the probability of these passing test cases executing the faulty statement, as per the heuristic explained earlier, is relatively low. Similarly, the version V14 of 'space' subject program is covering 45.86%, 38.73% and 70.63% of statements in MSS, respectively. Now, suppose that in this case (V14 of 'space' program) passing test cases T1 and T2 do not execute the faulty statement as they are covering very less portion of the MSS in the test suite. These test cases exhibit coverage over a minimal section of the MSS. As a result, the PTD (Passing Test Discrimination) score for the test suite computes to 66.66%, specifically denoting that two out of three passing test cases do not execute the faulty statement.

ii. Performance evaluation of the proposed approach based on EXAM Score

In this subsection, we apply the proposed approach to three existing SBFL methods, namely Ochiai, Jaccard, and DStar, and evaluate the resulting performance improvements with respect to the EXAM score metric. The primary objective of our proposed approach is to enhance the performance of existing SBFL techniques. When we apply the proposed approach to the existing SBFL techniques, we denote them by appending an asterisk symbol ('*') to their respective original names. For instance, our proposed approach for the baseline Ochiai method is denoted as Ochiai*, and similarly, we use Jaccard* and DStar* denote the proposed approaches for the baseline methods Jaccard and DStar, respectively. In other words, our improved version of the baseline Ochiai method is denoted as Ochiai*, while Jaccard* and DStar* represent the improved versions of the baseline methods Jaccard and DStar, respectively.

Table XV and Table XVI show a comparative analysis of the effectiveness between the proposed approach (Ochiai*) and the classic Ochiai method on Siemens programs and large real-world subject programs, respectively. The subject program, faulty version, and line of code (LOC) are displayed in the first three columns of Table XV and Table XVI, respectively. The performance of baseline method (classic Ochiai) in absolute and relative variants of EXAM score metric is shown in column 4 and 5, respectively. The absolute version of the EXAM score metric indicates the cost of finding the first fault by counting the number of program entities (in our study, statements) examined. On the other hand, the relative version of the metric takes into consideration the program's size and expresses the required effort as a percentage of the code examined to locate the fault.

The line of code need to be examined (developer's effort) using traditional Ochiai method in order to locate fault in terms of absolute measure is shown in column 4, and developer's effort in terms of relative measure of EXAM score is shown in column 5. Similarly, the performance of the proposed approach (Ochiai*) in absolute and relative variants of EXAM score metric is shown in column 6 and 7, respectively. Columns 8 and 9 of Table XV and Table XVI illustrate how the traditional Ochiai method and the proposed approach differ in terms of the absolute and relative measures of EXAM score. The improvement achieved by the proposed approach over the classic Ochiai method in terms of relative changes in the EXAM score values is summarized in column 10 of both the tables.

By observing row 1 of Table XV, we can note that in order to locate fault in 'print_tokens' program the traditional Ochiai method requires 16 statements to be examined (i.e. in relative terms it requires 2.83% of total code to be inspected), whereas our proposed approach in this particular case is able to locate the fault directly in one search with an improvement of 93.75%. Likewise, if we observe row 7 of Table XVI ('sed' UNIX utility program, version V2), we find that 31 statements need to be searched (0.26% of total code) by the classic Ochiai technique and only 3 statements need to be searched (0.02%) by our proposed approach, with an improvement of 90.32%. Similarly version 21 of 'space' program (row 12 of Table XVI) requires 95 statements to be checked (1.04% of code) by the classic Ochiai, while, only 9 statements are required to be examined (0.10% of code) by our proposed approach, with an improvement of 90.53% as shown in the last column of Table XVI.

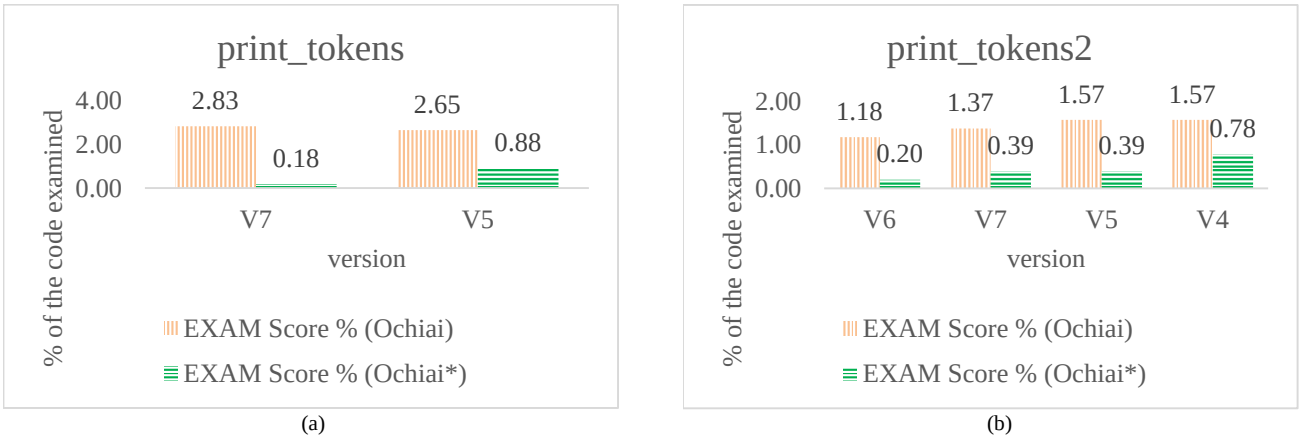
The average improvement achieved by the proposed approach (Ochiai*) over the classic Ochiai method in terms of relative EXAM score is shown in the last row of Table XV (for Siemens subject programs) and Table XVI (for large real-world subject programs) . We can note that there is an overall improvement of 62.76% on Siemens suite programs and 65.23% on large real-world programs.

TABLE XV
IMPROVEMENT IN THE PERFORMANCE OF FAULT LOCALIZATION ON SIEMENS PROGRAMS BY THE PROPOSED APPROACH (OCHIAI*)

Subject Program	Version	LOC	Line of Code Examined				Difference		Improvement %
			Ochiai	EXAM Score % (Ochiai)	Using Proposed Approach (Ochiai*)	EXAM Score % (Ochiai*)	LOC Examined	EXAM Score %	
print_tokens	V7	565	16	2.83	1	0.18	-15	-2.65	93.75
print_tokens	V5	565	15	2.65	5	0.88	-10	-1.77	66.67
print_tokens2	V6	510	6	1.18	1	0.20	-5	-0.98	83.33
print_tokens2	V7	510	7	1.37	2	0.39	-5	-0.98	71.43
print_tokens2	V5	510	8	1.57	2	0.39	-6	-1.18	75.00
print_tokens2	V4	510	8	1.57	4	0.78	-4	-0.78	50.00
replace	V6	562	28	4.98	1	0.18	-27	-4.80	96.43
replace	V7	562	13	2.31	4	0.71	-9	-1.60	69.23
replace	V1	562	22	3.91	4	0.71	-18	-3.20	81.82
replace	V4	562	28	4.98	17	3.02	-11	-1.96	39.29
replace	V3	562	76	13.52	36	6.41	-40	-7.12	52.63
schedule	V3	412	5	1.21	1	0.24	-4	-0.97	80.00
schedule	V2	412	19	4.61	6	1.46	-13	-3.16	68.42
schedule	V4	412	15	3.64	9	2.18	-6	-1.46	40.00
schedule2	V6	307	17	5.54	3	0.98	-14	-4.56	82.35
schedule2	V7	307	92	29.97	46	14.98	-46	-14.98	50.00
tcas	V4	173	2	1.16	1	0.58	-1	-0.58	50.00
tcas	V1	173	13	7.51	3	1.73	-10	-5.78	76.92
tcas	V2	173	16	9.25	3	1.73	-13	-7.51	81.25
tcas	V3	173	27	15.61	19	10.98	-8	-4.62	29.63
tcas	V5	173	30	17.34	24	13.87	-6	-3.47	20.00
tot_info	V7	406	9	2.22	2	0.49	-7	-1.72	77.78
tot_info	V2	406	35	8.62	10	2.46	-25	-6.16	71.43
tot_info	V4	406	15	3.69	12	2.96	-3	-0.74	20.00
tot_info	V5	406	24	5.91	14	3.45	-10	-2.46	41.67
Average			21.84	6.29	9.20	2.88	-12.64	-3.41	62.76

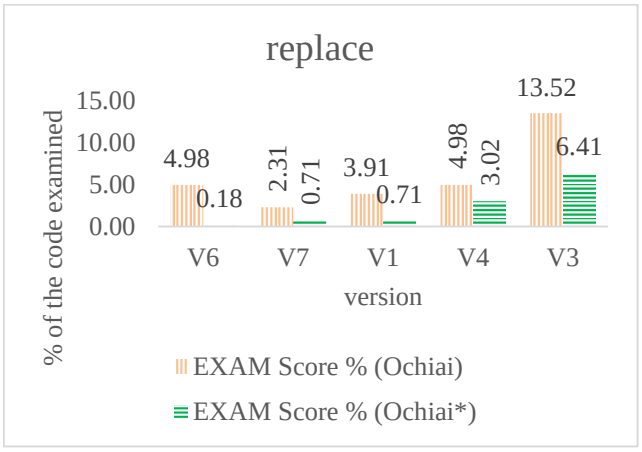
TABLE XVI
IMPROVEMENT IN PERFORMANCE OF FAULT LOCALIZATION ON LARGE REAL-WORLD PROGRAMS BY THE PROPOSED APPROACH (OCHIAI*)

Subject Program	Version	LOC	Line of Code Examined				Difference		Improvement %
			Ochiai	EXAM Score % (Ochiai)	Using Proposed Approach (Ochiai*)	EXAM Score % (Ochiai*)	LOC Examined	EXAM Score %	
flex	V3	13892	23	0.17	9	0.06	-14	-0.10	60.87
flex	V1	13892	38	0.27	17	0.12	-21	-0.15	55.26
flex	V2	13892	52	0.37	29	0.21	-23	-0.17	44.23
flex	V4	13892	58	0.42	32	0.23	-26	-0.19	44.83
grep	V1	12653	33	0.26	1	0.01	-32	-0.25	96.97
grep	V2	12653	20	0.16	17	0.13	-3	-0.02	15.00
sed	V2	12062	31	0.26	3	0.02	-28	-0.23	90.32
sed	V3	12062	38	0.32	5	0.04	-33	-0.27	86.84
space	V23	9126	23	0.25	2	0.02	-21	-0.23	91.30
space	V20	9126	147	1.61	7	0.08	-140	-1.53	95.24
space	V18	9126	50	0.55	8	0.09	-42	-0.46	84.00
space	V21	9126	95	1.04	9	0.10	-86	-0.94	90.53
space	V5	9126	19	0.21	9	0.10	-10	-0.11	52.63
space	V15	9126	29	0.32	16	0.18	-13	-0.14	44.83
space	V14	9126	207	2.27	154	1.69	-53	-0.58	25.60
Average			57.53	0.56	21.20	0.21	-36.33	-0.36	65.23

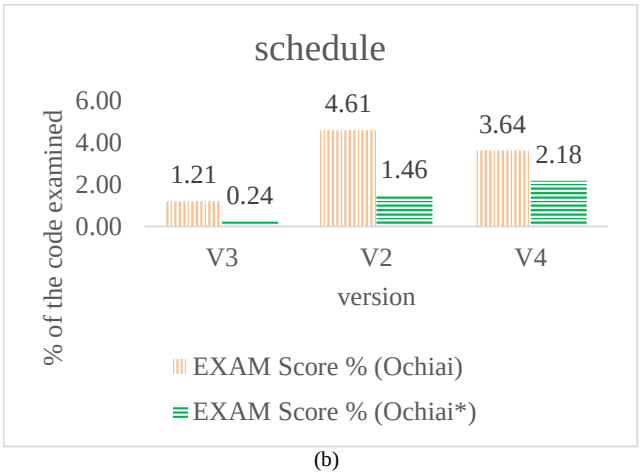


(a) (b)

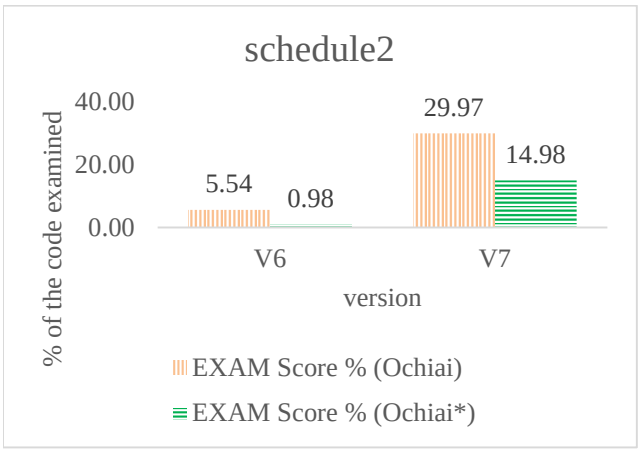
Fig. 4. Comparison of fault localization performance using EXAM score between the traditional Ochiai and the proposed approach Ochiai* on Siemens suite subject programs: (a) print_tokens (b) print_tokens2.



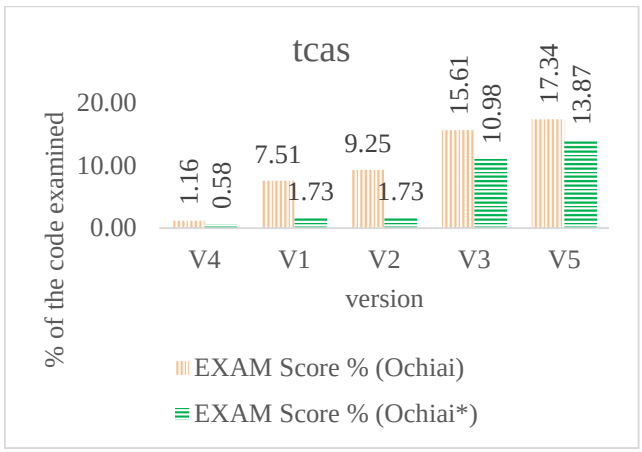
(a)



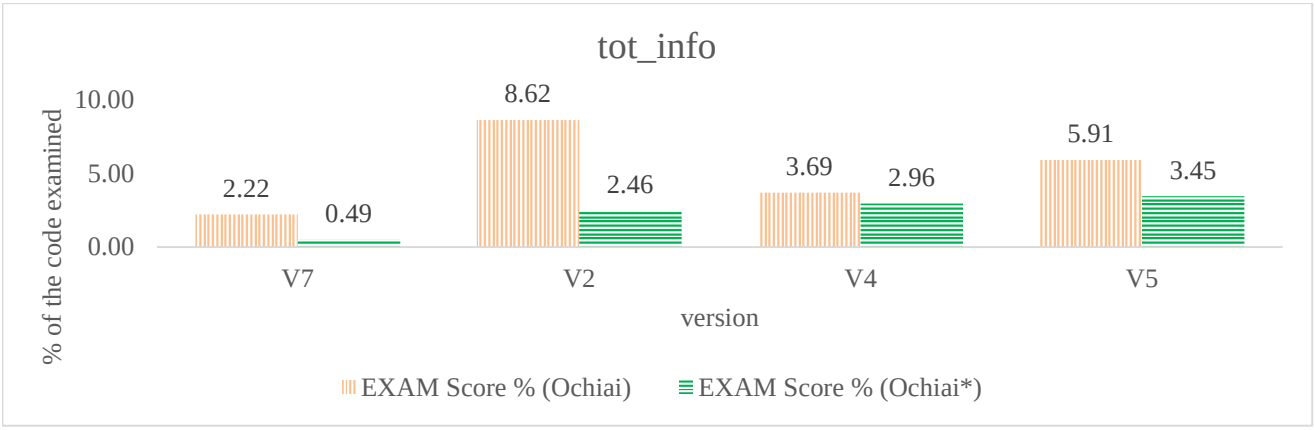
(b)



(c)

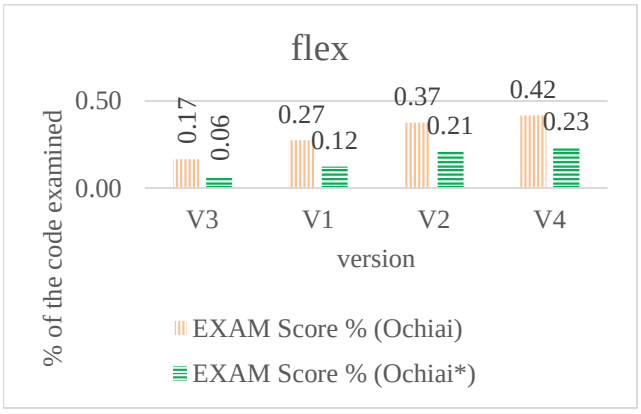


(d)

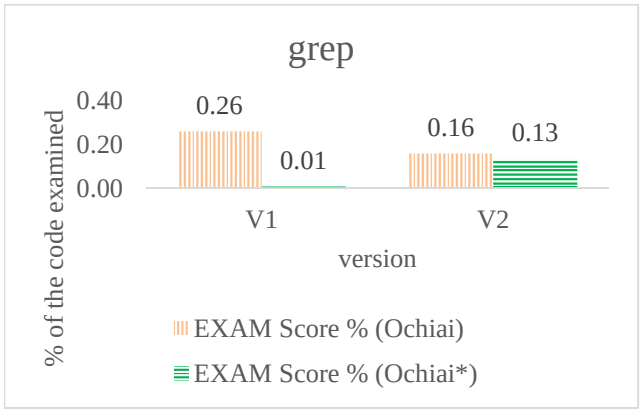


(e)

Fig. 5. Comparison of fault localization performance using EXAM score between the traditional Ochiai and the proposed approach Ochiai*on Siemens suite subject programs: (a) replace (b) schedule (c) schedule2 (d) tcas (e) tot_info.

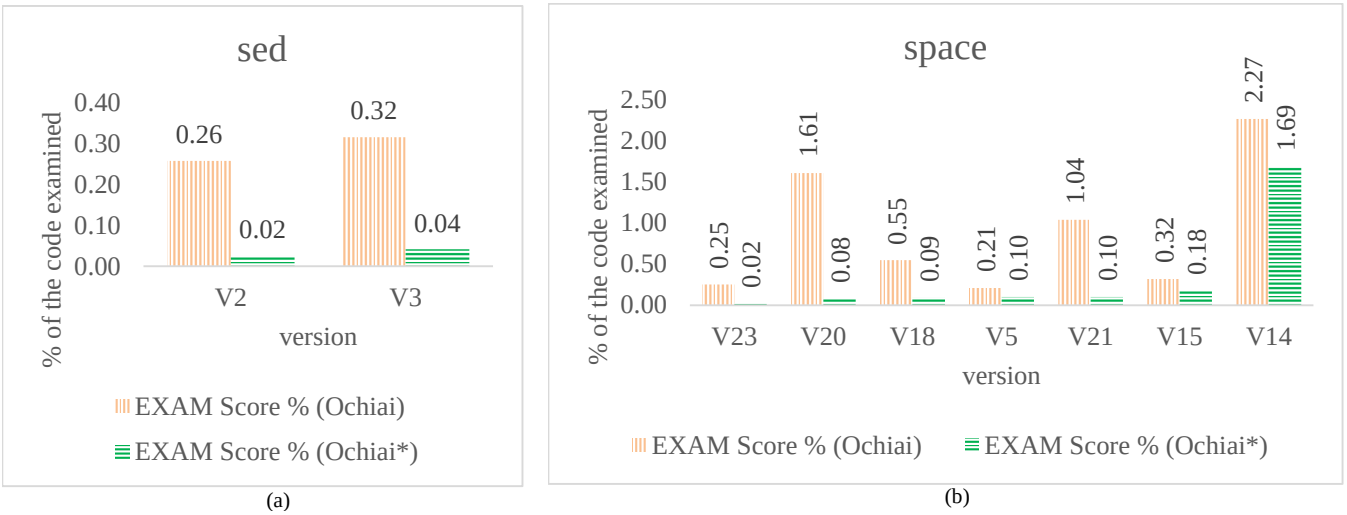


(a)



(b)

Fig. 6. Comparison of fault localization performance using EXAM score between the traditional Ochiai and the proposed approach Ochiai* on large real-world subject programs: (a) flex (b) grep.



(a) (b)

Fig. 7. Comparison of fault localization performance using EXAM score between the traditional Ochiai and the proposed approach Ochiai* on large real-world subject programs: (a) sed (b) space.

Furthermore, in order to better demonstrate the improvement achieved by our proposed approach compared to the existing SBFL techniques, we employ the EXAM score to assess its effectiveness in a more insightful manner. To illustrate this, we graphically depict the effectiveness of Ochiai* in comparison to classic Ochiai on the faulty versions of Siemens programs (Fig. 4 & Fig. 5) and the four large real-world programs (Fig. 6 & Fig. 7), respectively. The y axis represents the developer's effort in terms of percentage of code examined (relative EXAM score) and the x axis signifies the faulty version of the subject program being debugged. Upon analyzing Fig. 4(b), it becomes evident that in the case of version v6 of the 'print_tokens2' program, the proposed approach significantly reduces the effort required to locate the fault. For example, while 1.18% of the code needs to be searched using Ochiai metric, the proposed approach (Ochiai*) only requires tracing through 0.20% of the code. Likewise, when examining the 'grep' UNIX utility (version v1) in Fig. 6(b), the Ochiai method necessitates checking 0.26% of the code (33 LOC), while our proposed approach (referred to as Ochiai*) only requires searching through a mere 0.01% of the code (1 LOC).

We now present the results of applying our proposed fault localization approach to the widely-used Jaccard and DStar SBFL methods. This enhancement of Jaccard and DStar are referred to as Jaccard* and DStar*, respectively. Table XVII and Table XVIII compare the improvement achieved by Jaccard* over classic Jaccard on Siemens programs and large real-world programs, respectively. As a specific case, for example, the version V7 of the subject program 'replace' (row 8 in Table XVII) requires only 5 statements to be inspected for locating a fault by the Jaccard*, whereas 15 statements are required to be checked if we use simple Jaccard method.

TABLE XVII
IMPROVEMENT IN THE PERFORMANCE OF FAULT LOCALIZATION ON SIEMENS PROGRAMS BY THE PROPOSED APPROACH (JACCARD*)

Subject Program	Version	LOC	Line of Code Examined				Difference		Improvement %
			Using Jaccard	EXAM Score % (Jaccard)	Using Proposed Approach (Jaccard*)	EXAM Score % (Jaccard*)	Code Examined	EXAM Score %	
print_tokens	V7	565	16	2.83	1	0.18	-15	-2.65	93.75
print_tokens	V5	565	15	2.65	6	1.06	-9	-1.59	60.00
print_tokens2	V6	510	13	2.55	2	0.39	-11	-2.16	84.62
print_tokens2	V4	510	8	1.57	4	0.78	-4	-0.78	50.00
print_tokens2	V7	510	18	3.53	4	0.78	-14	-2.75	77.78
print_tokens2	V5	510	6	1.18	6	1.18	0	0.00	0.00
replace	V1	562	12	2.14	4	0.71	-8	-1.42	66.67
replace	V7	562	15	2.67	5	0.89	-10	-1.78	66.67
replace	V6	562	28	4.98	11	1.96	-17	-3.02	60.71
replace	V4	562	28	4.98	17	3.02	-11	-1.96	39.29
replace	V3	562	96	17.08	39	6.94	-57	-10.14	59.38
schedule	V3	412	2	0.49	1	0.24	-1	-0.24	50.00
schedule	V2	412	10	2.43	6	1.46	-4	-0.97	40.00
schedule	V4	412	15	3.64	9	2.18	-6	-1.46	40.00
schedule2	V6	307	17	5.54	3	0.98	-14	-4.56	82.35
schedule2	V7	307	81	26.38	48	15.64	-33	-10.75	40.74
tcas	V2	173	6	3.47	4	2.31	-2	-1.16	33.33
tcas	V4	173	10	5.78	5	2.89	-5	-2.89	50.00
tcas	V1	173	20	11.56	8	4.62	-12	-6.94	60.00
tcas	V3	173	27	15.61	13	7.51	-14	-8.09	51.85
tcas	V5	173	30	17.34	24	13.87	-6	-3.47	20.00
tot_info	V7	406	9	2.22	2	0.49	-7	-1.72	77.78
tot_info	V5	406	10	2.46	6	1.48	-4	-0.99	40.00
tot_info	V4	406	15	3.69	7	1.72	-8	-1.97	53.33
tot_info	V2	406	35	8.62	24	5.91	-11	-2.71	31.43
Average			21.68	6.22	10.36	3.17	-11.32	-3.05	53.19

TABLE XVIII
IMPROVEMENT IN PERFORMANCE OF FAULT LOCALIZATION ON LARGE REAL-WORLD PROGRAMS BY THE PROPOSED APPROACH (JACCARD*)

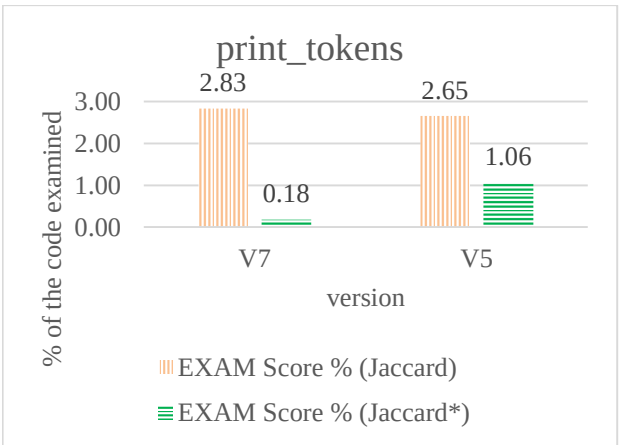
Subject Program	Version	LOC	Line of Code Examined				Difference		Improvement %
			Using Jaccard	EXAM Score % (Jaccard)	Using Proposed Approach (Jaccard*)	EXAM Score % (Jaccard*)	Code Examined	EXAM Score %	
flex	V3	13892	28	0.20	11	0.08	-17	-0.12	60.71
flex	V2	13892	45	0.32	20	0.14	-25	-0.18	55.56
flex	V1	13892	49	0.35	27	0.19	-22	-0.16	44.90
flex	V4	13892	63	0.45	27	0.19	-36	-0.26	57.14
grep	V1	12653	33	0.26	4	0.03	-29	-0.23	87.88
grep	V2	12653	28	0.22	17	0.13	-11	-0.09	39.29
sed	V2	12062	13	0.11	2	0.02	-11	-0.09	84.62
sed	V3	12062	18	0.15	2	0.02	-16	-0.13	88.89
space	V23	9126	23	0.25	7	0.08	-16	-0.18	69.57
space	V18	9126	50	0.55	8	0.09	-42	-0.46	84.00
space	V5	9126	23	0.25	9	0.10	-14	-0.15	60.87
space	V20	9126	147	1.61	27	0.30	-120	-1.31	81.63
space	V15	9126	57	0.62	29	0.32	-28	-0.31	49.12
space	V21	9126	145	1.59	49	0.54	-96	-1.05	66.21
space	V14	9126	235	2.58	107	1.17	-128	-1.40	54.47
Average			63.80	0.63	23.07	0.23	-40.73	-0.41	65.66

Thus, in this case the Jaccard* has achieved an improvement of 66.67% in fault localization result over the classic Jaccard method. Similarly, if we observe Table XVIII, we find that in case of 'sed' UNIX program (row 8, version v3), there is an improvement of 88.89% in fault localization results in favor of Jaccard*.

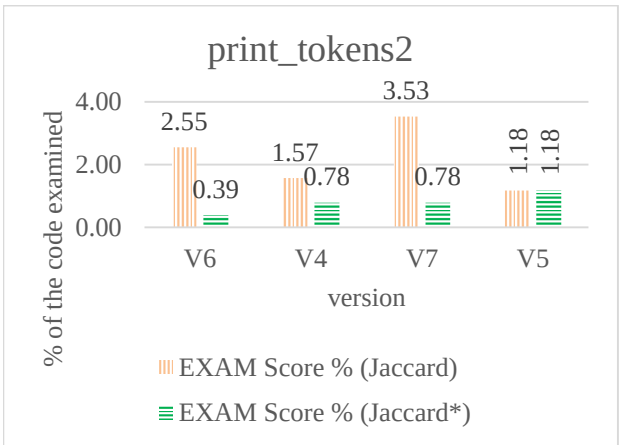
It can be observed that the Jaccard* method demonstrates a significant enhancement, with an overall average improvement of 53.19% on Siemens suite programs and 65.66% on large real-world subject programs compared to the traditional Jaccard approach, as depicted in the last row of Table XVII and Table XVIII, respectively.

Fig. 8 to Fig. 11 graphically highlight the effectiveness of Jaccard* in comparison to the classic Jaccard method for the faulty versions of each subject program in the Siemens suite and in the large real-world programs, respectively. In Fig. 8 (a), we observe that for version v7 of the 'print_tokens' subject, the classic Jaccard method necessitates checking 2.83% of the code, whereas the proposed approach (Jaccard*) only requires 0.18% of the code to be checked. Similarly, for version v5 of print_tokens, Jaccard* examines just 1.06% of the code compared to Jaccard, which needs 2.65% of the code to be examined. Based on the observation of Fig. 9 (c) for 'schedule2' program (version v6), it is evident that the Jaccard method requires 5.54% of the code to be searched in order to locate the fault, whereas Jaccard* only needs 0.98% of the code to be inspected for the same purpose. Moving on to version v7 of 'schedule2' program, Jaccard* demonstrates higher efficiency, as it requires only 15.64% of the code to be checked compared to Jaccard, which needs 37.46% of the code to be searched to identify the faulty statement. Similarly, if we observe other graphs shown in Fig. 8 and Fig. 9, it is evident that Jaccard* (proposed approach) outperforms the traditional Jaccard in most of the cases.

In the same way as shown in Fig. 10 (a), the 'flex' program (version V3) requires 0.20% (28 LOC) and 0.08% (11 LOC) of the code to be examined by Jaccard and Jaccard*, respectively. Likewise, when examining 'grep' UNIX utility for version v1 in Fig. 10 (b), we observe that the Jaccard method entails checking 0.26% of the code (33 LOC), whereas Jaccard* only needs to search through a mere 0.03% of the code (4 LOC). Also, in Fig. 11 (b), when analyzing 'space' program (version v20), Jaccard proves to be less efficient, as it necessitates the examination of 1.61% of the code (147 LOC) to locate the faulty statement. In contrast, Jaccard* outperforms it, as it only needs to search 0.30% of the code (27 LOC) to achieve the same result. Similarly, for the 'space' program (version v23), the classic Jaccard method necessitates inspecting 0.25% of the code (23 LOC), while Jaccard* requires only 0.08% of the code (7 LOC) to be searched to identify the fault.



(a)



(b)

Fig. 8. Comparison of fault localization performance using EXAM score between the traditional Jaccard and the proposed approach Jaccard* on Siemens suite subject programs: (a) print_tokens (b) print_tokens2.

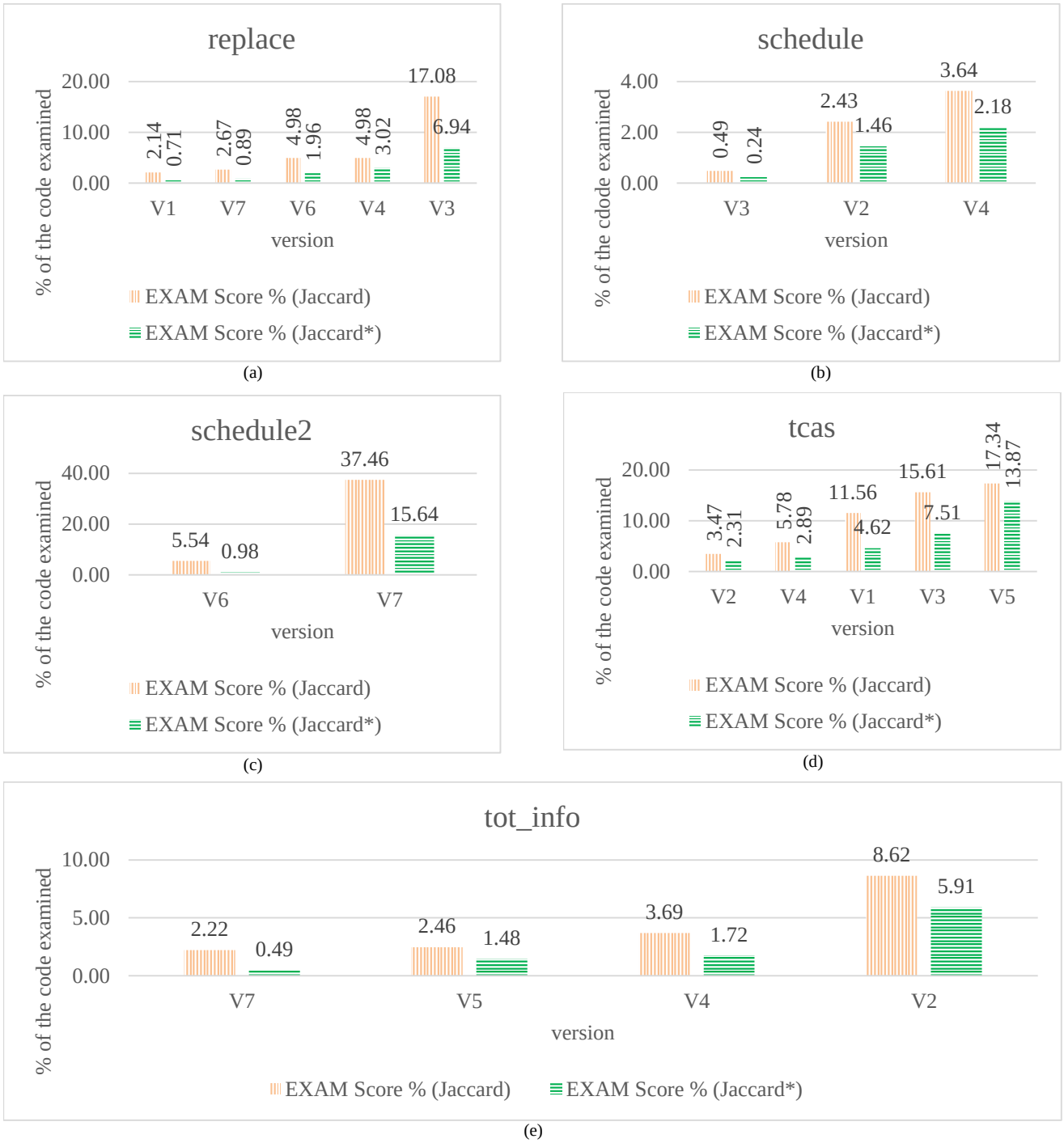


Fig. 9. Comparison of fault localization performance using EXAM score between the classic Jaccard and the proposed approach Jaccard* on Siemens suite subject programs: (a) replace (b) schedule (c) schedule2 (d) tcas (e) tot_info.

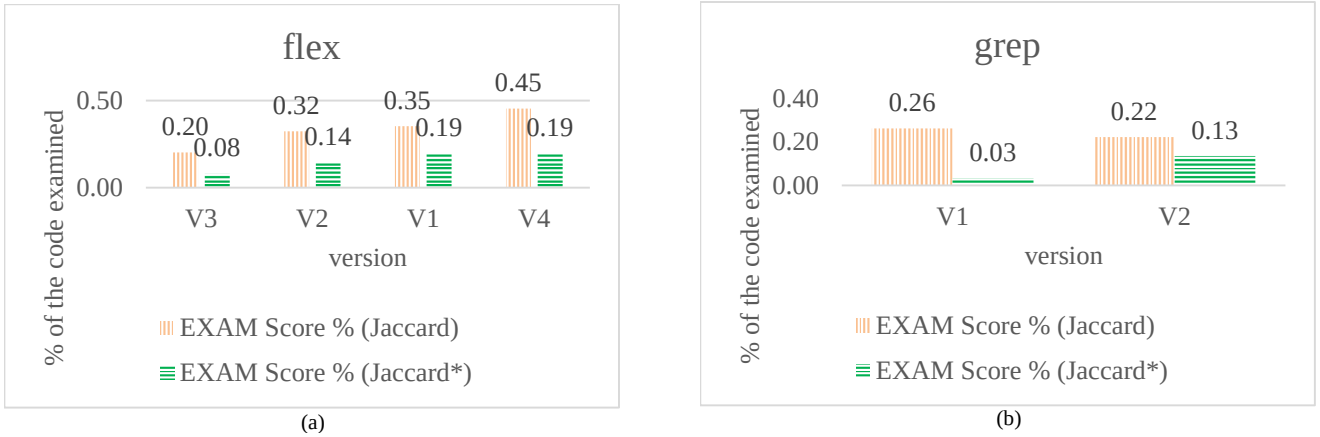
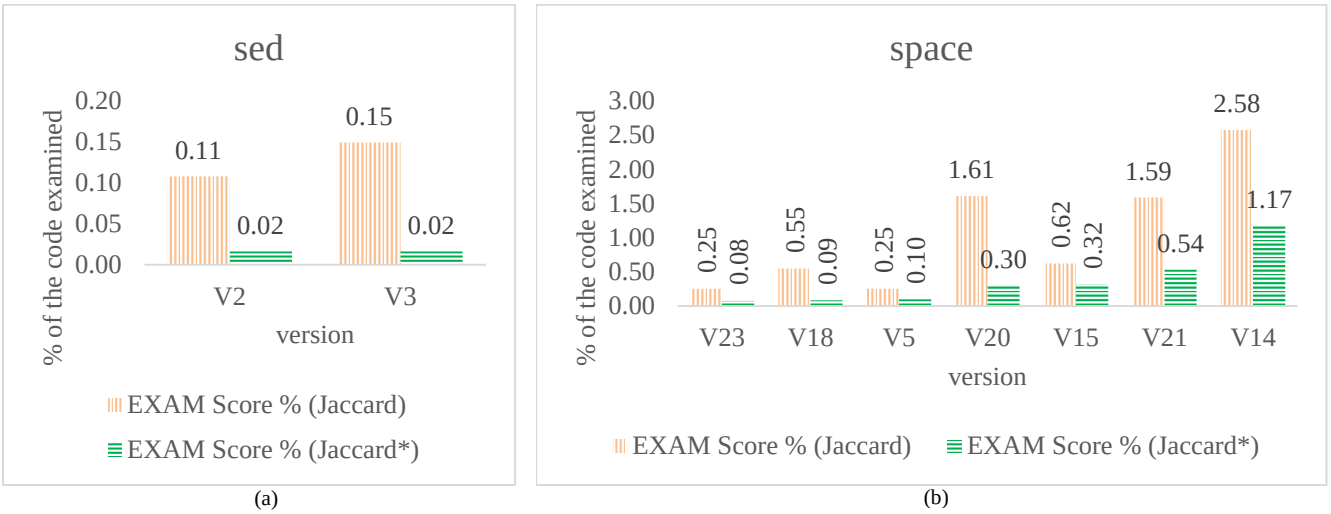


Fig. 10. Comparison of fault localization performance using EXAM score between the traditional Jaccard and the proposed approach Jaccard* on large real-world subject programs: (a) flex (b) grep.



(a) (b)

Fig. 11. Comparison of fault localization performance using EXAM score between the classic Jaccard and the proposed approach Jaccard* on large real-world subject programs: (a) sed (b) space.

TABLE XIX
IMPROVEMENT IN THE PERFORMANCE OF FAULT LOCALIZATION ON SIEMENS PROGRAMS BY THE PROPOSED APPROACH (DSTAR*)

Subject Program	Version	LOC	Line of Code Examined				Difference		Improvement %
			Using DStar	EXAM Score % (DStar)	Using Proposed Approach (DStar*)	EXAM Score % (DStar*)	Code Examined	EXAM Score %	
print_tokens	V7	565	16	2.83	1	0.18	-15	-2.65	93.75
print_tokens	V5	565	12	2.12	5	0.88	-7	-1.24	58.33
print_tokens2	V7	510	8	1.57	2	0.39	-6	-1.18	75.00
print_tokens2	V4	510	6	1.18	4	0.78	-2	-0.39	33.33
print_tokens2	V5	510	20	3.92	14	2.75	-6	-1.18	30.00
print_tokens2	V6	510	23	4.51	16	3.14	-7	-1.37	30.43
replace	V1	562	11	1.96	3	0.53	-8	-1.42	72.73
replace	V7	562	4	0.71	4	0.71	0	0.00	0.00
replace	V6	562	13	2.31	9	1.60	-4	-0.71	30.77
replace	V4	562	28	4.98	17	3.02	-11	-1.96	39.29
replace	V3	562	82	14.59	41	7.30	-41	-7.30	50.00
schedule	V3	412	14	3.40	1	0.24	-13	-3.16	92.86
schedule	V2	412	14	3.40	6	1.46	-8	-1.94	57.14
schedule	V4	412	9	2.18	7	1.70	-2	-0.49	22.22
schedule2	V6	307	7	2.28	3	0.98	-4	-1.30	57.14
schedule2	V7	307	28	9.12	18	5.86	-10	-3.26	35.71
tcas	V1	173	21	12.14	4	2.31	-17	-9.83	80.95
tcas	V2	173	15	8.67	4	2.31	-11	-6.36	73.33
tcas	V4	173	29	16.76	6	3.47	-23	-13.29	79.31
tcas	V5	173	24	13.87	8	4.62	-16	-9.25	66.67
tcas	V3	173	27	15.61	20	11.56	-7	-4.05	25.93
tot_info	V7	406	7	1.72	2	0.49	-5	-1.23	71.43
tot_info	V5	406	14	3.45	8	1.97	-6	-1.48	42.86
tot_info	V2	406	35	8.62	10	2.46	-25	-6.16	71.43
tot_info	V4	406	38	9.36	12	2.96	-26	-6.40	68.42
Average			20.20	6.05	9.00	2.55	-11.20	-3.50	54.36

TABLE XX
IMPROVEMENT IN THE PERFORMANCE OF FAULT LOCALIZATION ON LARGE REAL-WORLD PROGRAMS BY THE PROPOSED APPROACH (DSTAR*)

Subject Program	Version	LOC	Line of Code Examined				Difference		Improvement %
			Using Dstar	EXAM Score % (Dstar)	Using Proposed Approach (DStar*)	EXAM Score % (DStar*)	Code Examined	EXAM Score %	
flex	V3	13892	24	0.17	7	0.05	-17	-0.12	70.83
flex	V1	13892	34	0.24	18	0.13	-16	-0.12	47.06
flex	V2	13892	51	0.37	13	0.09	-38	-0.27	74.51
flex	V4	13892	55	0.40	24	0.17	-31	-0.22	56.36
grep	V1	12653	28	0.22	1	0.01	-27	-0.21	96.43
grep	V2	12653	156	1.23	67	0.53	-89	-0.70	57.05
sed	V3	12062	30	0.25	2	0.02	-28	-0.23	93.33
sed	V2	12062	20	0.17	11	0.09	-9	-0.07	45.00
space	V23	9126	15	0.16	2	0.02	-13	-0.14	86.67
space	V20	9126	57	0.62	6	0.07	-51	-0.56	89.47
space	V18	9126	74	0.81	8	0.09	-66	-0.72	89.19
space	V5	9126	68	0.75	17	0.19	-51	-0.56	75.00
space	V14	9126	34	0.37	19	0.21	-15	-0.16	44.12
space	V21	9126	83	0.91	29	0.32	-54	-0.59	65.06
space	V15	9126	187	2.05	104	1.14	-83	-0.91	44.39
Average			61.07	0.58	21.87	0.21	-39.20	-0.37	68.96

Table XIX and Table XX shown above compare the improvement achieved by DStar* over classic DStar. In the case of the 'tcas' program (version v5) given in row 20 of Table XIX, the traditional DStar method requires searching through 24 statements (13.87% of the code), whereas DStar* only needs to check 8 statements (4.62% of the code) to identify the faulty statement. The last column shows that there is an improvement of 66.67% in terms of reduction in the developer's effort in searching for the fault. For the 'space' program (version v18) as shown in Table XX, DStar examines 74 statements (0.81% of the code), while DStar* only needs to search 8 statements (0.09% of the code) to find the fault, reducing the developer's effort by 89.19%. It is noteworthy that the DStar* method attains a notable improvement, achieving an overall average improvement of 54.36% on Siemens suite programs and 68.96% on large real-world programs compared to the conventional DStar, as evidenced in the last rows of Table XIX and Table XX, respectively.

We will now graphically compare the performance of DStar* and classic DStar on Siemens programs (see Fig. 12, and Fig. 13) and large real-world programs (see Fig. 14), respectively. For the 'schedule' program (version v3) in Fig. 12 (d), DStar checks 3.40% of the code (14 LOC) to find the fault, whereas DStar* inspects only 0.24% of the code (1 LOC) to locate the fault. For the tot_info program (version v7) in Fig. 13, DStar identifies faults by checking 1.72% of the code (7 LOC), whereas DStar* achieves the same with just 0.49% of the code (2 LOC) inspected. Similarly, for the 'flex' program (version v3) as shown in Fig. 14 (a), classic DStar searches through 0.17% of the code (24 LOC) to find the fault, while DStar* examines only 0.05% of the code (7 LOC) to identify the faulty statement. Similar results can be observed from the other graphs shown in Fig. 12, Fig. 13, and Fig. 14.

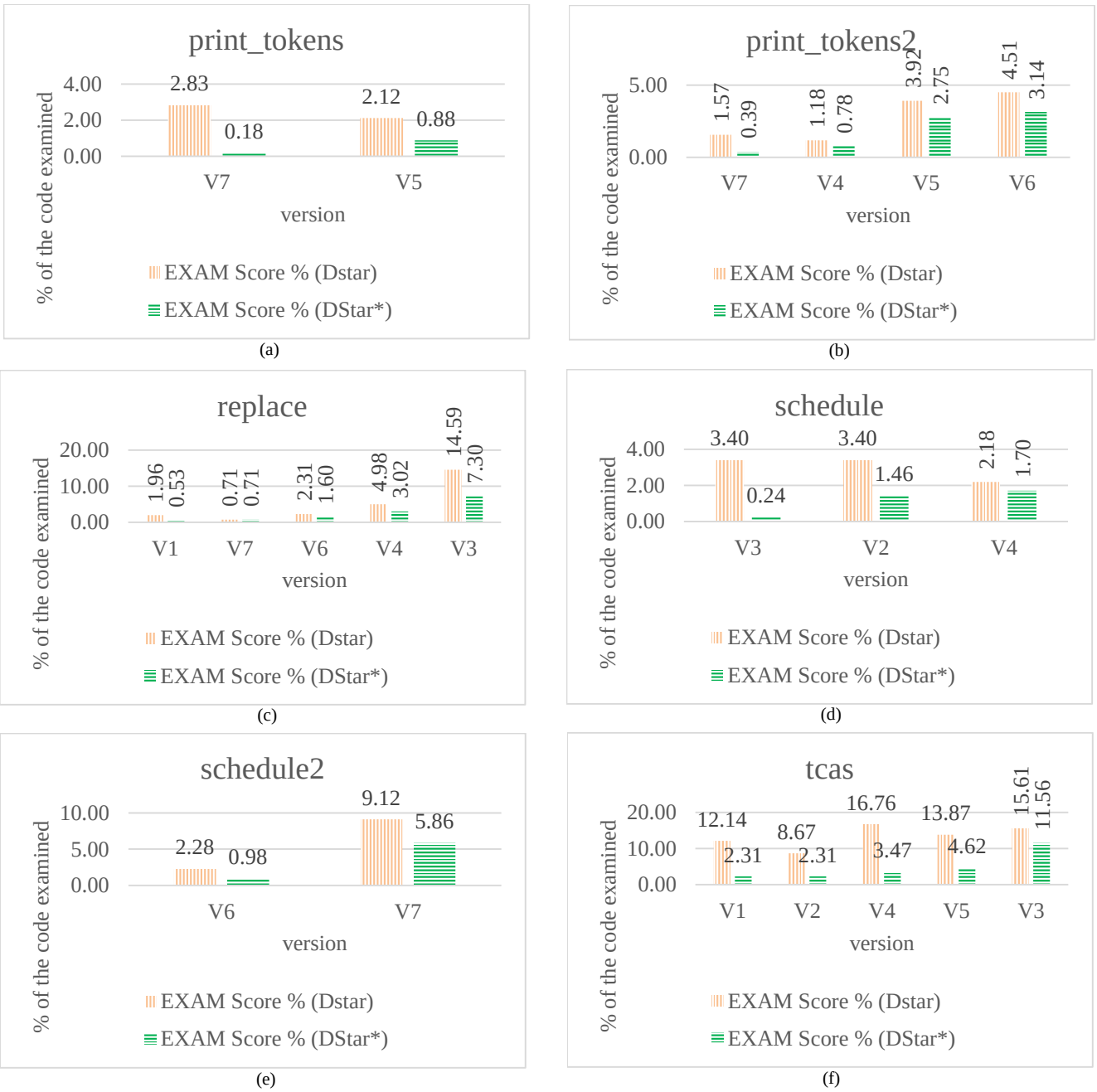


Fig. 12. Comparison of fault localization performance using EXAM score between the traditional DStar and the proposed approach DStar* on Siemens suite subject programs: (a) print_tokens (b) print_tokens2 (c) replace (d) schedule (e) schedule2 (f) tcas.

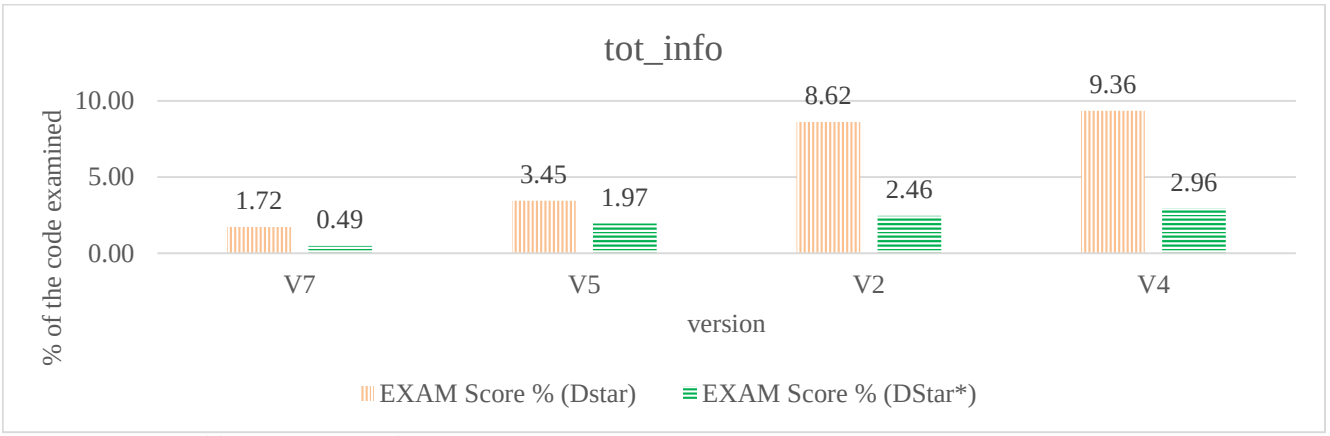
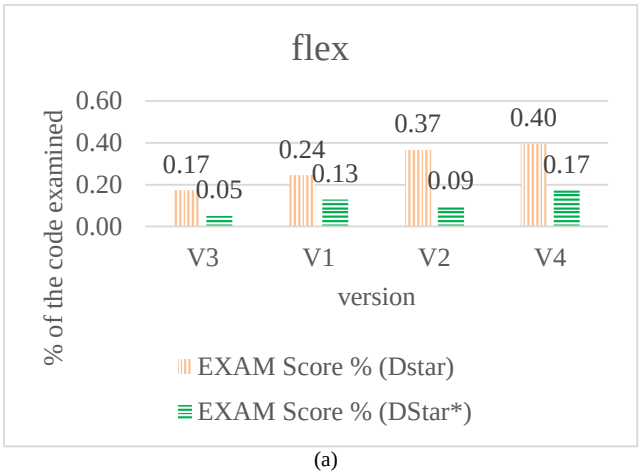
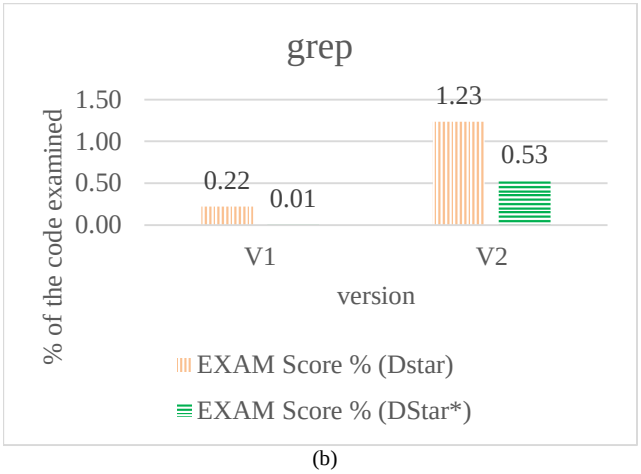


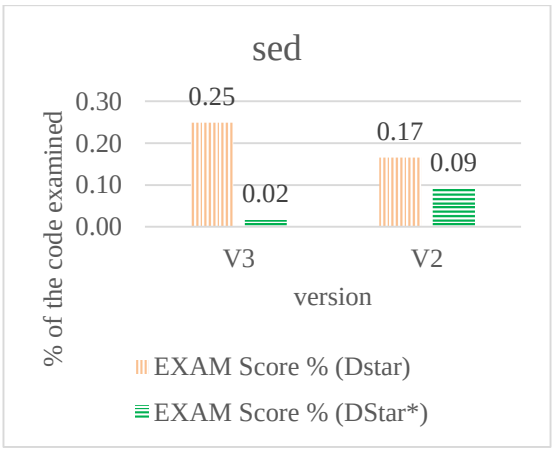
Fig. 13. Comparison of fault localization performance using EXAM score between the classic DStar and the proposed approach DStar* on Siemens suite subject program: tot_info.



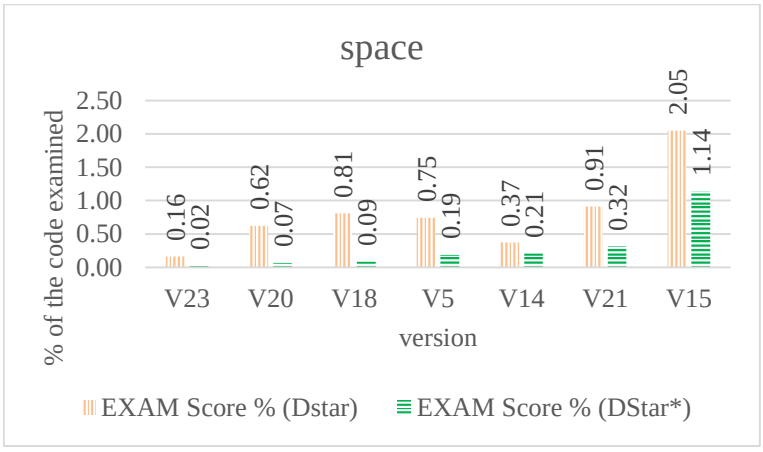
(a)



(b)



(c)

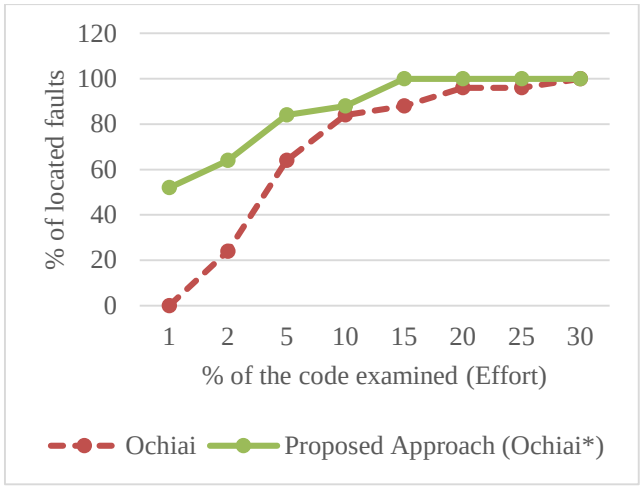


(d)

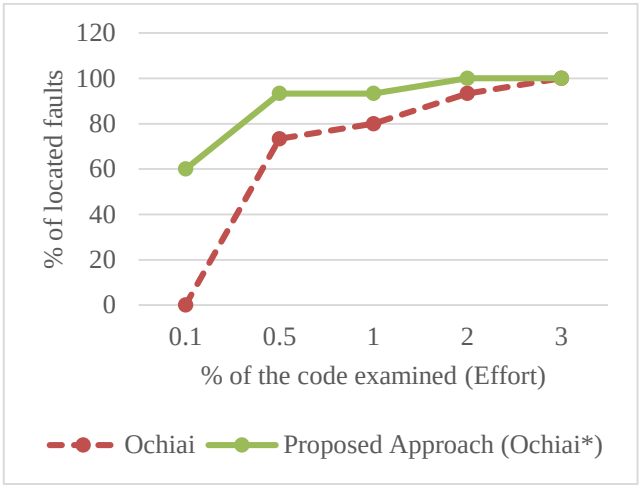
Fig. 14. Comparison of fault localization performance using EXAM score between the classic DStar and the proposed approach DStar* on large real-world subject programs: (a) flex (b) grep (c) sed (d) space.

Our results confirm improved fault localization with Jaccard* and DStar*, reinforcing the efficacy of our proposed approach/ framework.

In the following sections we present a comprehensive evaluation of our proposed approach's overall performance using three SBFL techniques Ochiai, Jaccard, and DStar. We assess their effectiveness based on the EXAM score, considering both Siemens programs and large real-world subject programs. Our proposed approach enhances the performance of classic SBFL techniques, resulting in improved versions labeled as Ochiai*, Jaccard*, and DStar*, as explained before. In Fig. 15, we compare the performance of Ochiai with that of Ochiai*, while Fig. 16 presents a comparison between Jaccard and Jaccard* and Fig. 17 displays the comparative analysis of DStar and DStar*. Fig. 15 (a) shows that Ochiai* can identify 52% of the faults in the faulty versions of the Siemens test suite subject programs by inspecting only 1% or less of the code. In contrast, the Ochiai method is unable to locate any fault by examining the same 1% or less of the code. Moreover, the Ochiai* can identify 64% of the faults in the faulty versions of the Siemens test suite by inspecting 2% or less of the code, whereas the traditional Ochiai method can only identify 24% of the faults.

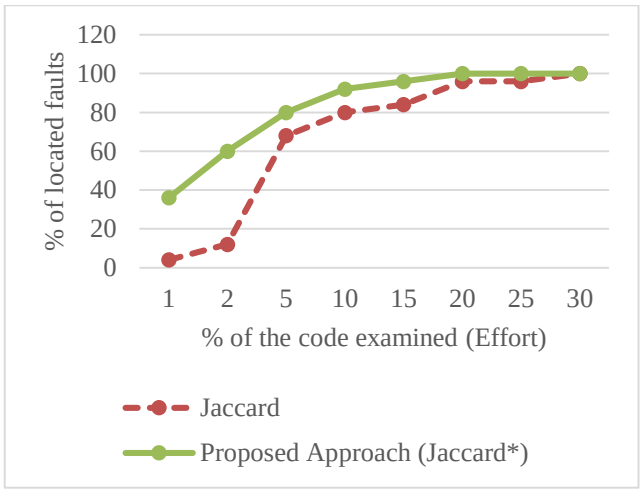


(a)

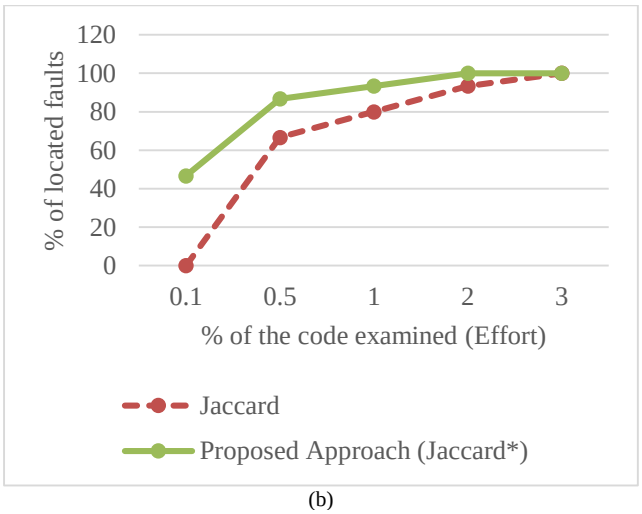


(b)

Fig. 15. Comparison of overall effectiveness using Exam Scores between classic Ochiai and the proposed approach Ochiai* (a) on Siemens test suite programs (b) on large real-world subject programs.

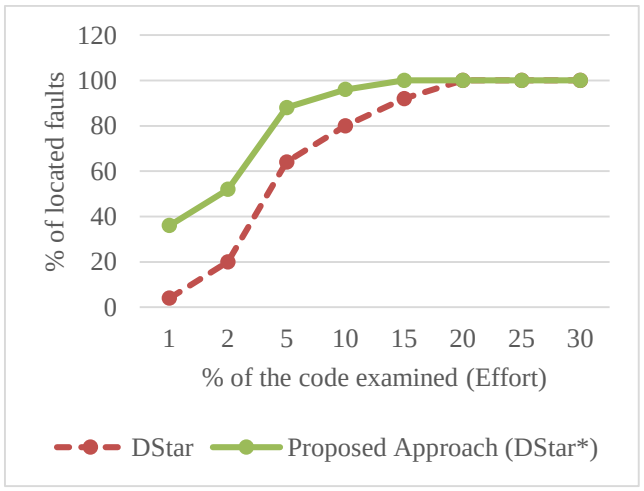


(a)

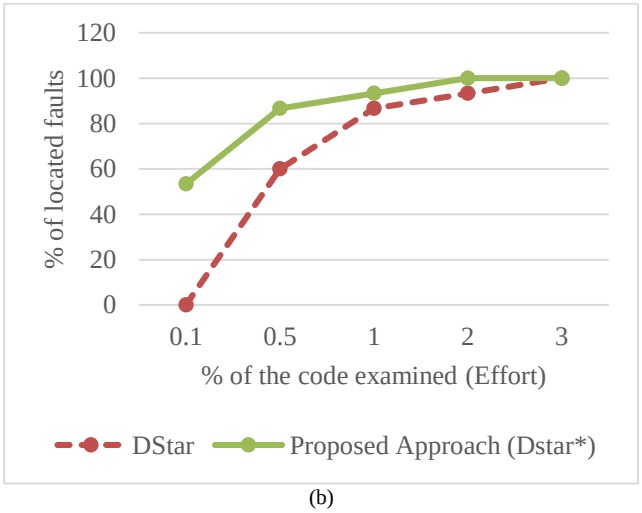


(b)

Fig. 16. Comparison of overall effectiveness using Exam Scores between classic Jaccard and the proposed approach Jaccard* (a) on Siemens test suite programs (b) on large real-world subject programs.



(a)



(b)

Fig. 17. Comparison of overall effectiveness using Exam Scores between classic DStar and the proposed approach DStar* (a) on Siemens test suite programs (b) on large real-world subject programs.

Now, as shown in Fig. 15 (b), we examine the fault localization performance of our proposed approach (Ochiai*) against the classic Ochiai method on the faulty versions of the large real-world subject programs. It can be noted that Ochiai* can locate 60% of the faults by inspecting less than or equal to 0.1% of the code, whereas the traditional Ochiai method is unable to locate any of the fault by examining the same percentage of the code (i.e. 0.1%). Similarly, 93.33% of the faults can be located by examining 0.5% of the code by Ochiai*, whereas, only 73.33% of the faults can be located by examining the same 0.5% of the code by the traditional Ochiai method.

Fig.16 shows a comparison between the overall effectiveness score of Jaccard* and the classic SBFL method Jaccard, in terms of EXAM score on both the Siemens test suite programs and the large real-world programs. By observing the curves in Fig. 16 (a), we can examine that the Jaccard* can locate 36% of the faults by examining less than or equal to 1% of the code with respect to the Siemens programs, whereas Jaccard can only locate 4% of the faults in this case. In the same way Jaccard* can locate 60% of the faults by inspecting less than or equal to 2% of the code, as against of this traditional Jaccard can only locate 12% of the faults on Siemens programs.

In Fig. 16 (b) we analyze the fault localization performance of Jaccard* in comparison to the classic Jaccard, using the faulty versions of the large real-world programs. By examining just 0.1% of the code, Jaccard* successfully identifies 47% of the faults, while the Jaccard method fails to locate any of the faults. Furthermore, when examining 0.5% of the lines of code, Jaccard* locates an impressive 87% of the faults, whereas the Jaccard method can only detect 67% of the faults.

In Fig. 17, we present a comparison of the overall effectiveness score between DStar* and the baseline SBFL method DStar. This evaluation is based on the EXAM score for both the Siemens test suite programs and the large real-world subject programs. By looking at the curves in Fig. 17 (a) it is clear that DStar* can locate 52% of the faults with respect to the Siemens programs by examining less than or equal to 2% of the code, in comparison to this classic DStar can only locate 20% of the faults. In the same way, Fig. 17 (b) shows that by examining 0.1% of the code, DStar* can locate 53% of the faults while the classic DStar method fails to locate any of the faults on large real-world programs. Similarly, by examining 0.5% of the code the DStar* can locate 87% of the faults while DStar is only able to locate 60% of the faults on large real-world subject programs.

The graphs depicted in Fig. 15, Fig. 16 and Fig. 17 clearly show that the proposed approach has a significant improvement over the baseline SBFL methods in terms of percentage of code examined (EXAM score) on all faulty versions in Siemens programs and in large real-world programs used in the experimentation. That means the proposed approach requires significantly less number of statements to be examined by the developer in order to locate faults.

Moreover, our proposed methodology has the ability to identify faults by examining a considerably smaller portion of code when compared to traditional approaches. This makes our framework a valuable tool for developers engaged in the development of real-world software applications.

Fig. 18 and Fig. 19 show comparison of improvement achieved by the three lightweight fault localization techniques after applying the proposed approach. The comparison is shown separately for Siemens subject programs and large real-world programs in Fig. 18 and Fig. 19, respectively. We can observe that Ochiai* is better in case of Siemens subject programs, whereas, DStar* is slightly better than the other two methods on large real-world subject programs.

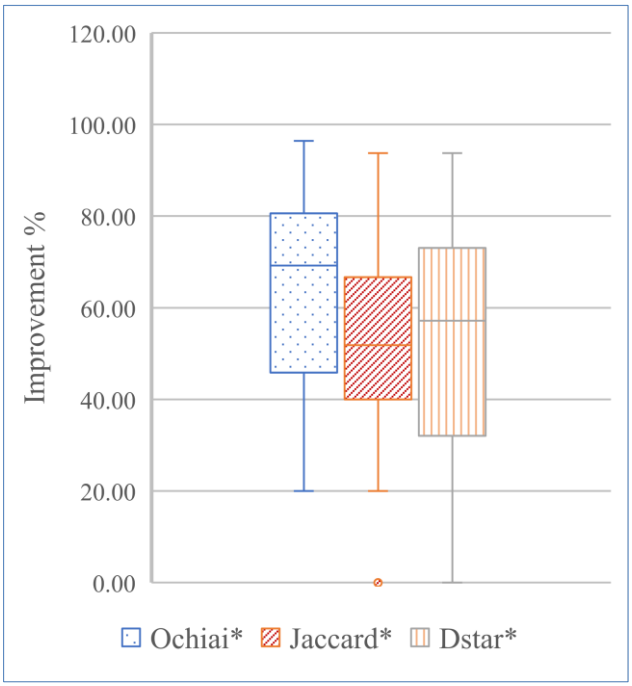


Fig. 18. Comparison of the improvement achieved by the proposed approach (Ochiai *, Jaccard* and DStar*) on Siemens programs.

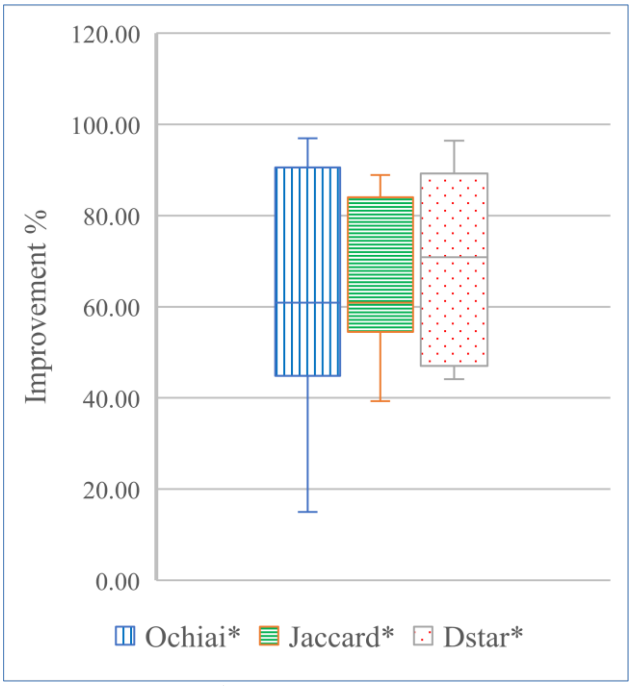


Fig. 19. Comparison of the improvement achieved by the proposed approach (Ochiai *, Jaccard* and DStar*) on large real-world subject programs.

iii. Assessing the efficiency of the proposed approach by using the Cumulative Number of Statements Examined metric

The Cumulative Number of Statements Examined (CSE) metric serves as a valuable tool for assessing the efficiency and effectiveness of fault localization approaches. It can assist researchers and developers in selecting the most suitable technique for their specific debugging needs.

TABLE XXI
CUMULATIVE NUMBER OF STATEMENTS EXAMINED TO LOCATE FAULTS FOR EACH SUBJECT PROGRAM IN SIEMENS SUITE

Approach/ Subject Program	print_tokens	print_tokens2	replace	schedule	schedule2	tcas	tot_info
Ochiai	31	29	167	39	109	88	83
Ochiai*	6	9	62	16	49	50	38
Jaccard	31	45	179	27	132	93	69
Jaccard*	7	16	76	16	51	54	39
DStar	28	57	138	37	35	116	94
DStar*	6	36	74	14	21	42	32

TABLE XXII
CUMULATIVE NUMBER OF STATEMENTS EXAMINED TO LOCATE FAULTS IN LARGE REAL-WORLD SUBJECT PROGRAMS

Approach/ Subject Program	flex	grep	sed	space
Ochiai	171	53	69	570
Ochiai*	87	18	8	205
Jaccard	185	61	31	680
Jaccard*	85	21	4	236
DStar	164	184	50	518
DStar*	62	68	13	185

TABLE XXIII
OVERALL CUMULATIVE NUMBER OF STATEMENTS EXAMINED TO LOCATE FAULTS ACROSS ALL FAULTY VERSIONS OF SUBJECT PROGRAMS USED IN THE EXPERIMENTAL STUDY

Approach/ Subject Program	Siemens Programs	Large Real-World Programs
Ochiai	546	863
Ochiai*	230	318
Jaccard	576	957
Jaccard*	259	346
DStar	505	916
DStar*	225	328

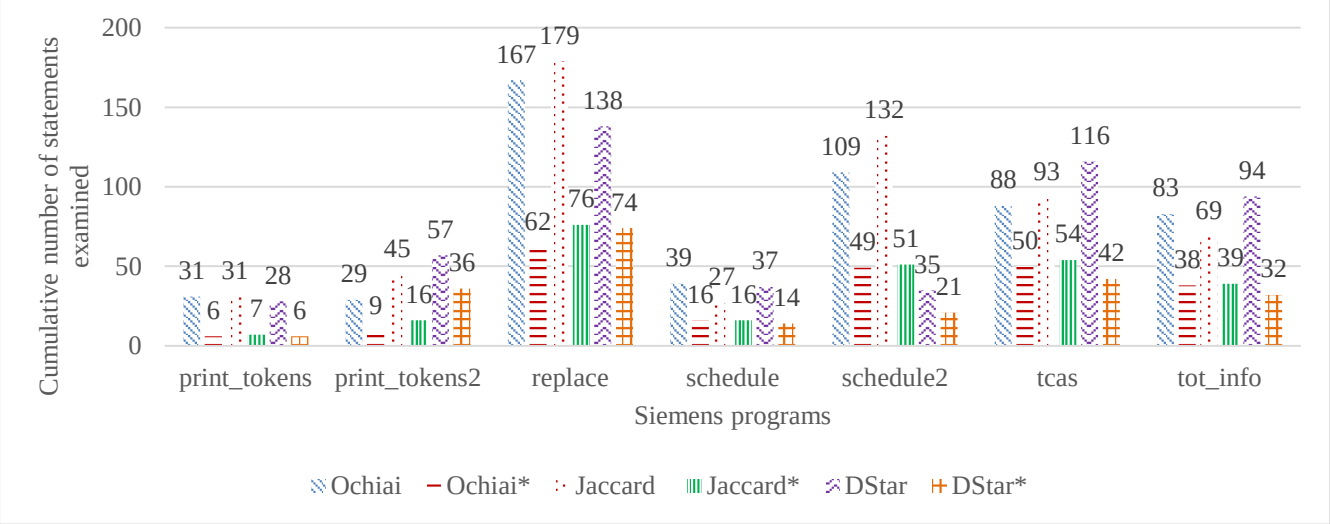


Fig. 20. Cumulative number of statements examined to locate faults across all faulty versions of Siemens test suite subject programs used in the experimental study.

In Table XXI and Table XXII, we present a comparative analysis of cumulative number of statements examined for fault localization. This assessment contrasts the performance of three classical SBFL methods, i.e., Ochiai, Jaccard, and DStar, before and after applying our proposed approach to Siemens programs and large real-world programs individually. Notably, our proposed approach enhances the efficiency of these classical SBFL techniques, and we denote their improved variants as Ochiai*, Jaccard*, and DStar*.

It is important to note that, the figures presented in Table XXI and Table XXII indicate the total number of statements that each fault localization method needs to examine to locate faults in all faulty versions of every subject program used in this study. The details of these subject programs and their different faulty versions (with single faults) used in the study is given in Table XIII. We have conducted experiments on a total of 40 faulty versions out of which 25 faulty versions are from standard benchmark seven Siemens programs and 15 faulty versions are from four large real-world programs- flex, grep, sed and space.

By observing Table XXI and Table XXII we can note that, for each program, the cumulative number of statements examined by the proposed approach is much smaller than the baseline methods Ochiai, Jaccard and DStar. In the case of the print_tokens program (see Table XXI), the Ochiai method requires examination of 31 statements to find the fault, while Ochiai* requires only 6 statements to be checked, reducing the examined statements by 25. In the same way, Jaccard requires 31 statements and DStar needs 28 statements, while Jaccard* and DStar* requires only 7 and 6 statements, respectively to locate the faults in print_tokens program. Analogously, it can be noted by examining Table XXII that the large real-world (UNIX utility) program ‘grep’ requires a total of 53 statements to be examined by the Ochiai method as against the total 18 statements required by our proposed approach Ochiai*. So, in this case 35 less number of statements need to be examined due to our proposed approach. Similarly, 61 statements are required by Jaccard and 184 statements by DStar in comparison to 21 and 68 statements by Jaccard* and DStar*, respectively. This highlights a significant decrease in the count of inspected statements when employing the proposed approach, thereby reducing the developer’s effort.

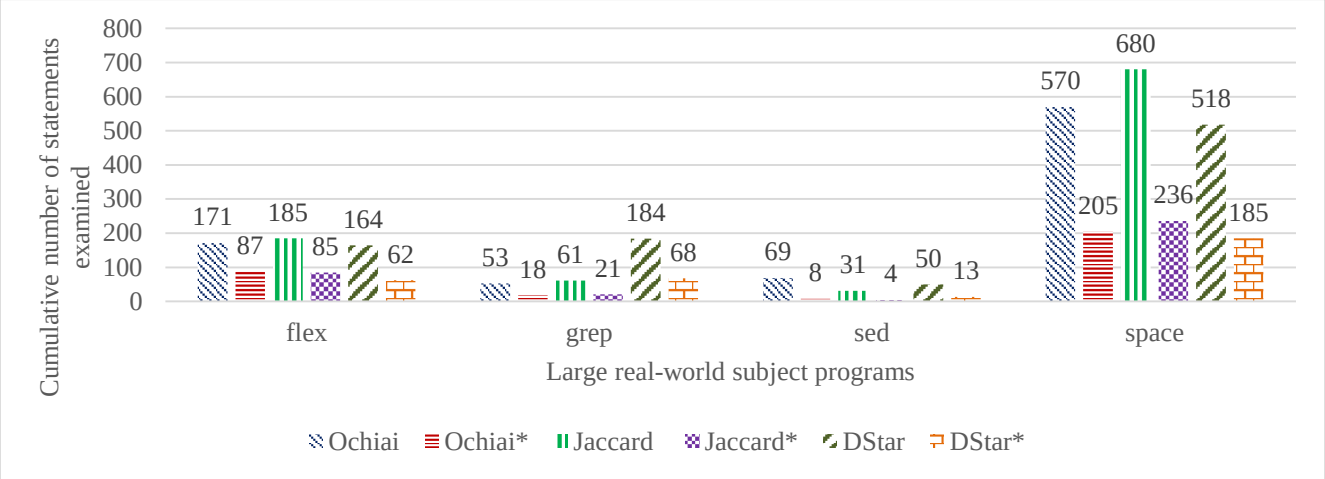


Fig. 21. Cumulative number of statements examined to locate faults across all faulty versions of large real-world subject programs used in the experimental study.

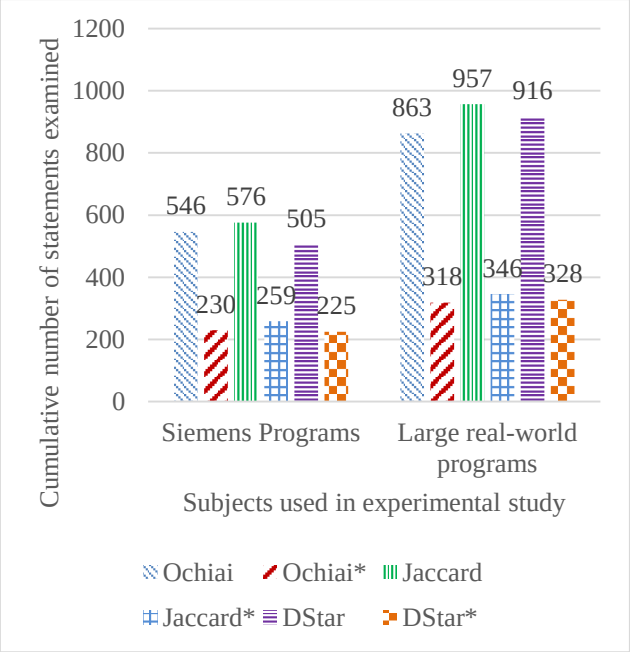


Fig. 22. Overall cumulative number of statements examined to locate faults across all faulty versions of subject programs (Siemens and large real-world programs) utilized in the experimental study.

Table XXIII provides an overall comparison between our proposed approach and the traditional SBFL methods. As previously mentioned, we performed experiments on 25 faulty versions of Siemens programs and 15 faulty versions of large real-world programs. Each version considered in our experimentation had a single fault. We can see that 546 statements need to be examined by Ochiai in order to locate faults in all 25 faulty versions of Siemens programs whereas the Ochiai* required only 230 statement to be examined in this case. Similarly, 863 statements need to be examined by the Ochiai method in order to locate faults in all 15 faulty versions of large real-world programs, whereas the Ochiai* requires only 318 statements to be examined. In the same way, we see that Jaccard needs 576 statements, while Jaccard* only requires 259 statements to be checked on Siemens programs, and 957 statements need to be examined on large real-world programs, as opposed to that 346 statements required by Jaccard*. If we compare the performance of DStar and DStar* we find the similar the trend. For Siemens programs and large real-world programs, DStar requires 505 and 916 statements to be checked, while DStar* requires 225 and 328 statements to be inspected to locate faults, respectively.

To enhance legibility and comprehension, we present a graphical comparison of the performance of the proposed approach with the traditional Ochiai, Jaccard and DStar methods using the Cumulative Number of Statements Examined metric on Siemens programs and large real-world programs in Fig. 20 and Fig. 21, respectively. Additionally, Fig. 22 provides an overall analysis of the performance between the proposed approach and the classic methods Ochiai, Jaccard and DStar, measured in terms of the cumulative number of statements examined metric, across all subject programs used in our experimental study (i.e. Siemens programs and large real-world programs).

iv.Assessing the efficiency of the proposed approach by using the Top-N metric

Table XXIV and Table XXV present fault localization results comparing the proposed approach with traditional SBFL techniques (i.e., Ochiai, Jaccard, and DStar) by means of the Top-N evaluation metric on Siemens programs and large real-world subject programs, respectively.

We use five metrics (Top-1, Top-5, Top-10, Top-15, and Top-20) to verify the effectiveness of our proposed approach against the traditional SBFL methods.

According to Table XXIV, the percentage of faults effectively located by the proposed approach Ochiai* on the Siemens test suite programs at the Top-1, Top-5, Top-10, and Top-15 positions are 20%, 60%, 72%, and 80%, respectively. As opposed to that, the percentages of faults located by the classic Ochiai method at the same positions are 0%, 8%, 28%, and 48%, respectively. In the same way, Jaccard* locates 8%, 44%, 72%, and 80% of the faults at the Top-1, Top-5, Top-10, and Top-15 positions, respectively, while Jaccard can locate only 0%, 4%, 32%, and 56% of the faults, respectively. Similar to Jaccard*, DStar* is able to locate 8%, 44%, 72%, and 80% of the faults at the Top-1, Top-5, Top-10, and Top-15 positions, respectively, while classic DStar is unable to locate any of the faults at the Top-1 position, and it can locate only 4%, 24%, and 52% of the faults at the Top-5, Top-10, and Top-15 positions, respectively.

Table XXV illustrates a comparison between the proposed approach and the classical SBFL methods on large real-world programs using the Top-N metric. The percentages of faults identified by the Ochiai* at the Top-1, Top-5, Top-10, and Top-15 positions, are 7%, 27%, 60%, and 60%, respectively. Conversely, the classic Ochiai method failed to locate any of the faults at these positions, as indicated in Table XXV.

TABLE XXIV
COMPARISON OF THE PERFORMANCE BETWEEN THE PROPOSED APPROACH AND THE TRADITIONAL SBFL METHODS IN TERMS OF TOP-N METRIC ON SIEMENS PROGRAMS

Approach	Percentage of faults identified for each Top-N metric					
	Top-1	Top-5	Top-10	Top-15	Top-20	Other
Ochiai	0%	8%	28%	48%	64%	36%
Ochiai*	20%	60%	72%	80%	88%	12%
Jaccard	0%	4%	32%	56%	72%	28%
Jaccard*	8%	44%	72%	80%	84%	16%
DStar	0%	4%	24%	52%	60%	40%
DStar*	8%	44%	72%	80%	96%	4%

TABLE XXV
COMPARISON OF THE PERFORMANCE BETWEEN THE PROPOSED APPROACH AND THE TRADITIONAL SBFL METHODS IN TERMS OF TOP-N METRIC ON LARGE REAL-WORLD PROGRAMS

Approach	Percentage of faults identified for each Top-N metric					
	Top-1	Top-5	Top-10	Top-15	Top-20	Other
Ochiai	0%	0%	0%	0%	13%	87%
Ochiai*	7%	27%	60%	60%	80%	20%
Jaccard	0%	0%	0%	7%	13%	87%
Jaccard*	0%	20%	40%	47%	60%	40%
DStar	0%	0%	0%	7%	13%	87%
DStar*	7%	20%	40%	53%	73%	27%

If we compare Jaccard* with Jaccard, we find that at the Top-1 position, both methods are unable to find any of the faults. However, at the Top-5, Top-10, and Top-15 positions, the Jaccard* can locate 20%, 40%, and 47% of the faults, respectively. As compared to that, Jaccard is still unable to locate any of the faults at the Top-5 and Top-10 positions, and is only able to locate 7% of the faults at the Top-15 position. As indicated in Table XXV, DStar and Jaccard exhibit nearly identical performances. Both methods fail to identify any faults at the Top-1, Top-5, and Top-10 positions. At the Top-15 position, both methods can only detect 7% of the faults. At the Top-1, Top-5, Top-10, and Top-15 positions, DStar* can locate 7%, 20%, 40%, and 53% of the faults, respectively.

To visually depict the comparison, we include graphical representations of the performance of the proposed approach against the classical SBFL methods (Ochiai, Jaccard, and DStar) using the Top-N metric. Fig. 23 and Fig. 24 illustrate this comparison on Siemens programs and large real-world programs, respectively.

It can be observed in more intuitive way by seeing Fig. 23 and Fig. 24 that our proposed approach outperforms the baseline SBFL methods in locating faults. In other words, applying the proposed approach to existing SBFL methods enhances their performance.

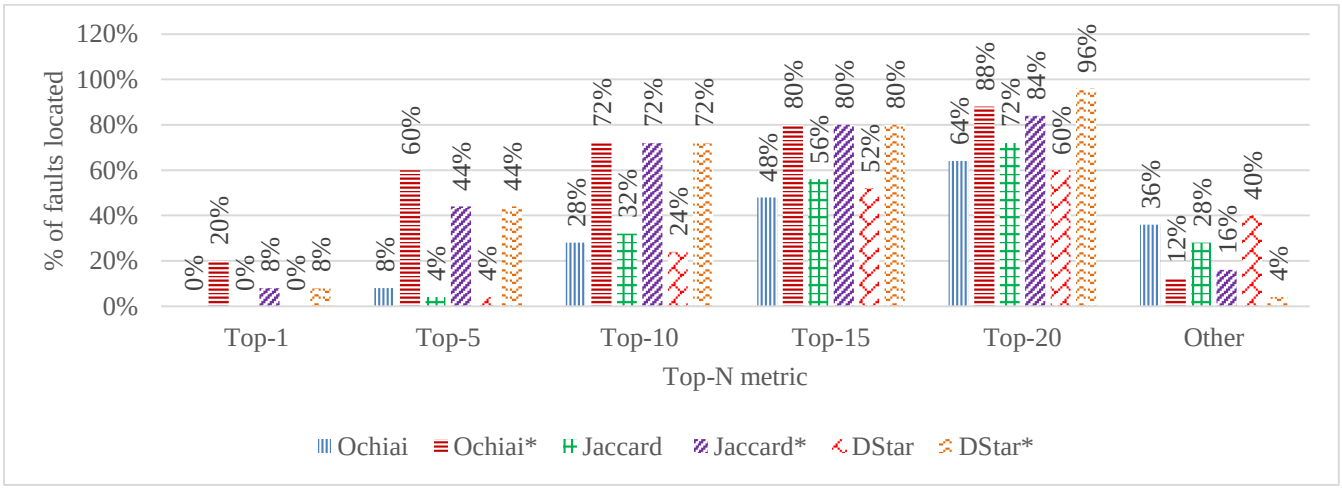


Fig. 23. Comparison of performance between conventional SBFL techniques and the proposed approach using the Top-N metric on Siemens programs.

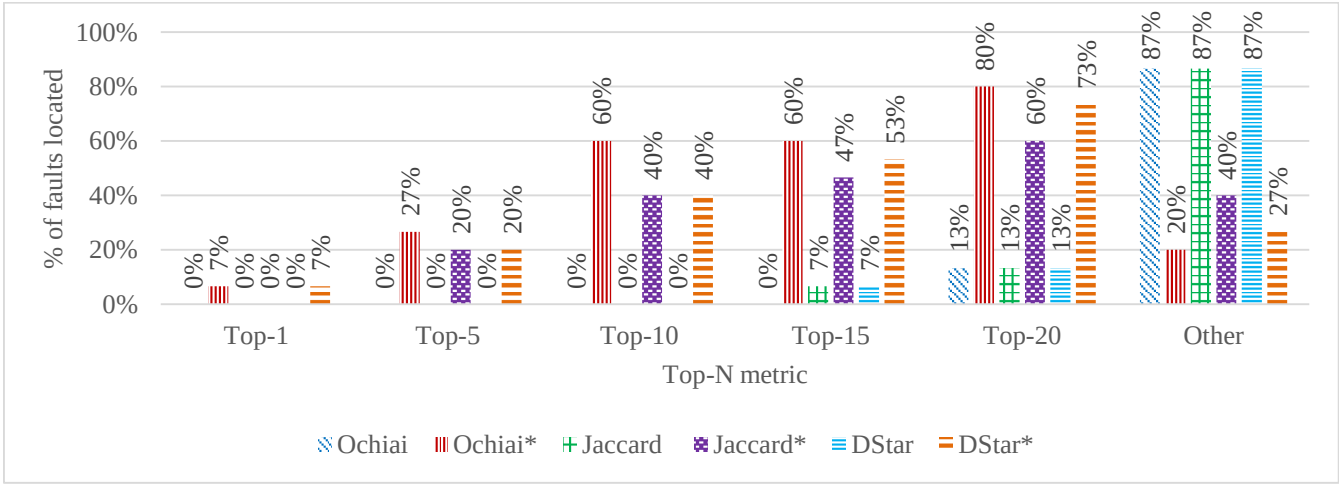


Fig. 24. Comparison of performance between conventional SBFL techniques and the proposed approach using the Top-N metric on large real-world subject programs.

v. Assessing the efficiency of the proposed approach by using Wilcoxon Signed-Rank Test

Fault localization research often involves the comparison of different techniques to determine which one is more effective at identifying the root cause of a fault. The Wilcoxon signed-rank test is a statistical test commonly used in fault localization research to compare the effectiveness of different techniques. This test is particularly useful for comparing two related samples, such as two fault localization techniques applied to the same set of programs.

The Wilcoxon signed-rank test involves ranking the differences between pairs of observations and determining whether the median difference is statistically significant. By using this test, fault localization researchers can evaluate the effectiveness of different techniques in a statistically sound manner, and make confident conclusions about the relative performance of each technique.

TABLE XXVI
WILCOXON SIGNED RANK TEST RESULTS OF THE THREE FAULT LOCALIZATION TECHNIQUES USING THE PROPOSED APPROACH AND WITHOUT USING THE PROPOSED APPROACH

Comparison	Subjects	2-tailed test (*p* values)	Conclusion
Ochiai vs Ochiai*	Siemens suite	1.29E-05	Better
	Large real-world programs	7.25E-04	Better
Jaccard vs Jaccard*	Siemens suite	1.92E-05	Better
	Large real-world programs	7.23E-04	Better
Dstar vs Dstar*	Siemens suite	1.92E-05	Better
	Large real-world programs	7.23E-04	Better

To reinforce our experimental results, we utilized the Wilcoxon signed-rank test as part of our evaluation methodology. This statistical test has extensive usage in comparing the efficacy of various methods and has been a conventional practice in earlier studies concerning fault localization. By adopting this approach, we can establish the validity of our conclusions through statistical analysis.

To assess the statistical significance of the difference between the traditional SBFL methods when employed with and without our proposed approach, we employ the Wilcoxon-Signed-Rank Test. This non-parametric statistical hypothesis test is used to examine the differences between pairs of measurements.

We conducted Wilcoxon-Signed-Rank tests, utilizing the ranks of faulty statements as pairs of measurements denoted as L(x) and M(y). Each test involved a two-tailed *p*-value assessment at a significance level (α level) of 0.01. Specifically, for L(x), we utilized the ranked list of faulty statements generated through the proposed approach (Ochiai*, Jaccard*, and DStar*) across all faulty versions of the subject programs from our experiments, as listed in Table XIII. For M(y), we used the ranked list of faulty statements obtained without using the proposed approach i.e. ranks calculated through the original SBFL methods (e.g. Ochiai, Jaccard, and DStar) on the same set of faulty program versions.

If the *p*-value is less than 0.01, then, in accordance with the alternative hypothesis (H_1), we accept that the suspiciousness ranks obtained using our proposed approach tend to be significantly smaller than those obtained without the proposed approach (i.e. obtained through traditional SBFL). This indicates that our proposed approach demonstrates superior effectiveness in comparison to the traditional SBFL methods. Conversely, if the *p*-value is greater than or equal to 0.01, in accordance with the null hypothesis (H0), we accept that the ranks obtained using our proposed approach do not significantly differ from those of the traditional SBFL methods, implying that the proposed approach does not outperform the established baseline SBFL methods.

Table XXVI shows the Wilcoxon-Signed-Rank Test results on this relationship, where the cells show the *p* values of Wilcoxon-Signed-Rank Tests. The results show that when existing SBFL methods use our proposed approach the ranks of the faulty statements are significantly smaller than those of baseline SBFL methods not using the proposed approach. Therefore, the alternative hypothesis is accepted at a confidence level of 99.99%., indicating that the proposed approach is more effective than the compared techniques (existing SBFL methods) in terms of examining fewer statements to detect faults. The 'Conclusion' column in Table XXVI indicates, that the improved versions of SBFL methods (Ochiai*, Jaccard*, DStar*), obtained by applying our proposed approach to the original SBFL method, exhibit better performance compared to the traditional SBFL methods.

In summary, the findings obtained through the Wilcoxon signed-rank test provide strong evidence that the proposed approach outperforms the compared techniques on the faulty versions of the subject programs in the Siemens test suite and the large real-world programs as listed in Table XIII. Our findings align with our previous conclusion that the proposed approach outperforms the traditional SBFL techniques in terms of efficiency, as evidenced by several metrics such as cumulative number of statements examined, Top-N, and Exam scores.

vi.Space and time complexity

In this section, we analyze the space and time complexity of our proposed approach. The input to our fault localization approach will be a program spectrum, which is a matrix of $N \times T$, where N is the number of program entities (i.e., program statements) and T is the number of test cases. According to the same problem settings used in Section II-D, test suite T consists of passing (T_P) and failing (T_F) test cases.

The space complexity is largely dependent on the space requirement to store the program spectrum matrix and the program execution result vector, which can be specified as $O(N \times T)$ and $O(T)$, respectively.

The time complexity can be determined by analyzing the time required for optimizing the test suite and computing the suspiciousness score for each program statement and the suspiciousness score of its fault context.

The test suite optimization requires primarily the calculation of the minimum suspiciousness set (MSS) and finding how closely it matches with the statement execution coverage of each passing test case (T_P) in the test suite (T). Therefore, execution coverage of each statement against each passing and failing test case is required. Thus, the time complexity for the test suite optimization phase can be

expressed as $O(N \times T)$.

Furthermore, for our fault localization approach, the time complexity mainly depends on the computation time required for both the suspiciousness score of each program entity ($O(N \times T)$), and the suspiciousness score of its fault context ($O(N \times T) + O(N \times T_F)$), and subsequently the time required to generate the ordered list of suspiciousness ranks ($O(N \times \log(N))$).

vii. Overall observations

It can be concluded that our proposed approach, which incorporates test suite optimization, statement execution frequency, and fault context, significantly improves the accuracy of fault localization when applied to existing SBFL methods. This hypothesis is validated by the experimental results that compare the performance of our proposed approach to the traditional SBFL methods. In this study we have compared the performance of the proposed approach with three traditional SBFL methods namely Ochiai, Jaccard and DStar.

We employed four widely used metrics to assess the performance of our proposed approach in comparison to the traditional SBFL methods, Ochiai, Jaccard and DStar. Our findings indicate that our approach/ framework outperforms existing SBFL methods. That means, the results illustrate a notable improvement in fault localization performance when integrating our proposed approach with existing SBFL methods.

Subsections II-B, II-C, and II-D demonstrate that the concepts of test suite optimization, statement execution frequency, and fault context, respectively, can overcome certain inherent limitations of SBFL when applied individually to existing SBFL methods.

Our hybrid approach, as explained in Section III, capitalizes on these concepts by integrating them to enhance the efficiency and accuracy of SBFL from a single fault perspective. The experimental outcomes validate the superiority of our proposed approach over the current baseline SBFL methods. Ochiai* demonstrates an average enhancement of 62.76% in terms of EXAM score on Siemens suite subjects, while Jaccard* and DStar* show average improvements of 53.19% and 54.36%, respectively. Likewise, for large real-world subject programs, the average improvements are 65.23% for Ochiai*, 65.66% for Jaccard*, and 68.96% for DStar*.

V. Related Work

In the following section, we will explore the existing literature on spectrum-based fault localization. This area has been widely researched and experimented in software engineering, and many literature reviews have provided an overview of the current advanced research in this field. In recent years, numerous studies have been carried out to enhance the accuracy and effectiveness of SBFL (spectrum-based fault localization).

In their study, Wong et al. [1] presented a comprehensive survey of various techniques relevant to software fault localization. The authors addressed relevant issues and concerns that are significant to this field, providing readers with necessary background knowledge and enabling the application of fault localization techniques that are efficient and scalable in terms of time and space complexity. Their study thoroughly examines different issues and concerns associated with various software fault localization techniques.

Lei et al. [19] found that there is no significant correlation between the size of the test suite and the accuracy of fault localization in a program under testing and debugging. Furthermore, they analyzed how different segments of test suites can positively or negatively affect fault localization performance.

In their work, Perez et al. [20] introduced a new metric called DDU that aims to enhance adequacy measurements by evaluating the diagnosability of a test suite. The term diagnosability refers to the effectiveness of utilizing spectrum-based fault localization to identify faults in the code when test failures occur. The authors claim that a diverse test suite that exercises multiple combinations of components is more comprehensive than one that solely aims to maximize code coverage.

In their study, Inozemtseva et al. [21] demonstrated that there is a weak correlation between the coverage of a test suite and the effectiveness of fault detection. Their analysis indicated that the relationship between test suite coverage and its effectiveness is only moderate to low.

In their work, Xuan et al. [22] introduced a new idea called "spectrum-driven test case purification" to enhance fault localization. The primary objective of this approach is to divide the current test cases into smaller subsets, known as purified test cases, and refine the test oracles to better identify faults. By integrating this technique with an existing fault localization method (such as Tarantula), the program statements can be ranked more accurately.

In their research, Zakari et al. [29] presented a fault localization approach called FLCN-S, which employs complex network theory to enhance the effectiveness of localizing faults in single-fault programs. This technique evaluates and prioritizes potentially faulty program statements by assessing their anomalous behavior and proximity to one another during failed test executions, using two network centrality measures (degree centrality and closeness centrality).

Ju et al. [30] created a fault localization framework that uses a multivariate logistic regression model. The model incorporates both static and dynamic features collected from the program being debugged.

Roychowdhury et al. [31] discovered in their study that the principles of machine learning's feature selection can be effectively utilized in fault localization. Their experiments showed that the lines of code with the most varied feature information can pinpoint the most dubious statements.

In their study, Shu et al. [32] introduced FLSF, which improves upon traditional fault localization techniques by considering statement execution frequency, resulting in greater reliability and effectiveness compared to Tarantula, particularly for faults within loop bodies or iteration statements, as per their experiments.

In their study, Sarhan et al. [33] introduced a novel SBFL formula that tackles the challenge of ties by giving weight to the significant number of failing test cases and the low number of passing ones for a specific code element. This approach enables straightforward handling of common tie

scenarios. Empirical findings indicate that the suggested formula outperforms three extensively researched SBFL formulas in terms of average ranking.

To improve the performance of bug localization, Lee et al. [34] integrated supplementary data into SBFL. They leveraged the statement execution count data obtained from program execution and transformed it into program spectrum properties using a weighted function. By evaluating different SBFL techniques, they discovered that incorporating this information resulted in a substantial enhancement in bug localization performance in comparison to the conventional approach, which only considers whether a statement was executed or not during the test execution (i.e., binary information).

Lun et al. [35] conducted research in the field of software architecture testing, a critical aspect for ensuring software quality and dependability. They indicated that achieving component path coverage stands as a significant benchmark for evaluating the adequacy of software architecture testing. They introduced two distinct criteria for component path coverage, one based on node coverage and the other on edge coverage. These coverage criteria were found to be effective in detecting various types of faults within software systems. The researchers proposed two algorithms for quantifying the component path coverage rates based on these criteria. Enhanced interaction among components resulted in higher path coverage, thus improving the diagnostic capability of the test suite for identifying faults. Empirical findings from their study highlighted that the proposed component path coverage criteria offer a robust framework for practical software architecture testing and lay a strong foundation for future research in this domain.

Oo et al. [36] suggested a mutation-based testing technique to fix object-oriented program errors. This method identifies the correct repair code within the search space using the faulty statement's type and the MuJava mutation system. Initially, program faults are detected by assessing the suspiciousness of statements. A two-level mutation system modifies the code, and similar candidates to the faulty statement type are collected. An ordered list of potential patches is tested one by one using a test suite until a valid fix is found. The approach was evaluated using the Defects4J dataset.

In their survey paper, Wu et al. [37] outlined the process flow for testing the artificially intelligent systems. Their study offers an extensive overview of methods used to isolate faulty behavior in intelligent systems. It summarizes techniques related to testing coverage metrics, test data generation, testing approaches, formal verification methods, and widely used datasets.

Alakeel [38] presented an automated approach to detect and fix test dependencies in web application test suites. The technique employs data flow analysis to identify noticeable test dependencies responsible for test failures and subsequently automates the repair process for the issues arising from these dependencies.

Setiadi et al. [39] introduced an algorithm designed to test concurrent programs efficiently by minimizing the number of test cases for fault identification. This is achieved by generating test cases through the analysis of execution traces, incorporating different interleaving. Redundant test cases are pruned while maintaining fault detection accuracy. The algorithm makes use of branch structures and data flows extracted from execution traces to isolate only those interleaving that impact branch outcomes. The effectiveness of the proposed approach was assessed using a suite of JAVA-based concurrent programs.

Integration testing is significant in identifying errors that emerge between class interfaces. Given the potentially numerous interfaces within object-oriented software's individual classes, conducting testing can prove costly. Laokok et al. [40] introduced an approach to generate test cases for integration testing by analyzing the static call graph. This technique ensures that the produced test cases traverse all branches within the static call graph at least once. By extracting data from the source code and constructing a static call graph that encompasses all class interfaces, the test cases achieve complete coverage, encompassing branch scenarios.

VI. Threats to Validity

The potential bias of the experimenters poses an internal validity threat to our proposed approach. To collect program execution traces, we used manually instrumented programs and ran them with the instrumentation. However, there is a risk of bias due to experimenter negligence, such as the inadvertent skipping or misplacement of certain program blocks during the instrumentation process.

The presence of external validity threats concerns the extent to which the experimental findings can be applied to subjects beyond the subject programs utilized in the study. In response to such threats we have conducted experimentation on widely used standard benchmark Siemens test suite programs and on some real-world programs which are quite larger in size.

We recognize that there is no empirical study that can be completely flawless, and that there may be complex programs and bugs that were not included in our experiments. In our future work, we intend to expand the scope of our subjects.

The issue of construct validity threats pertains to whether the performance metrics employed in empirical studies accurately depict the real-life scenario. The first threat relates to the suitability of evaluation metrics used in the empirical study. Considering this threat, our study employs four widely recognized metrics, namely Exam score, Cumulative Number of Statements Examined, Top-N, and Wilcoxon signed-rank test, to evaluate the effectiveness of our proposed approach in comparison to the traditional SBFL approaches. To compare our proposed approach with the classical SBFL methods, we used the EXAM score metric to calculate the improvement in the absolute rank of faulty program entities. Furthermore, we examined the suitability of the baseline approach in terms of stability and efficiency, considering the potential threat posed by its use as a comparison point for our proposed approach. Although we have used three popular and most studied approaches i.e. Ochiai, Jaccard and DStar as a benchmark for comparison with the proposed approach, in future we also intend to use other similarity coefficient metrics for a more comprehensive comparative analysis.

VII. Conclusion

In this paper, we have introduced a hybrid approach that aims to improve the accuracy and efficiency of spectrum-based software fault localization in single fault scenarios. The approach consists of the following major steps. Firstly, the test suite to be used in the fault localization process is optimized using passing test discrimination based approach. In the second step, the execution count of each statement is calculated with respect to each test case in the test suite. A frequency weighting function, specifically an adapted sigmoid function, is used to transform the statement execution frequency count to a normalized real value within the range of 0 and 1. Then, in the third step, the suspiciousness of each statement is calculated using one of the SBFL similarity coefficient metric (e.g., Ochiai, Jaccard, DStar, etc.). Here, the SBFL formula utilizes the normalized frequency count, instead of the binary coverage information (0 or 1) for the calculation of suspiciousness scores. Next, fault context for each program statement is generated in each failed execution, and then its suspiciousness score is calculated. Finally, a new improved fault rank list is generated based on the suspiciousness of a statement and its fault context.

The study conducted an empirical analysis on two standard benchmarks, namely the Siemens test suite and large real-world subject programs (flex, grep, sed, and space). The experimental results revealed that the proposed approach/ framework outperforms the traditional SBFL techniques in single fault perspective. The experimental results validate the effectiveness of our proposed approach compared to the existing baseline SBFL methods. Specifically, the Ochiai* technique demonstrates an average improvement of 62.76% in terms of EXAM score across Siemens suite subjects. Similarly, the Jaccard* and DStar* methods exhibit average improvements of 53.19% and 54.36%, respectively. Likewise, when examining large real-world programs, the average improvements are 65.23% for Ochiai*, 65.66% for Jaccard*, and 68.96% for DStar*. Here, the term ‘improvement’, refers to the average reduction in the number of examined statements (developer's effort) by the specified percentage. For example, the Ochiai* method achieving a 62.76% improvement implies that developers, on average, need to examine 62.76% fewer statements to locate faults across the faulty versions of Siemens programs utilized in the experimental study.

Our approach considerably reduces the developer’s effort in locating the faults, for example, our approach when applied to classic Ochiai (denoted as Ochiai*), identifies 52% of the faults in Siemens and 60% of the faults in large real-world programs by examining less than or equal to 1% and 0.1% of the code, respectively. In contrast, the classic Ochiai method is unable to locate any of the faults by examining the same percentages of codes.

Furthermore, in terms of the Top-N evaluation metric, applying our approach to existing Ochiai, Jaccard, and DStar methods on Siemens programs leads to fault identification percentages of 60%, 44%, and 44%, respectively within the Top-5 positions. In contrast, without applying our approach, these methods only manage to locate 8%, 4%, and 4% of faults in the same scenario. Similarly, for large real-world programs, our approach applied to existing Ochiai, Jaccard, and DStar methods results in fault identification percentages of 27%, 20%, and 20%, respectively within the Top-5 positions. In contrast, without applying our approach, these traditional SBFL methods fail to identify any faults at the Top-5 position.

We also performed statistical tests (Wilcoxon signed-rank test) to justify the significance of the improved fault localization performance achieved with our proposed approach, in comparison to traditional SBFL techniques. These results indicate that our proposed approach achieves a substantial improvement over the classical SBFL approaches, and therefore can effectively improve the performance and accuracy of existing SBFL techniques.

Our research outcomes provide a new perspective on fault localization and open up interesting directions for future exploration. In future work, we intend to extend our approach to locate multiple faults. Additionally, we plan to investigate the efficacy of our technique on diverse subject programs written in various programming languages, such as Java and Python, in order to provide further evidence for our assertions. Further research is also necessary to establish a fault-context structure that facilitates a deeper comprehension of the underlying cause of failure, ultimately enhancing the accuracy of fault ranking.

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” IEEE Trans. Softw. Eng., vol. 42, no. 8, pp. 707–740, 2016.

[2] Amol Saxena, Roheet Bhatnagar, and Devesh Kumar Srivastava, "Software Fault Localization: Techniques, Issues and Remedies," IAENG International Journal of Computer Science, vol. 49, no.2, pp299-317, 2022.

[3] H. A. de Souza, M. L. Chaim, and F. Kon, “Spectrum-based software Fault Localization: A survey of techniques, advances, and challenges,” arXiv [cs.SE], 2016.

[4] H. Cleve and A. Zeller, “Locating causes of program failures,” in Proceedings of the 27th international conference on Software engineering - ICSE ’05, 2005, pp.342-351.

[5] R. Gao and W. E. Wong, “MSeer—an advanced technique for locating multiple bugs in parallel,” IEEE Trans. Softw. Eng., vol. 45, no. 3, pp. 301–318, 2019.

[6] A. S. Namin, "Statistical Fault Localization Based on Importance Sampling," 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 2015, pp. 58-63, doi: 10.1109/ICMLA.2015.91.

[7] S. Parsa, M. Vahidi-Asl, and M. Asadi-Aghbolaghi, “Hierarchy-Debug: a scalable statistical technique for fault localization,” Softw. Qual. J., vol. 22, no. 3, pp. 427–466, 2014.

[8] F. Wotawa, M. Nica, and I. Moraru, “Automated debugging based on a constraint model of the program and a test case,” J. Log. Algebr. Program, vol. 81, no. 4, pp. 390–407, 2012.

[9] W. Zheng, D. Hu, and J. Wang, “Fault localization analysis based on deep neural network,” Math. Probl. Eng., vol. 2016, pp. 1–11, 2016.

[10] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, “Effective software fault localization using an RBF neural network,”IEEE Trans. Reliab., vol. 61, no. 1, pp. 149–169, 2012.

[11] R. Abreu, P. Zoeteweij and A. J. c. Van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06), Riverside, CA, USA, 2006, pp. 39-46, doi: 10.1109/PRDC.2006.18.

[12] J. A. Jones and M. J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” in Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 273–282.

[13] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer, "Pinpoint: problem determination in large, dynamic Internet services," Proceedings International Conference on Dependable Systems and

Networks, Washington, DC, USA, 2002, pp. 595-604, doi: 10.1109/DSN.2002.1029005.
[14] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," IEEE Trans. Reliab., vol. 63, no. 1, pp. 290–308, 2014.
[15] R. Abreu, P. Zoeteweij and A. J. C. v. Gemund, "Localizing Software Faults Simultaneously," 2009 Ninth International Conference on Quality Software, Jeju, Korea (South), 2009, pp. 367-376, doi: 10.1109/QSIC.2009.55.
[16] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Simultaneous debugging of software faults," J. Syst. Softw., vol. 84, no. 4, pp. 573–586, 2011.
[17] E. Wong, T. Wei, Y. Qi and L. Zhao, "A Crosstab-based Statistical Method for Effective Fault Localization," 2008 1st International Conference on Software Testing, Verification, and Validation, Lillehammer, Norway, 2008, pp. 42-51, doi: 10.1109/ICST.2008.65.
[18] A. Saxena, R. Bhatnagar and D. K. Srivastava, "Improving Effectiveness of Spectrum-based Software Fault Localization using Mutation Testing," 2021 2nd International Conference for Emerging Technology (INCET), Belagavi, India, 2021, pp. 1-7, doi: 10.1109/INCET51464.2021.9456109.
[19] Y. Lei, C. Sun, X. Mao, and Z. Su, "How test suites impact fault localisation starting from the size," IET Softw., vol. 12, no. 3, pp. 190–205, 2018.
[20] A. Perez, R. Abreu and A. van Deursen, "A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches," 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 2017, pp. 654-664, doi: 10.1109/ICSE.2017.66.
[21] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 435-445, https://doi.org/10.1145/2568225.2568271
[22] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp.52-63 https://doi.org/10.1145/2635868.2635906
[23] A. Saxena, R. Bhatnagar, and D. Kumar Srivastava, "Effective lightweight software fault localization based on test suite optimization," in 15th Innovations in Software Engineering Conference,2022, pp.1–10. https://doi.org/10.1145/3511430.3511437
[24] T.-D. B. Le, D. Lo, and F. Thung, "Should I follow this fault localization tool's output?: Automated prediction of fault localization effectiveness," Empir. Softw. Eng., vol. 20, no. 5, pp. 1237–1274, 2015.
[25] Y. Wang, Z. Huang, Y. Li, and B. Fang, "Lightweight fault localization combined with fault context to improve fault absolute rank," Sci. China Inf. Sci., vol. 60, no. 9, 2017.
[26] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," Empir. Softw. Eng., vol. 10, no. 4, pp. 405–435, 2005.
[27] R. Abreu, P. Zoeteweij and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), Windsor, UK, 2007, pp. 89-98, doi: 10.1109/TAIC.PART.2007.13.
[28] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using PageRank," in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis,2017,pp. 261-272, https://doi.org/10.1145/3092703.3092731
[29] A. Zakari, S. P. Lee, and I. A. T. Hashem, "A single fault localization technique based on failed test input," Array (N. Y.), vol. 3–4, no. 100008, p. 100008, 2019.
[30] X. Ju, J. Qian, Z. Chen, C. Zhao, and J. Qian, "Mulr4FL: Effective fault localization of evolution software based on multivariate logistic regression model," IEEE Access, vol. 8, pp. 207858–207870, 2020.
[31] S. Roychowdhury and S. Khurshid, "Software fault localization using feature selection," in Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering, 2011, pp 11-18, https://doi.org/10.1145/2070821.2070823
[32] T. Shu, T. Ye, Z. Ding, and J. Xia, "Fault localization based on statement frequency," Inf. Sci. (Ny), vol. 360, pp. 43–56, 2016.
[33] Q. I. Sarhan and Á. Beszédes, "A Survey of Challenges in Spectrum-Based Software Fault Localization," in IEEE Access, vol. 10, pp. 10618-10639, 2022, doi: 10.1109/ACCESS.2022.3144079.
[34] H. J. Lee, L. Naish and K. Ramamohanarao, "Effective Software Bug Localization Using Spectral Frequency Weighting Function," 2010 IEEE 34th Annual Computer Software and Applications Conference, Seoul, Korea (South), 2010, pp. 218-227, doi: 10.1109/COMPSAC.2010.26.
[35] Lijun Lun, Xin Chi, and Hui Xu, "Coverage Criteria for Component Path-oriented in Software Architecture," Engineering Letters, vol. 27, no.1, pp40-52, 2019.
[36] Cherry Oo, and Hnin Min Oo, "Automatic Program Repair of Java Single Bugs using Two-level Mutation Operators," IAENG International Journal of Computer Science, vol. 47, no.2, pp223-233, 2020.
[37] Tingting Wu, Yunwei Dong, Zhiwei Dong, Aziz Singa, Xiong Chen, and Yu Zhang, "Testing Artificial Intelligence System Towards Safety and Robustness: State of the Art," IAENG International Journal of Computer Science, vol. 47, no.3, pp449-462, 2020.
[38] Ali M. Alakeel, "Dependency Detection and Repair in Web Application Tests," IAENG International Journal of Computer Science, vol. 49, no.2, pp369-384, 2022.
[39] Theodorus E. Setiadi, Akihiko Ohsuga, and Mamoru Maekawa, "Efficient Execution Path Exploration for Detecting Races in Concurrent Programs," IAENG International Journal of Computer Science, vol. 40, no. 3, pp143-163, 2013.
[40] Sitdhibong Laokok, and Taratip Suwannasart, "An Approach for Test Case Generation from a Static Call Graph for Object-Oriented Programming," Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2017, 15-17 March, 2017, Hong Kong, pp545-549.

Amol Saxena is a PhD candidate of Department of Computer Sc. & Engineering, SCSE, Manipal University Jaipur, Jaipur, Rajasthan, India (303007). He possesses a master's degree in computer science and engineering, along with a degree of master of computer applications. He is working as an Assistant Professor at Department of Information Technology, Poornima College of Engineering, Jaipur, Rajasthan, India (302022). His research focus is on automated software engineering, software reliability, software fault localization, software testing & debugging and data mining. His other areas of interest include databases, big data analytics, artificial intelligence, software engineering, object oriented technologies, computer networks and operating systems.

Dr. Roheet Bhatnagar is a Professor of Computer Sc. & Engineering Department, SCSE, and Director, Directorate of Research, Manipal University Jaipur, Jaipur, Rajasthan, India (303007). He received PhD in Computer Sc. & Engineering from Sikkim Manipal University, Sikkim, India in 2011. He has a total 26 years of experience in the areas of teaching, research and administration. He is associated with Manipal University Jaipur since 2012 and with Manipal Group since 2008. His areas of interests include Database, Data Mining, Big Data Analytics, Soft Computing, Machine Learning, Software Engineering, GIS, and Remote Sensing.

He has published or co-authored more than 80 research papers in various indexed journals and conferences of International repute. Additionally, he has chaired several technical sessions in various prestigious national and international conferences and is a reviewer of a number of reputed international journals.

Dr. Bhatnagar is a senior member of IEEE, life members of Indian Society of Technical Education and Indian Society of Remote Sensing, and member of Computer Society of India.

Dr. Devesh Kumar Srivastava is a Professor at Department of Information Technology, SIT, Manipal University Jaipur, Jaipur, Rajasthan, India (303007). He received PhD in Software Engineering from Uttarakhand Technical University, Dehradun, India in 2012. He has a total 23 years of experience in the areas of teaching, research and administration. He is associated with Manipal University Jaipur since August-2012 where he was promoted as a Professor in August 2016. His areas of interests include Software Engineering, Data Mining, Machine Learning and Cloud Computing.

He has contributed to over 100 research papers published in Scopus/ Web of Science/ SCI indexed Journals and conferences proceedings. Furthermore, he has chaired 42 technical sessions in various prominent national and international conferences. He was invited as keynote speaker in FDPs and in the conferences. He is a reviewer of several reputed international journals. He supervised six PhD scholars, many PG/ UG scholars for their project.

Dr. Srivastava is a professional members of IEEE, IAENG (Software Engineering) and senior member of ACM.