Journal of Innovations in Computer Science and Engineering
Shahid Beheshti Unversity

June 2025, Volume 3, Issue 1

Improving the Accuracy of Spectrum-Based Fault Localization Techniques by Focusing on the Program's Changes and Dependencies

Faeze aghazade-par, ORCID ID: 0009-0003-5530-1310
Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran,
f_aghazadepar@sbu.ac.ir
MOJTABA VAHIDI-ASL✉, **CODE ORCID**: 0000-0003-4964-992X
Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran,
mo_vahidi@sbu.ac.ir

ABSTRACT
Spectrum-Based Fault Localization (SBFL) techniques are widely used and studied. These techniques are straightforward, low-cost, and fast in comparison to other fault localization techniques, such as Mutation-Based and Learning-Based ones. However, the accuracy of these techniques is always controversial. The main criticism of these techniques refers to the mere use of coverage and spectra information. Hence, this study seeks to improve SBFL techniques by enhancing their accuracy while maintaining simplicity, aiming to make them applicable for regression faults. To achieve this, the first step involves integrating SBFL techniques with two features: (1) Change and (2) Test Case Weight, which utilizes the same coverage and spectrum data in a novel way. Additionally, the study examines the impact of incorporating Data and Control Dependency into the formula for fault localization accuracy. It also considers the suspiciousness scores generated by SBFL formulae as a feature.
Three SBFL techniques—Tarantula, Ochiai, and Jaccard—are applied in this study both as a feature and as a baseline for result comparison. The findings reveal that combining SBFL suspiciousness scores with Change and Test Case Weight significantly enhances performance by identifying at least 27% more faults at the Top-3 ranking. Furthermore, integrating these features with Data Dependency leads to even greater improvements, locating minimum 45% more faults at the Top-3 ranking compared to aforementioned SBFL formulae. Overall, this study emphasizes the limitations of existing SBFL formulae while highlighting the advantages of augmenting SBFL with additional features and repurposing spectral information to achieve greater accuracy in fault localization.

KEYWORDS
Spectrum-Based Fault Localization, Spectra Information, Versioning, Data Dependency, Control Dependency, Test Cases

1. INTRODUCTION

Regardless of the extent of effort invested in their development, software systems are often prone to faults and defects. A key factor contributing to these issues is the inherent complexity of the software. This challenge has become increasingly significant with the advent and widespread adoption of complex and sophisticated systems [1]. Addressing software faults requires accurately pinpointing their locations as an essential preliminary step before initiating the repair procedures [2, 3]. However, fault localization is a notably challenging, labor-intensive, and intricate process, accounting for approximately 54% of the time and effort expended by software developers during the development lifecycle [4, 5, 6, 7].
Hence, automated software fault localization techniques have been developed to mitigate the challenges associated with debugging by reducing costs and enhancing efficiency. Fault localization, a prominent area of research in software engineering, is defined as "identifying locations in a program's source code that are implicated in some observed failures (such as crashes or other kinds of runtime errors) as a key step of debugging" [8]. These techniques streamline the fault localization process, enabling developers to focus their efforts more effectively on resolving faults and improving software quality [9]. Fault localization methods have been categorized in diverse ways across various studies. For instance, [10] classifies them into four main types: slice-based, spectrum-based, learning-based, and mutation-based techniques. Similarly, [11] groups these methodologies into state-based, slice-based, statistical, machine learning-based, and spectrum-based categories. Another classification introduced by [12] includes slice-based, model-based, predicate-based, and spectrum-based approaches.
Additionally, [13] organizes fault localization techniques into spectrum-based, mutation-based, dynamic slicing, stack trace analysis, prediction switching, information retrieval-based, and history-based categories. Expanding upon these categorizations, [2] incorporates spectrum-based, slicing-based, state-based, statistical, machine learning-based, model-based, and mutation-based methods, among others. Furthermore, [14] delineates fault localization methodologies into three principal groups: spectrum-based fault localization (SBFL), mutation-based fault localization (MBFL), and fault localization techniques leveraging machine learning (ML) and deep learning (DL). These varied classifications underscore the dynamic and evolving nature of fault localization approaches, offering multiple perspectives on effectively addressing software faults.

Submit Date: 2025-11-23
Revise Date: 2026-02-09
Accept Date: 2026-04-28
✉ Corresponding author

Among the various fault localization techniques, Spectrum-Based Fault Localization (SBFL) stands out as a widely adopted and highly effective approach. This dynamic method leverages two key types of information: test case execution results and the program spectrum. Test case execution results are systematically recorded to determine whether each test has succeeded or failed, with failures indicating the presence of faults in the program. The program spectrum, on the other hand, provides detailed code coverage information for each test case [2]. SBFL operates by analyzing the statements of the code executed during test cases and assigning a suspiciousness score to each statement. These scores are determined based on the frequency of a statement's occurrence in both successful and failed test executions.

Spectrum-based techniques are highly valued for their simplicity and efficiency, making them especially well-suited for large-scale and real-world software systems [11]. They are particularly effective in identifying faults in single-fault programs, showcasing strong performance in such scenarios [15]. Consequently, numerous studies have proposed various formulae and methodologies based on SBFL, many of which have achieved significant recognition. Among these, approaches such as *Tarantula* [16], [17], *Jaccard* [18], *Ochiai*, *D-Star* [19], *Barinel*, and *OP2* have emerged as some of the most widely used and effective methods, highlighting their prominence in fault localization research.

While spectrum-based techniques have earned widespread recognition for their simplicity and efficiency, they are subject to significant critiques. Many current Spectrum-Based Fault Localization (SBFL) methods fail to account for two crucial aspects: (1) the varying contributions of failed test cases to identifying specific faults and (2) the collaborative interactions among program elements that differently influence the success or failure of the test cases [6]. Overlooking these dimensions limits the ability of SBFL methods to accurately pinpoint faults and grasp the nuanced behavior of the test cases. A further limitation lies in the occurrence of ties, where multiple statements receive identical suspiciousness scores [20, 10, 4, 13, 2, 15]. This problem stems from the presence of certain statements, such as conditional ones, in all test case executions, whether successful or unsuccessful, making it difficult to discern between faulty and non-faulty statements. Moreover, SBFL relies exclusively on the test case coverage and execution results [21, 22, 23, 24, 25], restricting its capacity to identify the root cause of faults [21, 4].

In summary, SBFL's shortcomings include its relatively low accuracy [13, 24, 26], its inability to consider program structure [23], inefficiency in managing large-scale software systems [24], and its weak performance in multi-fault environments [27, 28]. Furthermore, SBFL methods are unable to interpret the behavior of statements close to one another [29]. These limitations underscore the necessity for advanced fault localization techniques that effectively address these challenges while enhancing accuracy and adaptability.

Consequently, this study aims to: (1) enhance SBFL techniques by incorporating additional information into the formula and (2) improve the accuracy of SBFL formulae for locating faults in different software versions. The proposed method, referred to as "*SBCT*", incorporates three primary features: the SBFL Formula's Score, Change, and Test Case Weight. The SBFL Formula's Score represents the suspiciousness score of a statement derived from the SBFL formula. Given the effectiveness and value of SBFL techniques, this feature provides critical data for fault localization. The Test Case Weight, introduced in our previous studies [30, 31], has been selected due to its simplicity, as it does not require a separate process for value extraction. It utilizes test case results and coverage information for each statement, similar to SBFL techniques, while accounting for how statements are covered during test case executions. In essence, the Test Case Weight differentiates statements based on the specific test cases that cover them.

Additionally, the Change feature is included to account for software modifications and assess the applicability of SBFL in identifying regression faults. It is worth noting that these two features, Change and Test Case Weight, have been demonstrated to be effective in fault localization [30, 31]. Furthermore, dynamic data and control dependencies, suggested by prior research for fault localization, are incorporated into *SBCT*. These additional features are evaluated to determine their potential for enhancing the accuracy of SBFL techniques.

In this method, the program is instrumented and executed against the test cases, simultaneously generating a coverage matrix for SBFL formula calculations. Following the SBFL routine, suspiciousness scores for the statements are computed after all test cases are executed, providing the SBFL Formula's Score for each statement. For every test case execution, the values for Test Case Weight are calculated using the coverage matrix. Subsequently, the Test Case Weight values from all test case executions are aggregated to obtain the final value of this feature for each statement. Finally, Change values are assigned to the statements, with a value of 1 if the statement has been modified and zero otherwise.

These features were then utilized to formulate a new method (*SBCT*) for fault localization. To evaluate the formula, the extracted feature values were substituted into *SBCT* to determine the suspiciousness scores for individual statements. For assessment, codes with at least three versions were selected from the *Code4Bench* benchmark dataset [32]. The results demonstrated that *SBCT* significantly outperformed existing SBFL formulae in terms of *EXAM* metrics [33] and *Top-N* accuracy, where N equals 1, 3, and 5.

This study highlights the critical role of Change values and demonstrates that SBFL can be improved effectively without requiring additional external data. To encapsulate the essence of this research, the study offers the following key contributions:

- **Introducing a Unified Feature Extraction Framework:** Combining multiple features—Coverage Matrix, Test Case Weight, Change, and Dynamic Data Dependency—into a cohesive and efficient framework for fault localization.
- **Developing an Adaptable Fault Localization Formula:** Creating *SBCT* formula that not only enhances SBFL techniques for regression faults but also adapts seamlessly to different program types and sizes.
- **Minimizing Cost through Feature Efficiency:** Demonstrating that effective fault localization can be achieved by utilizing easily collectible and low-cost features (Test Case Weight and Change) without relying on external data sources.
- **Empirical Validation on Real-World Programs:** Offering robust evaluations on diverse real-world programs with varying sizes and complexities, thereby establishing broad applicability and reliability.
- **Revolutionizing Regression Fault Handling:** Demonstrating the utility of Change in locating regression faults and emphasizing its role in future fault localization advancements.

The structure of this paper is organized as follows: Section 2 presents an illustrative example to elucidate the core concept of the proposed methodology. Section 3 introduces fundamental concepts of the research. Section 4 offers an extensive review of related literature. Section 5 provides a detailed explanation of the proposed approach. Section 6 describes the evaluation framework and experimental design. Section 7 reports the evaluation results. Finally, the concluding sections focus on an in-depth discussion of the results and outline the overarching conclusions drawn from the study.

2. MOTIVATIONAL EXAMPLE

This section provides an illustrative example to examine the limitations of SBFL techniques and demonstrate the advantages of *SBCT*. SBFL is a widely utilized technique that relies exclusively on the spectral information of the code, calculating suspiciousness scores based on statement coverage and test case results. However, this reliance on simple information can sometimes mislead the fault localization process in specific scenarios.

To analyze SBFL performance, Table 1 presents a straightforward example involving 10 test cases, showcasing both the coverage matrix and SBFL results. The three leftmost columns are dedicated to three commonly used SBFL formulae—*Tarantula*, *Ochiai*, and *Jaccard*. These columns detail the suspiciousness scores assigned to each statement. Based on these scores, statements are sorted and ranked according to their likelihood of being faulty. The formulae for *Tarantula*, *Ochiai*, and *Jaccard* are defined using specific notations to clarify their functionality and application within the context of SBFL:

N_{cf}: Number of failed runs in which the statement is covered
N_{uf}: Number of failed runs in which the statement is not covered
N_{cp}: Number of passed runs in which the statement is covered
N_{up}: Number of passed runs in which the statement is not covered

$$\text{suspiciousness score}_{(\text{Tarantula})} = (N_{cf}/(N_{cf}+N_{uf})) \,/\, ((N_{cf}/(N_{cf}+N_{uf})) + (N_{cp}/(N_{cp}+N_{up}))) \quad [16]\ (1)$$

$$\text{suspiciousness score}_{(\text{Ochiai})} = N_{cf}/\sqrt{(N_{cf}+N_{uf}) \cdot (N_{cf}+N_{cp})} \quad [18]\ (2)$$

$$\text{suspiciousness score}_{(\text{Jaccard})} = N_{cf}/(N_{cf}+N_{uf}+N_{cp}) \quad [18]\ (3)$$

Table 1. Example of SBFL formulae

Statements of Code	*Ncf*	*Nuf*	*Ncp*	*Nup*	Tarantula	Ochiai	Jaccard
num = int(input())	6	0	4	0	0.5	0.301511	0.6
factorial = 0 //fault	6	0	4	0	0.5	0.301511	0.6
if num < 0:	6	0	4	0	0.5	0.301511	0.6
print("Sorry, factorial does not exist for negative numbers")	0	6	3	1	0	0	0
elif num == 0:	6	0	1	3	0.8	0.353553	0.857143
print("The factorial of 0 is 1")	0	6	1	3	0	0	0
else:	6	0	0	4	1	0.377964	1
for i in range(1,num + 1):	6	0	0	4	1	0.377964	1
factorial = factorial*i	6	0	0	4	1	0.377964	1
print("The factorial of", num, "is", factorial)	6	0	0	4	1	0.377964	1

As shown in Table 1, these formulae struggle to effectively differentiate between statements, often assigning identical suspiciousness scores to multiple statements. This limitation compromises the accuracy of fault localization, as the faulty statement becomes indistinguishable from two other statements with identical scores. Notably, in all three formulae, the faulty statement is consistently ranked as the 7th most suspicious, yielding an *EXAM* score of 70%.

Following this analysis, Table 2 introduces *SBCT*, utilizing the same code and test cases as Table 1 for direct comparison. To enhance clarity and facilitate better comprehension, Table 2 employs the following notations for improved readability.

D1: Data dependency (*def*)
D2: Data dependency (*use*)
CH: Change
TCW: Test Case Weight

Table 2. Example of *SBCT*

Statements of Code	Jaccard Score	D1	D2	CH	TCW	SBCT score
num = int(input())	0.6	20	0	0	5	-2.6338
factorial = 0 //fault	0.6	5	0	1	5	-1.2358
if num < 0:	0.6	0	5	0	5	-2.7928
print("Sorry, factorial does not exist for negative numbers")	0	0	0	0	0	-2.785
elif num == 0:	0.857143	0	5	0	5	-2.9144
print("The factorial of 0 is 1")	0	0	0	0	0	-2.785
else:	1	0	0	0	5	-2.983
for i in range(1,num+1):	1	10	10	1	5	-1.383
factorial = factorial*i	1	9	14	0	5	-2.9082
print("The factorial of", num, "is", factorial)	1	0	10	0	5	-2.981

Table 2 highlights two critical differences when compared to Table 1. First, the suspiciousness scores across statements show apparent variation, underscoring the effectiveness of the proposed features in distinguishing between different statements. Second, the "PM Score" column demonstrates the superior performance of *SBCT*, enabling more efficient fault localization compared to SBFL techniques. Notably, based on the "SBCT Score," the faulty statement is accurately identified as the most suspicious one, with the highest score assigned. In the subsequent sections, a comprehensive explanation of *SBCT*'s formula and the methodology utilized for computing suspiciousness scores is provided to detail the process and validate its effectiveness.

3. BACKGROUND

3.1 FAULT LOCALIZATION

In order to improve software quality, it is essential to identify faults by software testing and locate them for repair [3]. Fault localization is a process for identifying the location of the fault, which begins after software testing and following the failure of the test execution result [2]. This process identifies specific elements of the program that are responsible for the program fault [5]. Given the importance of fault localization, numerous studies have been conducted in this field; and each provides a different classification of existing techniques. According to [28], there are three categories of fault localization techniques, which are trace-based debugging, delta debugging, and coverage-based fault localization. Another research has presented a different classification. According to [10], there are four categories of techniques for fault localization, including slicing-based, spectrum-based, learning-based, and mutation-based. Another classification is provided in [12] for fault localization techniques, which are slicing-based, model-based, predicate-based, and spectrum-based. Furthermore, in another classification by [13], fault localization techniques are divided into spectrum-based, mutation-based, dynamic slicing-based, stack trace analysis, prediction switching, information retrieval-based, and history-based. In addition, the classification provided by [2] considers fault localization techniques to include spectrum-based, slicing-based, state-based, statistics-based, machine learning-based, model-based, mutation-based, etc. All in all, the presented categories show that fault localization approaches are mainly categorized into three groups: mutation-based, spectrum-based, and learning-based approaches. Moreover, with recent advances in artificial intelligence and neural networks, learning-based approaches have accounted for the majority of recent research.

3.2 SPECTRUM-BASED FAULT LOCALIZATION

Spectrum-based fault localization (SBFL) is one of the most widely used and popular techniques in the field of fault localization. Spectrum-based fault localization is a dynamic technique that operates on two types of information: test execution results and program spectrum. In this method, test execution information, including its execution result, is stored. The test execution result can be successful or unsuccessful. Failure of the test indicates the presence of a fault in the program. The program spectrum refers to the code coverage information by each test case [10].

In detail, it can be said that in spectrum-based techniques, an instrumented version of the faulty program is executed using a test suite. Then, for each execution, a coverage matrix is generated to record the coverage of the code by the test case and the test case result. According to this matrix, it is determined how many code sections (statements/branches/functions) were present in faulty executions and vice versa. Subsequently, a SBFL formula is used to calculate the suspiciousness score for each section in the matrix, and rank the program entities [11]. The rule is that the more a given code entity is present in failed executions and fewer successful executions, the more likely it is to be faulty [13]. Many studies consider spectrum-based techniques to be lightweight and simple, which makes them more applicable to large and real-world applications [11].

3.3 MUTATION-BASED FAULT LOCALIZATION

In the mutation-based techniques, syntactic changes are made to the program under test, and then different versions of the program are generated that include unrealistic faults. These versions are then run with the existing tests [2]. Mutation-based fault localization extends the code coverage information by changing the statements with the changed versions of the program, called mutants, and then collects the new coverage information obtained by running the mutants and calculates the suspiciousness score of the statements [13]. In the mutation-based technique, the faulty program is changed regularly and re-run with the help of the existing tests with each change. Suppose that statement S is changed. If a larger number of test cases succeed as a result of changing S, it can be concluded that S was faulty. This technique is superior in accuracy to spectrum-based localization [2]. It also operates finer-grained than spectrum-based techniques. In other words, it measures the effect of each statement in the code [22]. In addition, this method is very efficient in simulating the behavior of real faults [2].

3.4 LEARNING-BASED FAULT LOCALIZATION

With the development of neural networks and their widespread use, machine learning and deep learning were also used in fault localization. In this method, the learning network extracts several features from the code and performs fault localization based on those features. The use of machine learning in cost and value-based techniques has shown high effectiveness. Also, experiments show that deep learning methods perform more effectively than information retrieval methods [20]. Despite the fact that learning methods are more effective than other techniques, the efficiency of these models decreases with increasing program size [29], and the accuracy of learning networks is still controversial [20]. Moreover, another criticism of this method is the need for a huge data set for learning to get high accuracy and precision.

4. RELATED WORKS

This section examines research efforts aimed at improving spectrum-based fault localization (SBFL) techniques. Various studies have proposed methodologies, features, and frameworks to address the limitations of SBFL, such as accuracy, efficiency, and adaptability, ensuring better fault localization outcomes.

Thus, *EXCEPT* is a technique developed to improve the accuracy of spectrum-based fault localization. Unlike SBFL, which relies on the relationship between executed statements and failed tests, *EXCEPT* focuses on the semantics of failures, particularly those caused by exceptions, and identifies both the faults and their locations based on the meaning of the exception and its underlying cause. The technique takes as input a faulty program, a test that failed with an uncaught exception, and a ranked list

of suspicious statements generated by SBFL. Upon execution of the program and detection of a failure, *EXCEPT* expands the list of suspicious statements by adding newly identified candidates for repair and then integrates these additions with the existing SBFL-ranked list. This process aims to enhance the accuracy and effectiveness of fault localization [34].

Additionally, an automated framework was developed using a chaos-based genetic algorithm to address the challenge of multi-fault localization. This approach builds upon the foundation of spectrum-based techniques. In this framework, program statements serve as chromosomes, while their corresponding suspiciousness scores represent the genes. The genetic algorithm is then applied, employing uniform crossover for recombination and utilizing the logistic mapping function to introduce chaos order into the process. The *D-Star* method is leveraged to compute the suspiciousness scores for each program statement, guiding the localization of faults effectively [28].

Moreover, a hybrid fault localization approach combines spectrum-based, mutation-based, and neural network-based techniques. Mutants are generated by modifying a correct program's operators and operands. Spectra and test execution results from both the faulty program and mutants are processed through two modules: one spectrum-based and the other neural network-based. These results are analyzed for similarity scores using the *RankBoost* algorithm, which merges scores from various methods to compute final suspiciousness values. Statements linked to multiple mutants are assigned the highest similarity scores, improving fault localization accuracy [35].

Similarly, *ABFL* is a fault localization approach that leverages a combination of features and ranking models for enhanced accuracy. It begins by extracting 32 features from the program's static source code through an automatic encoder. This process involves tokenizing program statements, enabling the encoder to generate fixed-length representations for each statement, thus capturing features from the source code. Additionally, 14 types of suspiciousness scores are calculated using spectrum-based techniques, which are then incorporated as an extra feature. All extracted features are subsequently integrated into ranking learning algorithms to construct a ranking model. This model is utilized to rank program statements, effectively pinpointing faulty ones. By combining spectrum-based insights with static code features, ABFL achieves an accurate fault localization [36].

Similar to the previous two, [13] merges spectrum-based and mutation-based methods to enhance fault localization accuracy. Initially, the faulty program is assessed using spectrum-based techniques, which rank potentially faulty statements by their suspiciousness scores. Following this, a mutant program is generated and executed, and the high-ranking statements from the spectrum-based evaluations are re-examined and re-ranked based on their mutation scores, refining the localization process further. Correspondingly, [37] combines spectrum-based localization with fault influence propagation analysis for enhanced fault detection. A dynamic instrumented execution tracker is developed to collect test coverage data and method call relationships concurrently. Using complex network theory, a fault influence network is constructed by abstracting the network topology from these method call relationships, with node weights determined through raw fault localization.

Besides, another research suggested a predicate switching technique, a lightweight mutation-based technique, to refine fault localization by augmenting spectrum-based methods. Initially, the spectrum-based routine is executed to obtain coverage information and a ranking list of suspicious statements. Statements with high suspiciousness scores are then selected for mutation. For each mutated statement: a) If the mutation alters the test case result, all statements likely to affect the values of this statement are marked as suspicious for the failed test case execution. b) If the mutation does not change the test case result, statements solely influencing the values of the mutated statement are identified as less suspicious for the successful test case execution. Finally, these findings are integrated with the spectrum-based formula results, enabling a refined re-ranking of statements to improve fault localization accuracy [22].

Furthermore, [38] explores a novel source of information for fault localization by integrating data flow analysis into the calculation of the suspiciousness scores. The method calculates the suspiciousness of program statements through the execution of data flow components. An evolutionary algorithm is then employed to optimize the weights derived from various fault sources, including data flow, control flow, and combination flow. Unlike traditional coverage-based approaches that focus on control flow, this method utilizes data flow and *def-use* relations to enhance fault localization accuracy. A genetic algorithm further refines the process by searching for optimal weights within the linear combination of suspiciousness values, ensuring a more accurate fault localization.

Additionally, [23] explores how program structure influences the effectiveness of spectrum-based fault localization techniques. To address the structure bias in suspiciousness calculations, a structure-aware method is proposed that incorporates debug history. The approach involves running test cases across all program versions and recording their execution traces. For each version, the suspiciousness scores of the basic blocks are calculated using an SBFL method. To refine the results, the average suspiciousness of each block across different versions is computed and subtracted from the suspiciousness of the same block in the current version. This adjustment aims to reduce structural bias and improve the accuracy of fault localization.

Similarly, a fault localization in continuous integration is achieved through the *Ochiai* spectrum-based technique by [39]. The process begins by collecting daily test results, focusing on failed tests. In cases of multiple test runs, only the latest run is considered. Coverage data and the spectrum-based method are then employed to perform fault localization for each daily test, with weekly coverage information serving as the reference. Suspiciousness scores for statements are calculated and aggregated using a max aggregation scheme, which scales the scores from the statement level to the component and file levels. Within this scheme, the highest suspiciousness score among a file's statements is assigned to the file.

Another history-based research identifies the bug-inducing commits for each test that uncovered a bug, and pinpoints the first commit where the failure occurred [4]. For every suspicious code entity, a historical spectrum is created within these bug-inducing commits, mapping the evolution of the suspicious code elements from the faulty version to the present. At this stage, history slicing is applied to refine the analysis. Suspiciousness scores for code entities are then calculated using a spectrum-based method applied to the historical spectrum. Entities that have undergone numerous changes in bug-inducing commits but minimal changes in healthy commits are deemed more likely to be the source of the bug.

To enhance the accuracy of spectrum-based fault localization, a customized metric is designed and tailored to the program's specific nature and runtime characteristics by [11]. Multiple faulty and mutated versions are generated for each program and tested using relevant data. The effectiveness of each version is evaluated using existing ranking metrics. From this evaluation, a selection of the best-performing metrics is made, which are then combined to create a new, optimized metric for improved fault localization.

Moreover, *ProFL* is a dynamic fault localization approach that refines fault prioritization by incorporating patch generation data from the *PraPR* tool. Capable of handling diverse bug types, *ProFL* begins with a faulty program and a failure test suite as inputs. Suspiciousness scores are initially calculated using spectrum-based techniques like *Ochiai* at the statement level, and scores are aggregated for each program element. A matrix of patches and their corresponding test execution results is then generated. Based on this matrix, program elements are re-ranked by combining their suspiciousness scores with information from patch groups, enhancing the accuracy and efficiency of the fault localization [40].

To address spectrum-based incapability in multi-fault scenarios, an enhanced spectrum-based fault localization method named *FLITSR* is designed to improve performance in these scenarios. The technique uses an iterative process that incorporates test case reduction (TCR) to identify faults efficiently. Initially, the test suite is executed to locate a single fault, after which TCR is applied to exclude test cases heavily influenced by the coverage of the located fault. This refinement directs subsequent testing toward uncovering new faults [15]. Besides, a research evaluates seven fault localization techniques across four methodological families: spectrum-based, mutation-based, predicate switching, and stack tracing-based approaches, focusing on real Python programs with *FauxPy* as the analysis tool. Spectrum-based methods, including *Tarantula*, *D-Star*, and *Ochiai*, outperformed others in terms of effectiveness, while mutation-based approaches like *Metallaxis* and *Muse* ranked second. The study also examined combining these techniques in two ways, finding enhanced effectiveness with minimal impact on execution time. This work offers insights into the relative strengths of various approaches in large-scale fault localization tasks [8].

Similarly, [41] evaluates the effectiveness of spectrum-based fault localization formulae across real Python projects, focusing on *Tarantula*, *Barinel*, *Ochiai*, *Op*, and *D-Star*. The formula's performance was compared within Python projects, as well as across Java and C projects. Results indicate consistent performance across fault types, except for faults caused by deletion. Notably, older formulae like *Tarantula*, *Ochiai*, and *Barinel* outperform newer ones. Interestingly, while *Tarantula* excels in Python projects, the *Op* formula proves more effective in Java, emphasizing the robustness of traditional methods and uncovering language-specific performance differences.

Additionally, [6] employs spectrum-based fault localization as a classification model to predict test case outcomes (pass/fail) based on spectral information from their execution. EXplainable Artificial Intelligence (XAI) techniques are integrated to examine the localized significance of code segments in influencing these results. XAI analysis on failed test cases identifies the statements with the highest impact on test failures. The methodology dynamically learns how failed test cases contribute to the suspiciousness of various code segments, connecting test execution results with the implicated code parts to enhance fault localization accuracy.

In summary, the reviewed research highlights that analyzing program structure and its evolution history plays a critical role in improving the accuracy of SBFL techniques. This insight emphasizes the need to integrate structural and historical dimensions into these methods to achieve more effective debugging practices in software engineering. The study also points out a notable gap in the literature: limited focus on program evolution and the role of test cases in SBFL. Identifying research that examines how test cases contribute to localization effectiveness remains a significant challenge. To address this, the proposed study introduces a novel approach that combines SBFL with program evolution, structural characteristics, and the test suite, aiming to enhance spectrum-based fault localization techniques.

PROPOSED APPROACH

This section reviews *SBCT* and its research context, highlighting its aim to enhance the accuracy of spectrum-based fault localization techniques by addressing challenges stemming from software evolution. The approach adapts SBFL methods by considering changes introduced during iterative development. It also utilizes test cases and dependencies to ensure precise localization of faulty components in evolving systems.

Building on the validation of individual feature effectiveness in prior research, *SBCT* takes a novel step by innovatively combining features. This approach is designed to identify the most impactful features while ensuring maximum accuracy with the smallest possible set of features, advancing the research objective effectively. The architecture and methodology are illustrated in Fig. 1, with key inputs including:

- **Program p:** Along with its preceding version, to examine evolutionary changes.
- **Test Suite:** Incorporating both successful and failed test cases.
- **Code Change Locations:** Pinpointing the sections of the code altered during evolution.

Initially, the program under evaluation, referred to as p, is executed using its corresponding test suite. During this process, essential data points, such as the coverage matrix and test case outcomes, are collected. These values are aggregated for each program statement to derive the "Test Case Weight" feature. Simultaneously, suspiciousness scores for statements are calculated using the SBFL formulae, forming the "SBFL score." Additionally, information about changes in program statements compared to its prior version is systematically recorded as the "Change" feature. These features are then input into the proposed formula, which computes a suspiciousness score for each statement. Subsequently, the statements are ranked based on these scores. Considering these scores, statements with higher values are a more substantial likelihood of containing faults. This structured process ensures precise localization of the potentially faulty statements. The proposed method also integrates dynamic control and data dependencies into its framework, complementing the previously discussed features. This addition aims to assess how these dependencies influence the fault localization process, enhancing the overall accuracy and depth of the analysis.

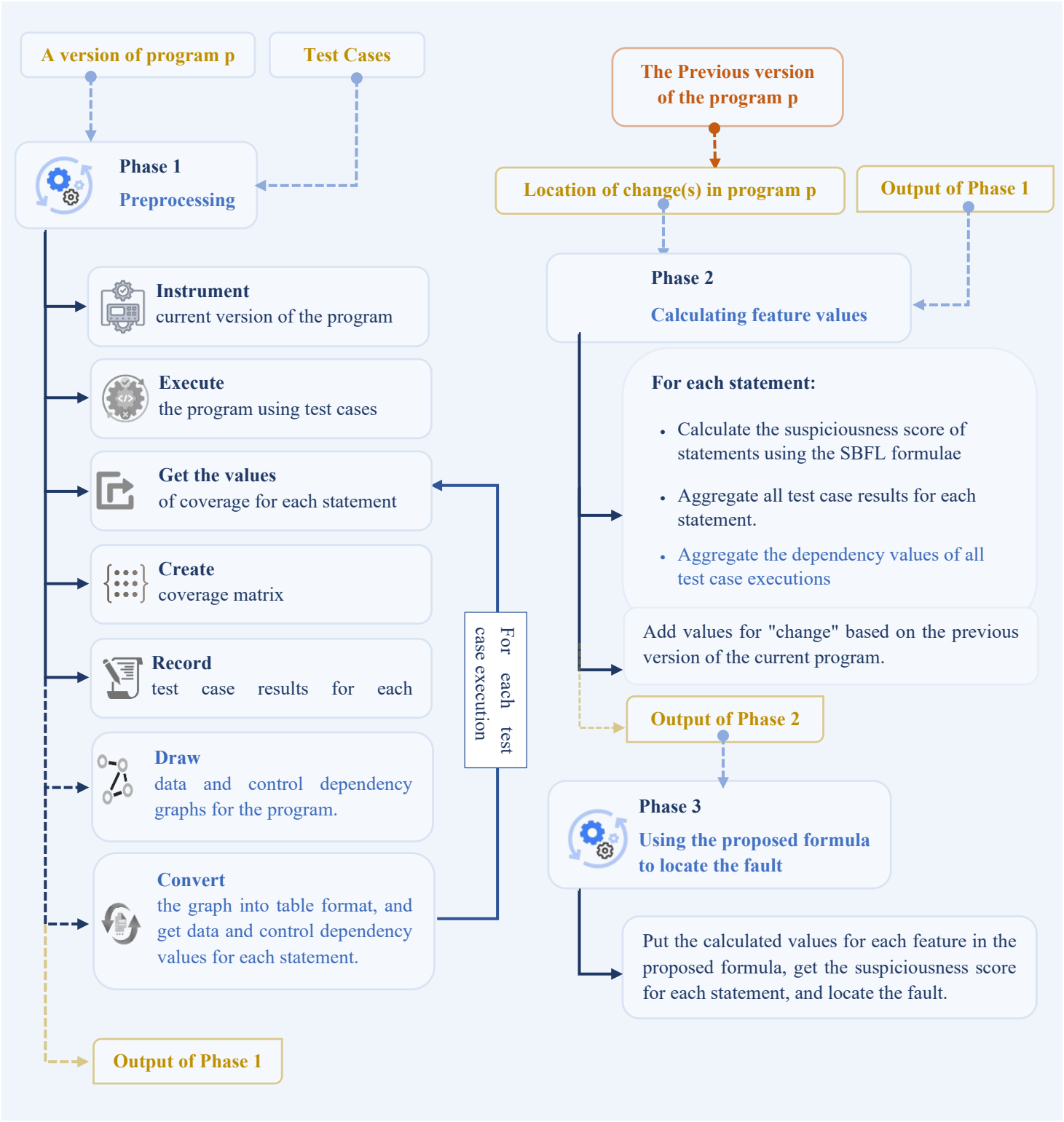

Fig. 1 Architecture and Methodology of *SBCT*

result is captured for the “Test Case Weight” feature. This phase involves documenting the outcomes of the test cases for each statement. As each test case is executed, values in the "Test Case Weight" column are assigned to each statement based on its coverage and test case results. These values are defined as follows:

- If a statement is covered by the test case and the result indicates the test case has passed, the corresponding matrix cell is
- If a statement is covered and the test case result has failed, the matrix cell is assigned a value of 1.
- If a statement is not covered by the test case, the respective matrix cell is marked with 0.

This process ensures that each statement's test case execution and coverage status are systematically recorded, enabling further analysis in the fault localization method. This comprehensive data serves as the foundation for subsequent analytical processes in *SBCT*.

After executing all test cases, the recorded values of “Test Case weight” for each statement are aggregated separately. For example, statement S_1 was executed by 5 test cases. The first and second test cases did not cover S_1, so the related values were 0. The third and fourth test cases covered S_1 and their result were *Pass*, so the related values were -1. The fifth test case covered

S_1 and its result was *Fail*, so its related value was 1. Overall, according to this data, the value for "Test Case Weight" feature for S_1 is -1 (0 + 0 + (-1) + (-1) + 1 = -1)

To expand the experiment, both dynamic data and control dependencies are collected and added to the aforementioned coverage matrix. To incorporate dependencies into the coverage matrix, they are first represented numerically and formatted as table entries. Each dependency is divided into two components—"in" and "out"—resulting in four additional columns in the matrix: Data Dependency (*def*), Data Dependency (*use*), Control Dependency (*dom*), and Control Dependency (*domed*). To briefly introduce these features, Data Dependency (*def*) refers to a value defined within the statement that is subsequently utilized by other statements. Data Dependency (*use*) applies when the statement uses a value previously defined elsewhere. Control Dependency (*dom*) indicates that the statement exerts dominance over other statements in terms of control flow. Control Dependency (*domed*) refers to the statement being dominated by another statement that governs its execution.

Using these definitions, the matrix values are incremented systematically. If a statement defines a value that is later used, the corresponding cell of "Data Dependency (*def*)" in the matrix is incremented by 1. When a statement utilizes a previously defined value, its related cell of "Data Dependency (*use*)" is similarly updated. Regarding "Control Dependency (*dom*)", when a statement dominates others, its associated cell in the matrix is increased by 1 for each dominated statement; and for "Control Dependency (*domed*)", if another statement controls its execution, the respective cell in the matrix is incremented similarly. This structured approach captures the intricate relationships between statements. It also ensures that dependencies are accurately represented and integrated into the fault localization process.

The output of this phase is a table consisting of coverage information, location of modifications, test case results, and, subsequently, dynamic data and control dependencies presented as distinct columns.

5.2 Phase 2: Calculating feature values

In this phase, the matrix generated during Phase 1 undergoes detailed processing. The primary objective is to achieve a comprehensive understanding of the behavior of each program statement against test cases. The aggregated data provides insights into patterns and interactions within the matrix. To this end, a new table is generated to construct the dataset for fault localization. This table contains only features and their final values for applying the formula.

To initialize the "SBFL score" feature, the suspiciousness score of each statement is calculated using an SBFL formula. Then, it is placed in the corresponding cell in the table. Subsequently, an additional feature, referred to as "Change," is incorporated into the matrix to track whether each statement has been modified compared to its previous version. For programs in version 2 or beyond, a statement modified in the current version is assigned a "Change" value of 1. Conversely, statements that remain unchanged are assigned a value of 0. For the first version of the program, all entries in the "Change" column are automatically set to 0, as no prior versions exist for comparison.

Regarding the "Test Case Weight", the values accumulated across multiple test case executions are aggregated for every statement in the code. "Test Case Weight" helps to achieve a comprehensive view of each statement. This aggregation is also applied to data and control dependencies. By integrating both historical and behavioral data, this approach offers a holistic perspective for fault localization.

Table 3 illustrates the outcomes derived from this phase, showcasing the aggregated feature values for each program statement. In this example, the *Tarantula* formula is used for fault localization; however, other SBFL formulae can be used as alternatives. Additionally, the main approach focuses on the "SBFL score," "Change," and "Test Case Weight," with dependencies introduced later to examine their effectiveness alongside other features.

Table 3. Output of Phase 2

Statements	Tarantula scores	Sum of data dependency (def)	Sum of data dependency (use)	Sum of control dependency (dom)	Sum of control dependency (domed)	Change	Test Case Weight
1	0.5	22	0	0	0	0	-8
2	0.5	22	22	0	0	0	-8
3	0.5	22	22	0	0	0	-8
4	0.5	32	0	0	0	0	-8
5	0.5	22	32	66	0	1	-8
6	0.5	85	31	0	22	0	-8
7	0.5	0	22	12	22	0	-8
8	0.666666667	0	0	0	12	0	0
9	0.5	0	22	22	22	0	-8

5.3 Phase 3: Using the proposed formula to locate the fault

In the third phase, the extracted and compiled features from earlier stages are utilized. This phase involves applying specific formulae proposed in this research (Eq. 4-12). The formulae are used to determine a suspiciousness score for each statement in the program. Equation 4 relates to the fundamental idea of the research, using the scores of the *Tarantula* formula as a feature. Equations 5 and 6 pertain to the same experiment, incorporating data dependency and data and control dependencies. The following three equations present the same formulae applied with *Ochiai*, and the subsequent three are dedicated to *Jaccard*. As with all statistical and spectrum-based techniques, a higher suspiciousness score indicates a stronger likelihood that a statement

contains a fault. To improve clarity, the same abbreviations introduced in the motivational example are reused within these formulae.

Suspiciousness score of Sn = Tarantula Score(1.889) + CH*1.938+ TCW*0.03- 3.827* (4)

Suspiciousness score of Sn =
Tarantula Score(1.351) + D1*0.00004+ D2*0.007 + CH*1.509+ TCW*0.004- 3.637* (5)

Suspiciousness score of Sn =
Tarantula Score(1.743) + D1*0.001+ D2*0.01+ C1*(-0.002) + C2*(-0.009) + CH*1.632+ TCW*0.036 - 3.533* (6)

Suspiciousness score of Sn = Ochiai Score(2.512) + CH*1.824+ TCW*0.005- 1.368* (7)

Suspiciousness score of Sn =
Ochiai Score(0.432) + D1*(-0.0001) + D2*0.007 + CH*1.551+ TCW*0.038- 3.203* (8)

Suspiciousness score of Sn =
Ochiai Score(1.168) + D1*0.0009+ D2*0.009+ C1*(-0.003) + C2*(-0.01) + CH*1.728+ TCW*0.026 - 3.611* (9)

Suspiciousness score of Sn = Jaccard Score(0.952) + CH*1.899+ TCW*0.025 - 3.301* (10)

Suspiciousness score of Sn =
Jaccard Score(0.637) + D1*0.002 + D2*0.015 + CH*2.153+ TCW*0.044- 0.744* (11)

Suspiciousness score of Sn =
Jaccard Score(0.628) + D1*0.001+ D2*0.009+ C1*(-0.002) + C2*(-0.008) + CH*1.729+ TCW*0.035 - 3.297* (12)

Using the specified formulae, suspiciousness scores are calculated for all program statements. These scores form the basis for ranking, with statements assigned higher scores being considered more likely to contain faults. Therefore, they are placed higher in the ranking.

It is worth mentioning that the correlation coefficients of each feature in these formulae are obtained by applying logistic regression on the train set using Python libraries. To obtain the most appropriate correlation coefficients, different sampling methods were tested, which are also detailed in Tables 4 to 12.

6. EXPERIMENTAL DESIGN

This section describes the procedure followed in the experiment, detailing the setup for both experimentation and evaluation, as well as the methods employed for data collection.

6.1 Experimental setup

To generate the aforementioned formulae, data were gathered from 14 programs, each containing over three faulty versions. This dataset included 61 source codes and 116 faults, to which Phases 1 and 2 of the proposed method were applied. An additional feature, termed "result," was incorporated into the Phase 2 output table. This feature specified whether a statement was faulty or non-faulty. Specifically, the "result" column assigned a value of 1 for faulty statements and 0 for non-faulty ones.

The primary objective of this task was to distinguish faulty statements from non-faulty ones, framing it as a binary classification problem. To achieve this, logistic regression was chosen to generate the formulae, as the primary objective of the formulae is to classify code statements into two categories: faulty and non-faulty. We added a column titled "result". It acted as the dependent (categorical) variable, and all other features served as independent (explanatory) variables. Given that the dataset's dependent variable included only two possible values—0 for non-faulty and 1 for faulty—there was an evident imbalance. Non-faulty statements significantly outnumbered faulty ones, as is typical in most programs, where the majority of the code is non-faulty.

To address the imbalance and enhance model performance, both oversampling and undersampling methods were tested. For oversampling, the *RandomOverSampler* and *SMOTE* techniques were utilized, while *EditedNearestNeighbours* and *RandomUnderSampler* were applied for undersampling. These methods were explored to determine the most effective solution for the dataset. Once balanced, logistic regression was applied, generating a model that provided correlation coefficients for each independent variable. These coefficients formed the foundation for constructing the final formulae. This experiment aimed to address specific research questions regarding the method and its efficacy:

Q1: What is the role of code modifications in SBFL fault localization techniques?

Q2: Is SBFL a sufficient formula or a supplementary feature for fault localization in evolutionary software?

Q3: Which features work best together to achieve the highest accuracy at locating faults?

In summary, these questions comprehensively capture the essential capabilities of *SBCT.* In other words, they demonstrate how *SBCT* utilizes SBFL techniques alongside historical code modifications, test case weighting mechanisms, and aggregated features. By integrating these elements, *SBCT* significantly enhances the accuracy of fault localization, particularly in single-fault scenarios. It also ensures a more targeted and efficient approach to locating faults.

6.2 Data collection

The study utilized programs and test cases sourced from the *Code4Bench* benchmark, specifically focusing on Java programs written in Java 6. Java was chosen for its extensive availability of source code and corresponding test cases. However, the proposed methodology is designed to be language-agnostic, ensuring applicability across various programming languages. The *Code4Bench* dataset offers authentic code samples with real faults collected from *Codeforces*, featuring concise and straightforward examples. These smaller programs intensify challenges like "tie" issues, making them an ideal testing ground for demonstrating the efficacy of *SBCT*. Only programs with at least three versions were included in the research, with an emphasis on those containing faults that led to incorrect test results. This focus was deliberate, as faults producing erroneous outputs are often more challenging to detect compared to those causing program crashes. Additionally, the dataset includes programs with both single and multiple faults. It provides a comprehensive environment for evaluation. Regarding test cases, only valid test cases were employed in the experiment, encompassing both passing and failing scenarios to ensure robust and meaningful analysis. It is worth mentioning that the train set and test set are two distinct files containing individual programs.

6.3 Evaluation

The effectiveness of *SBCT* was evaluated using two well-established metrics: *EXAM* score and *TOP-N* accuracy. The *EXAM* score measures the proportion of the code that must be inspected to locate a fault. Lower *EXAM* scores signify higher accuracy in fault localization, as fewer code sections require examination. *TOP-N* accuracy assesses the method's effectiveness in identifying faults within the top N positions of a ranked list. For this study, the focus was on *Top-1*, *Top-3*, and *Top-5*, representing whether the fault was detected at the 1st, third, or fifth rank, respectively.

7. EXPERIMENTAL RESULTS

This section explores the analysis of the experimental data to address the research questions. A key part of the evaluation focuses on the correlation coefficient values obtained from applying various undersampling and oversampling algorithms. These values are critical for assessing the effect and efficiency of each feature under examination. To address the research questions, the results of all experiments, along with their analysis, are first presented. Subsequently, the questions are answered based on these results. Tables 4, 5, and 6 provide a comprehensive ranking of the algorithms based on their respective correlation coefficients. They highlight the importance and role of each feature in fault localization using the three main features. These rankings serve as valuable indicators of the relative effectiveness of each feature within the experiment's context. Additionally, the abbreviations used in the tables are outlined to enhance understanding and clarity of the results presented.

EN: EditedNearestNeighbours
RU: Random UnderSampler
RO: Random OverSampler

Table 4. Logistic Regression Correlation Coefficients for the first experiment with *Tarantula* Score

	EN	RU	RO	SMOTE
Tarantula Score	1.889	1.852	3.255	3.418
Change	1.938	1.165	1.644	1.863
Test case Weight	0.03	0.031	0.031	0.034

Table 5. Logistic Regression Correlation Coefficients for the first experiment with *Ochiai* Score

	EN	RU	RO	SMOTE
Ochiai Score	1.473	1.603	2.512	2.838
Change	1.886	1.898	1.824	2.03
Test case Weight	0.017	0.018	0.005	0.001

Table 6. Logistic Regression Correlation Coefficients for the first experiment with *Jaccard* Score

	EN	RU	RO	SMOTE
Jaccard Score	0.952	1.348	2.2	2.423
Change	1.899	1.918	1.852	2.039
Test case Weight	0.025	0.015	0.01	0.007

As shown in Tables 4, 5, and 6, the "SBFL score" and "Change" are both practical and significant features for fault localization. "Change" is the most important feature for fault localization when using undersampling algorithms, followed by "SBFL score" as the second most important. Nonetheless, both features exhibit a similar effect on fault localization. However, "Test Case Weight" demonstrates the least importance and the lowest correlation with the faultiness of statements.

Based on our prior studies [30, 31], "Test Case Weight" proves to be a valuable feature for fault localization in the absence of SBFL, as it provides a similar perspective. Therefore, the "SBFL score" appears to be more significant and informative than "Test Case Weight" in this combination. To further examine these claims, Tables 7, 8, and 9 offer a detailed ranking of the algorithms, incorporating data dependency in addition to the previously evaluated features.

Table 7. Logistic Regression Correlation Coefficients for the second experiment with *Tarantula* Score

	EN	RU	RO	SMOTE
Tarantula Score	1.351	2.121	3.06	3.289
Data dependency (*def*)	0.00004	0.007	0.003	0.003
Data dependency (*use*)	0.007	0.012	0.013	0.014
Change	1.509	1.655	1.736	1.916
Test case Weight	0.004	0.016	0.037	0.035

Table 8. Logistic Regression Correlation Coefficients for the second experiment with *Ochiai* Score

	EN	RU	RO	SMOTE
Ochiai Score	0.432	1.747	2.11	1.985
Data dependency (def)	-0.0001	0.0004	0.002	0.001
Data dependency (use)	0.007	0.013	0.011	0.012
Change	1.551	1.603	1.918	2.121
Test case Weight	0.038	0.01	0.017	0.021

Table 9. Logistic Regression Correlation Coefficients for the second experiment with *Jaccard* Score

	EN	RU	RO	SMOTE
Jaccard Score	-0.473	1.052	0.955	0.637
Data dependency (def)	0.0002	0.003	0.004	0.002
Data dependency (use)	0.008	0.009	0.013	0.015
Change	1.518	1.25	1.793	2.153
Test case Weight	0.055	0.04	0.038	0.044

Tables 7, 8, and 9 confirm the earlier claim. As evident in these tables, “SBFL score” and "Change" have the most impact on locating faulty statements, with "Test Case Weight" ranked as the third most important feature. This indicates that the basic idea involving three features in the first experiment comprises the most critical elements for fault localization.

Regarding data dependencies, "Data Dependency (*use*)" is significantly more impactful than "Data Dependency (*def*)." This distinction arises from how faults manifest during program execution. Even if a value is improperly defined within a statement, its impact remains dormant until the value is utilized. Thus, faults only become observable when the program attempts to use the incorrect value, making "Data Dependency (*use*)" more critical for fault identification and localization. This assertion aligns with the findings of [30]. The following three tables focus on the third experiment, which incorporates control dependency alongside all features from the second experiment.

Table 10. Logistic Regression Correlation Coefficients for the third experiment with *Tarantula* Score

	EN	RU	RO	SMOTE
Tarantula Score	1.743	1.961	3.12	3.382
Data dependency (def)	0.001	0.005	0.003	0.005
Data dependency (use)	0.01	0.026	0.019	0.021
Control dependency (dom)	-0.002	-0.012	-0.009	-0.018
Control dependency (domed)	-0.009	-0.023	-0.013	0.005
Change	1.632	0.955	1.732	2.098
Test case Weight	0.036	0.041	0.039	0.042

Table 11. Logistic Regression Correlation Coefficients for the third experiment with *Ochiai* Score

	EN	RU	RO	SMOTE
Ochiai Score	1.168	1.306	2.21	2.088
Data dependency (def)	0.0009	0.008	0.004	0.003
Data dependency (use)	0.009	0.029	0.017	0.017
Control dependency (dom)	-0.003	-0.012	-0.009	-0.014
Control dependency (domed)	-0.01	-0.028	-0.017	-0.017
Change	1.728	1.606	2.003	2.321
Test case Weight	0.026	0.044	0.018	0.022

Table 12. Logistic Regression Correlation Coefficients for the third experiment with *Jaccard* Score

	EN	RU	RO	SMOTE
Jaccard Score	0.628	0.797	1.155	0.889
Data dependency (def)	0.001	0.002	0.003	0.003
Data dependency (use)	0.009	0.026	0.019	0.214
Control dependency (dom)	-0.002	-0.012	-0.01	-0.013
Control dependency (domed)	-0.008	-0.028	-0.013	-0.013
Change	1.729	1.757	1.829	2.299
Test case Weight	0.035	0.05	0.038	0.048

As seen in Tables 10, 11, and 12, the top three most important features are "Change," "SBFL score," and "Test Case Weight". Similar to the findings of the second experiment, "Data Dependency (*use*)" is more critical than "Data Dependency (*def*)." Conversely, "Control Dependency (*dom*)" appears to hold greater significance than "Control Dependency (*domed*)." This observation underscores the more decisive influence exerted by dominating statements on the program's control flow. It makes them a more pivotal factor in guiding the fault localization process. By identifying dominating statements, the approach gains a clearer understanding of critical control points within the code, which are instrumental in pinpointing faults more effectively. Comparing the two dependencies, data dependency is slightly more influential than control dependency based on their correlation with faulty statements.

Synthesizing the results of these three experiments leads to the answer to the first research question. The correlation coefficient of "Change" with faulty statements is among the highest observed. It highlights its far greater importance compared to "Test Case Weight" and the dependencies. This indicates that code modifications serve as strong indicators for fault localization in regression fault scenarios. To further support this claim, Tables 13, 14, and 15 illustrate the accuracy of the formulae generated in the first experiment. Additionally, Table 16 presents the accuracy of three SBFL formulae—*Tarantula*, *Ochiai*, and *Jaccard*. This assessment focuses on the proportion of programs within the experimental dataset where the formula effectively identified the fault location, as measured by the selected evaluation criteria. Consequently, these tables showcase *Top-1*, *Top-3*, and *Top-5* accuracy. It also demonstrates *EXAM* scores computed for thresholds of 5, 10, 15, and 20, utilizing the aforementioned sampling methods.

Table 13. Results of *SBCT*- First experiment with *Tarantula*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	**27.28%**	**59.1%**	**77.28%**	**27.28%**	**59.1%**	**68.19%**	**77.28%**	**69**	**2.5**
SBCT + RU	27.28%	59.1%	77.28%	27.28%	59.1%	68.19%	77.28%	69	2.5
SBCT + RO	27.28%	59.1%	72.73%	27.28%	59.1%	63.64%	72.73%	69	2.5
SBCT + SMOTE	27.28%	59.1%	72.73%	27.28%	59.1%	63.64%	72.73%	69	2.5

Table 14. Results of *SBCT*- First experiment with *Ochiai*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	36.37%	68.19%	72.73%	36.37 %	63.64%	68.19%	72.73%	62	2.5
SBCT + RU	36.37%	68.19%	72.73%	36.37 %	63.64%	68.19%	72.73%	62	2.5
SBCT + RO	**36.37%**	**68.19%**	**72.73%**	**36.37 %**	**63.64%**	**72.73%**	**72.73%**	**62**	**2.5**
SBCT + SMOTE	36.37%	68.19%	72.73%	36.37 %	63.64%	72.73%	72.73%	62	2.5

Table 15. Results of *SBCT*- First experiment with *Jaccard*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	**36.37%**	**68.19%**	**72.73%**	**36.37 %**	**63.64%**	**68.19%**	**72.73%**	**69**	**2.5**
SBCT + RU	36.37%	68.19%	72.73%	36.37 %	63.64%	68.19%	72.73%	62	2.5
SBCT + RO	36.37%	68.19%	72.73%	36.37 %	63.64%	68.19%	72.73%	62	2.5
SBCT + SMOTE	36.37%	68.19%	72.73%	36.37 %	63.64%	68.19%	72.73%	62	2.5

Table 16. Results of the SBFL formulae

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
Tarantula	9.1%	22.73%	45.46%	9.1%	18.19%	22.73%	45.46%	69	2.5
Jaccard	**27.28%**	**40.91%**	**50%**	**27.28%**	**36.37%**	**50%**	**50%**	**62**	**2.5**
Ochiai	4.55%	18.19%	27.28%	4.55%	13.64%	27.28%	27.28%	71	2.5

Starting with *Tarantula*, as shown in Tables 13 and 16, *SBCT* located 18% more faults by *Top-1* accuracy, 36% more faults by *Top-3* accuracy, and 32% more faults by *Top-5* accuracy. Comparing the *EXAM* scores, the enhanced *Tarantula* outperformed the original *Tarantula* by locating 18% more faults at the maximum *EXAM* of 5%, 41% more faults at an *EXAM* of 10%, 45% more faults at an *EXAM* of 15%, and 32% more faults at an *EXAM* of 20%.

Based on Tables 14 and 16, the results for the enhanced *Ochiai* compared to the original *Ochiai* indicate that *SBCT* locates 32%, 50%, and 45% more faults at the *Top-1*, *Top-3*, and *Top-5* ranks, respectively. In terms of *EXAM* scores, *SBCT* outperformed *Ochiai* by locating 32% more faults by examining a maximum of 5% of the code, 50% more faults by examining a maximum of 10%, and 45% more faults by studying up to 15% and 20% of the code.

Regarding *Jaccard*, comparing the results from Tables 15 and 16 demonstrates that the enhanced *Jaccard* outperformed the original *Jaccard* by locating 9% more faults at *Top-1*, 27% more faults at *Top-3*, and 23% more faults at *Top-5*. Furthermore, *SBCT* located 9% more faults at the maximum *EXAM* of 5%, 27% more faults at the maximum *EXAM* of 10%, 18% more faults at the maximum *EXAM* of 15%, and 23% more faults by examining a *maximum of 20% of the code.*

These findings demonstrate that SBFL can be significantly improved in accuracy by incorporating code modifications. Consequently, these three tables strengthen the claim in response to the first research question. They also indicate that code changes serve as effective clues for locating faulty statements.

Tables 17 to 22 present the results of the evaluation metrics, consistent with Tables 13, 14, and 15, for the second and third experiments. To clarify, Tables 17, 18, and 19 show the results of the experiment incorporating data dependency in addition to the basic features.

Table 17. Results of *SBCT*- Second experiment with *Tarantula*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	36.37%	81.82%	90.91%	40.91%	72.73%	86.37%	90.91%	31	2.5
SBCT + RU	27.28%	77.28%	90.91%	31.82%	72.73%	81.82%	90.91%	41	3
SBCT + RO	**36.37%**	**81.82%**	**90.91%**	**45.46%**	**68.19%**	**86.37%**	**90.91%**	**28**	**2.5**
SBCT + SMOTE	36.37%	81.82%	90.91%	40.91%	72.73%	86.37%	90.91%	28	2.5

Table 18. Results of *SBCT*- Second experiment with *Ochiai*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	**40.91%**	**81.82%**	**95.46%**	**45.46%**	**77.28%**	**90.91%**	**95.46%**	**21**	**2.5**
SBCT + RU	40.91%	90.91%	95.46%	40.91%	72.73%	90.91%	95.46%	21	2.5
SBCT + RO	36.37%	90.91%	95.46%	40.91%	72.73%	95.46%	95.46%	24	2.5
SBCT + SMOTE	36.37%	90.91%	95.46%	40.91%	72.73%	90.91%	95.46%	21	2.5

Table 19. Results of *SBCT*- Second experiment with *Jaccard*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	36.37%	86.37%	95.46%	40.91%	68.19%	86.37%	95.46%	28	2.5
SBCT + RU	31.82%	81.82%	95.46%	36.37 %	68.19%	90.91%	95.46%	31	2.5
SBCT + RO	36.37%	81.82%	95.46%	40.91%	68.19%	90.91%	95.46%	31	2.5
SBCT + SMOTE	**36.37%**	**86.37%**	**95.46%**	**40.91%**	**68.19%**	**90.91%**	**95.46%**	**21**	**2.5**

Considering these three tables, all indicate a significant improvement in the accuracy of *SBCT* compared to the SBFL formulae. Specifically, the enhanced *Tarantula* (Table 17), incorporating "Change," "Test Case Weight," and "data dependency," located 27%, 60%, and 45% more faults than *Tarantula* (Table 16) at *Top-1*, *Top-3*, and *Top-5* rankings, respectively. Similarly, it outperformed *Tarantula* by locating 36%, 50%, 64%, and 45% more faults by examining a maximum of 5%, 10%, 15%, and 20% of the code, respectively. Additionally, the "Max EXAM" column in the tables refers to the maximum *EXAM* score achieved during the experiment. The enhanced *Tarantula* located all faults by examining a maximum of 28% of the code, whereas the original *Tarantula* required analysis of up to 69% of the code.

Regarding *Ochiai*, the enhanced *Ochiai* in the second experiment (Table 18) outperformed the original *Ochiai* (Table 16) by locating 36% more faults at *Top-1*, 64% more at *Top-3*, and 68% at *Top-5* rankings. Moreover, the enhanced *Ochiai* located

41%, 64%, 64%, and 68% more faults by examining a maximum of 5%, 10%, 15%, and 20% of the code, respectively. The enhanced *Ochiai* identified all faults by examining up to 21% of the code, while the original *Ochiai* required a maximum examination of 71%.

In terms of *Jaccard*, the enhanced *Jaccard* (Table 19) located 9%, 45%, and 45% more faults at *Top-1*, *Top-3*, and *Top-5* rankings, respectively. It further outperformed the original *Jaccard* (Table 16) by identifying 14% more faults by examining a maximum of 5% of the code, 32% more faults at 10%, 41% more faults at 15%, and 45% more faults at 20%. The enhanced *Jaccard* located all faults by analyzing up to 21% of the code, whereas the original *Jaccard* required examination of 62%.

This experiment underscores the importance of data dependency in fault localization. The results from the second experiment also demonstrate higher accuracy compared to the first. As evident from this experiment and two prior studies [30, 31], data dependency proves to be an effective distinguishing feature for differentiating statements in the code. Similar to the second experiment, Tables 20 through 22 focus on the third experiment, which incorporates the control dependency feature alongside the features from the second experiment. This inclusion highlights the impact of control dependency on the results, providing a comparative perspective between experiments.

Table 20. Results of *SBCT*- Third experiment with *Tarantula*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	**40.91%**	**81.82%**	**90.91%**	**45.46%**	**68.19%**	**86.37%**	**90.91%**	**41**	**2.5**
SBCT + RU	45.46%	59.1%	77.28%	45.46%	54.55%	63.64%	86.37%	41	3
SBCT + RO	40.91%	81.82%	90.91%	45.46%	63.64%	86.37%	90.91%	34	2.5
SBCT + SMOTE	40.91%	77.28%	90.91%	45.46%	68.19%	81.82%	90.91%	48	2.5

Table 21. Results of *SBCT*- Third experiment with *Ochiai*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	**40.91%**	**77.28%**	**90.91%**	**45.46%**	**68.19%**	**90.91%**	**90.91%**	**41**	**2.5**
SBCT + RU	40.91%	59.1%	81.82%	40.91%	50%	72.73%	81.82%	45	2.5
SBCT + RO	36.37%	77.28%	90.91%	40.91%	63.64%	86.37%	90.91%	48	2.5
SBCT + SMOTE	40.91%	72.73%	90.91%	45.46%	68.19%	81.82%	90.91%	48	2.5

Table 22. Results of *SBCT*- Third experiment with *Jaccard*

	Top-1	Top-3	Top-5	Exam<5	Exam<10	Exam<15	Exam<20	Max EXAM	Min EXAM
SBCT + EN	**40.91%**	**77.28%**	**95.46%**	**45.46%**	**68.19%**	**86.37%**	**95.46%**	**41**	**2.5**
SBCT + RU	36.37%	72.73%	90.91%	40.91%	68.19%	77.28%	90.91%	38	2.5
SBCT + RO	36.37%	81.82%	90.91%	40.91%	68.19%	86.37%	90.91%	38	2.5
SBCT + SMOTE	36.37%	59.1%	81.82%	36.37%	45.46%	68.19%	81.82%	77	2.5

The data from these three tables demonstrates a substantial improvement in the accuracy of *SBCT* compared to traditional SBFL formulae. Specifically, the enhanced *Tarantula* in Table 20 identified 32%, 59%, and 45% more faults than the original *Tarantula* (Table 16) at *Top-1*, *Top-3*, and *Top-5* rankings, respectively. Furthermore, it exceeded the original *Tarantula* by detecting 36%, 50%, 64%, and 45% more faults when analyzing a maximum of 5%, 10%, 15%, and 20% of the code, respectively. Regarding the maximum *EXAM* score, the enhanced *Tarantula* uncovered all faults by inspecting a maximum of 41% of the code, whereas the original *Tarantula* required up to 69% of the code to achieve the same result.

Focusing on *Ochiai*, the enhanced version in Table 21 outperformed the original (Table 16) in the second experiment by identifying 36% more faults at *Top-1*, 59% more at *Top-3*, and 64% more at *Top-5* rankings. Additionally, it surpassed the original *Ochiai* by locating 41%, 54%, 64%, and 64% more faults after examining a maximum of 5%, 10%, 15%, and 20% of the code, respectively. The enhanced *Ochiai* successfully located all faults by reviewing a maximum of 41% of the code, whereas the original required inspecting up to 71%.

As for *Jaccard*, the enhanced version in Table 22 identified 14%, 36%, and 45% more faults at *Top-1*, *Top-3*, and *Top-5* rankings, respectively. It further surpassed the original *Jaccard* (Table 16) by locating 18%, 32%, 36%, and 45% more faults within a maximum of 5%, 10%, 15%, and 20% of the code, respectively. Notably, the enhanced *Jaccard* achieved complete fault detection after analyzing just 41% of the code, while the original version required inspection of up to 62%.

To summarize the results, "Change" leads to a significant improvement in the accuracy of SBFL. Moreover, these findings highlight the critical role of data dependency in fault localization, as the second experiment achieved markedly higher accuracy than both the first and third experiments. The improvements across all experiments demonstrate the enhanced method's greater efficiency in fault localization by substantially reducing the portion of code that needs to be inspected.

To address the second research question, all three experiments with three different SBFL formulae indicate that the SBFL formula functions more effectively as a feature for fault localization than as a standalone formula for locating faults. Regarding the third research question, the second experiment produced higher accuracy and better performance compared to the first and third

experiments. In the third experiment, although Control Dependency improved the basic SBFL formulae, its impact was not as significant as the effect of Data Dependency. Therefore, we conclude that achieving high accuracy in fault localization requires extracting coverage information, incorporating data dependency, and considering code modifications. These three features and the associated calculations to derive feature values prove to be the most efficient approach for fault localization.

8. CONCLUSION

Fault localization is an integral yet labor-intensive phase of software development. To streamline this process without sacrificing accuracy, various methodologies have been developed, including spectrum-based, mutation-based, and learning-based approaches. Among these, spectrum-based fault localization (SBFL) is particularly notable for its simplicity and reasonable accuracy. However, SBFL techniques often fall short in specific scenarios. Additionally, while fault localization has been extensively studied, there is a notable gap in research addressing the role of software evolution. To address these gaps, this research focuses on enhancing fault localization by incorporating the effects of software evolution and the impact of test cases on fault proneness, with an emphasis on the evolving nature of software.

The proposed solution integrates four critical features: "SBFL score" as a declarative feature to locate faulty statements, "Test Case Weight" as an effective feature for fault localization using easily gathered data, "Change" as a feature capturing software evolution, and "program dependencies" as a feature to distinguish statements with similar spectra. These features are utilized within a logistic regression-based machine learning framework to derive the proposed formula. Experimental evaluations demonstrate that *SBCT* outperforms three widely used SBFL methods across three different experiments. Based on the best results from the second experiment, *SBCT* achieved *Top-3* success rates of 82% for *Tarantula*, 82% for *Ochiai*, and 86% for *Jaccard*. Moreover, the findings underscore "Change" and "SBFL score" as pivotal features in improving fault localization accuracy.

Future directions could explore the effectiveness of *SBCT* with a larger variety of programs, encompassing diverse attributes such as size and programming language. Additionally, the integration of the “SBFL score” as a feature into other research could be examined to evaluate its impact across various combinations and scenarios. Furthermore, with the advancement of neural networks and the expansion of their use in the field of software testing and debugging, the proposed method can be investigated using neural networks such as GNNs.

9. THREATS TO VALIDITY

Several factors could impact the reliability of our findings. One such factor is the relatively small scale of the benchmark used in this study, both in terms of source codes and the accompanying test suite. To improve this, future research should test the method using larger, more diverse benchmark suites with varying sizes, domains, and types of faults. Additionally, the collection of numerous features individually makes data gathering and preprocessing time-consuming and resource-heavy. Although we have taken measures to minimize these issues, such as applying consistent extraction procedures and automating tasks where possible, additional large-scale validation is needed to further strengthen the reliability of the results.

10. REFERENCES

[1] A. Zakari, S. P. Lee, K. A. Alam, and R. Ahmad, Software fault localisation: a systematic mapping study, IET Software, vol. 13, no. 1, pp. 60–74, Feb. 2019.

[2] Z. Li, H. Wang, and Y. Liu, HMER: A Hybrid Mutation Execution Reduction approach for Mutation-based Fault Localization, Journal of Systems and Software, vol. 168, p. 110661, Oct. 2020.

[3] Y. Yang, F. Deng, Y. Yan, and F. Gao, A Fault Localization Method Based on Conditional Probability. In IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Jul. 01, 2019.

[4] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S. C. Cheung, Historical Spectrum Based Fault Localization, IEEE Transactions on Software Engineering, vol. 47, no. 11, pp. 2348–2368, Nov. 2021.

[5] A. Zakari, S. P. Lee, and I. A. Targio Hashem, A Community-Based Fault Isolation Approach for Effective Simultaneous Localization of Faults, IEEE Access, vol. 7, pp. 50012–50030, 2019.

[6] R. Widyasari, G. A. A. Prana, S. A. Haryono, Y. Tian, H. N. Zachiary, and D. Lo, XAI4FL: Enhancing spectrum-based fault localization with explainable artificial intelligence. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 499-510, 2022.

[7] R. Widyasari, J. W. Ang, T. G. Nguyen, N. Sharma, and D. Lo, Demystifying faulty code: Step-by-step reasoning for explainable fault localization. In 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 568-579, 2024.

[8] M. Rezaalipour and C. A. Furia, An empirical study of fault localization in Python programs, Empirical Software Engineering, vol. 29, no. 4, 2024.

[9] X. Xiao, Y. Pan, B. Zhang, G. Hu, Q. Li, and R. Lu, ALBFL: A novel neural ranking model for software fault localization via combining static and dynamic features, Information and Software Technology, vol. 139, p. 106653, 2021.

[10] A. Dutta, S. S. Srivastava, S. Godboley, and D. P. Mohapatra, Combi-FL: Neural network and SBFL-based fault localization using mutation analysis, Journal of Computer Languages, vol. 66, p. 101064, 2021.

[11] A. Majd, M. Vahidi-Asl, A. Khalilian, and B. Bagheri, ConsilientSFL: using preferential voting system to generate combinatorial ranking metrics for spectrum-based fault localization, Applied Intelligence, vol. 52, no. 10, pp. 11068–11088, 2022.

[12] Z. Li, Y. Song, G. Gong, S. Zhou, and K. Lv, A multi-technique fusion approach for fault localization in manufacturing software, Journal of Intelligent, and Fuzzy Systems, vol. 38, no. 1, pp. 229–238, 2020.

[13] Z. Cui, M. Jia, X. Chen, L. Zheng, and X. Liu, Improving Software Fault Localization by Combining Spectrum and Mutation, IEEE Access, vol. 8, pp. 172296–172307, 2020.
[14] Y. Li, S. Wang, and T.N. Nguyen, Fault localization to detect co-change fixing locations. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 659-671, 2022.
[15] D. Callaghan and B. Fischer, Improving spectrum-based localization of multiple faults by iterative test suite reduction. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1445–1457, 2023.
[16] 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 - ASE '05, 2005.
[17] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, Pinpoint: problem determination in large, dynamic Internet services. In Proceedings International Conference on Dependable Systems and Networks, pp. 595-604, 2002.
[18] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund, An Evaluation of Similarity Coefficients for Software Fault Localization. In 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06), pp. 39-46, 2006.
[19] W. E. Wong, V. Debroy, R. Gao, and Y. Li, The DStar Method for Effective Software Fault Localization, IEEE Transactions on Reliability, vol. 63, no. 1, pp. 290–308, 2014.
[20] Y. Li, S. Wang, and T. Nguyen, Fault Localization with Code Coverage Representation Learning. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 661-673, 2021.
[21] Y. Küçük, T. A. D. Henderson, and A. Podgurski, Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 649-660, 2021.
[22] X. Xu, C. Zou, and J. Xue, Every Mutation Should Be Rewarded: Boosting Fault Localization with Mutated Predicates. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 196-207, 2020.
[23] L. Zhang, Z. Li, Y. Feng, Z. Zhang, W. K. Chan, J. Zhang, and Y. Zhou, Improving Fault-Localization Accuracy by Referencing Debugging History to Alleviate Structure Bias in Code Suspiciousness, IEEE Transactions on Reliability, vol. 69, no. 3, pp. 1021–1049, Sep. 2020.
[24] J. Sohn and S. Yoo, FLUCCS: using code and change metrics to improve fault localization. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 273-283, 2017.
[25] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and Li. Zhang, Boosting coverage-based fault localization via graph-based representation learning. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 664-676, 2021.
[26] X. Liu, Y. Liu, Z. Li, and R. Zhao, Fault Classification Oriented Spectrum Based Fault Localization. In 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Vol. 1, pp. 256-261, 2017.
[27] H. Zhong and H. Mei, Learning a graph-based classifier for fault localization, Science China Information Sciences, vol. 63, no. 6, 2020.
[28] D. Ghosh and J. Singh, Spectrum-based multi-fault localization using Chaotic Genetic Algorithm, Information and Software Technology, vol. 133, p. 106512, 2021.
[29] A. Maru, A. Dutta, K. V. Kumar, and D. P. Mohapatra, Software fault localization using BP neural network based on function and branch coverage, Evolutionary Intelligence, vol. 14, no. 1, pp. 87–104, 2019.
[30] F. Aghazade-Par and M. Vahidi-Asl, Feature-Based Fault Localization in Evolving Software: Leveraging Regression Testing Insights, IEEE Access, Vol. 13, pp. 147369 – 147382, 2025.
[31] F. Aghazade-Par and M. Vahidi-Asl, EAFL: an effective combination of features for fault localization in evolving programs, Software Quality Journal, Vol. 33, no. 34, pp. 1-22, 2025.
[32] A. Majd, M. Vahidi-Asl, A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani, Code4Bench: A multidimensional benchmark of Codeforces data for different program analysis techniques, *Journal of Computer Languages*, Vol. 53, pp. 38–52, 2019.
[33] E. Wong, T. Wei, Y. Qi, and L. Zhao, A crosstab-based statistical method for effective fault localization. In 2008 1st international conference on software testing, verification, and validation, pp. 42–51, 2008.
[34] D. Ginelli, O. Riganelli, D. Micucci, and L. Mariani, Exception-Driven Fault Localization for Automated Program Repair. In 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) pp. 598-607, 2021.
[35] A. Dutta, S. S. Srivastava, S. Godboley, and D. P. Mohapatra, Combi-FL: Neural network and SBFL based fault localization using mutation analysis, Journal of Computer Languages, 66, 101064, 2021.
[36] Z. Peng, X. Xiao, G. Hu, A. Kumar Sangaiah, M. Atiquzzaman, and S. Xia, ABFL: An autoencoder based practical approach for software fault localization, Information Sciences, vol. 510, pp. 108–121, 2020.
[37] H. He, J. Ren, G. Zhao, and H. He, Enhancing Spectrum-Based Fault Localization Using Fault Influence Propagation, IEEE Access, vol. 8, pp. 18497–18513, 2020.
[38] D. Silva-Junior, P. S. Leitao-Junior, A. Dantas, C. G. Camilo-Junior, and R. Harrison, Data-flow-based evolutionary fault localization. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, pp. 1963-1970, 2020.
[39] J. Sohn, G. An, J. Hong, D. Hwang, and S. Yoo, Assisting Bug Report Assignment Using Automated Fault Localisation: An Industrial Case Study. In 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 284-294, 2021.
[40] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, Can automated program repair refine fault localization? a unified debugging approach. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 75-87, 2020.
[41] R. Widyasari, G. A. A. Prana, S. A. Haryono, S. Wang, and D. Lo, Real world projects, real faults: evaluating spectrum based fault localization techniques on Python projects, Empirical Software Engineering, Vol. 27, no. 6, 147, 2022.

FAEZE AGHAZADE-PAR received the M.S. degree in information technology engineering from Shahid Beheshti University, Tehran, Iran, where she is currently pursuing the Ph.D. degree in software engineering. Her research interests include software debugging and gamification.
Application ID: JICSE-2511-1096 (R1)
ORCID: 0009-0003-5530-1310
Email: f_aghazadepar@sbu.ac.ir
Address: Shahid Beheshti University, Tehran 1983969411, Iran

MOJTABA VAHIDI-ASL received the B.S. degree in computer engineering from the Amirkabir University of Technology and the M.S. and Ph.D. degrees in software engineering from Iran University of Science and Technology. He is currently an Assistant Professor with the Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran. His research interests include program analysis, software testing, and debugging.
Application ID: JICSE-2511-1096 (R1)
ORCID: 0000-0003-4964-992X
Email: mo_vahidi@sbu.ac.ir
Address: Shahid Beheshti University, Tehran 1983969411, Iran