PRCMHFL: A Fault Localization Technique based on Predicate Ranking and CMH method

Sangharatna Godboley[1,*], Shubhank Kulshreshtha[2], Prayanshu Agarwal[3], B Ranilbala[4]
[1,2,3,4]Department of Computer Science and Engineering,
National Institute of Technology, Warangal, Telangana, India
sanghu@nitw.ac.in, {kulshr_971981,agarwa_941942,ranil_971956}@student.nitw.ac.in
*corresponding author

Abstract—Fault localization is an important step in the debugging process, as it targets to identify the root cause of failures in software systems. Despite the existence of several techniques and tools for fault localization, it remains a challenging and time-taking task. In this paper, we propose a method for fault localization that takes the advantages of reduced code examination in predicate ranking and the effectiveness of the Cochran–Mantel–Haenszel (CMH) method. We experiment using the exam score metric and demonstrate its effectiveness. It accurately identifying faulty program elements. Our work represents a significant improvement in fault localization compared to other important SBFL techniques.

Keywords–Fault localization;CMH;Coverage;Testing;

1. Introduction

Software testing [3, 5, 40, 41, 42, 43] is a critical stage in ensuring the quality of software products. Even with the best efforts of developers and testers, software defects / faults can still skipped. Later they cause issues in production. Finding the reasons of these faults is a challenging and time-consuming task. It also requires expertise and several skill. Fault localization is for detecting the defective code segment [11]. It is a fundamental step in debugging and fixing the software issue [7, 8, 10, 12].

Fault localization can be approached in several ways [4, 14, 20]. It can be manual inspection or automated techniques [13, 16]. As per the literature survey, researchers have developed several techniques and tools to aid in fault localization. Techniques such as spectrum-based techniques, data mining approaches, and model-based techniques are considered to be important. These techniques use various forms of program execution data, such as execution traces, program spectra, and program models, to detect the faulty code. Also, there are debugging, fault localization, and software analytics tools that can be used to facilitate the fault localization process.

Spectrum-Based Fault Localization (SBFL) [6] techniques are important in software engineering domain. The main objective is to detect and localize faults in software systems. These techniques provides information about the coverage of program elements (such as statements, blocks, and functions) and test case outcomes to calculate a suspiciousness score for each element, which can be used to prioritize debugging efforts.

SBFL techniques such as Ample [21], DStar [17], Barinel [6], crosstab [24], Zoltar [23], and Ochiai [22]. Ample depends on the intuition that defects/faults are more likely to be located in code that is executed by failing test cases rather by passing test cases. Barinel proposed an approach by combining SBFL with probabilistic reasoning. This is to estimate the probability that each program element is faulty. Ochiai, like Tarantula, calculates the suspiciousness score of a program element as the ratio of the number of failing test cases that cover the element to the square root of the product of the total number of failing test cases and the total number of test cases that cover the element. DStar, on the other hand, uses the binary similarity coefficient derived from the Kulczynski coefficient to measure the similarity between the coverage vectors of failing and passing test cases and uses this information to compute the suspiciousness score of each program element.

The study [25] revealed that predicates from branch conditions are the most significant contributors to the Top-1 recall of fault localization accuracy among all predicates. Using spectrum-based fault localization formulas for statistical debugging predicates resulted in a significant increase in localization accuracy, with a 227.9% increase in Top-1 compared to the original statistical debugging formula and a 52.7% increase compared to a revised statistical debugging formula. Collecting information at the statement level increased localization accuracy by 69.9% with respect to Top-1, although it led to a 40.0% increase in execution time compared to the method level. These findings have important implications for the development of more accurate and efficient fault localization techniques.

2. Related Work

In this section, we present an overview of relevant literature and approaches in the field of fault localization. We discuss various fault localization techniques, including program slicing, spectrum-based methods, machine learning-based approaches, mutation-based strategies, and miscellaneous methods. Each technique is summarized briefly, providing a comprehensive understanding of the state-of-the-art in fault localization within the context of software development.

2.1. Slice-based Fault Localization

Weiser et al. [32] proposed program slicing to reduce the size of a program by selecting certain criteria. The criteria being

2693-9371/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS-C60940.2023.00085

only those lines/statements that affect the value of a specific variable at a location in the program. This helps developers to isolate the portion of the program that may contain faults. Lyle et al. [33] introduced program dices, which are the intersection of two program slices. Further these can be narrow down for the search scope of faulty statements. This will make the debugging process more efficient.

Korel et al. [34] proposed Dynamic slicing. This creates a slice of the program that includes only the statements which are running by a failed test case. It analyzes their effect on the targeted variables. This will be useful when dealing with complex programs. Other Program slicing methods have been proposed, including thin slicing and hybrid slicing. Thin slicing involves selecting only the statements that directly affect the variable's value. On the other hand hybrid slicing combines both static and dynamic slicing. This is to create a more comprehensive slice of the program. Different Program slicing methods can be applied in various scenario for effective and efficient process.

2.2. Spectrum-based Fault Localization

While slice-based techniques have limitations, as bugs may not be found within the chosen slice and examining many statements can be time-consuming, spectrum-based (SBFL) techniques provide a susceptibility score to each statement using a mathematical formula that considers execution information and test case results. SBFL techniques such as Tarantula, Jaccard, Crosstab, Ochiai, and Barinel require less code examination than other fault localization techniques. D* is a well-known SBFL technique. Despite their effectiveness, SBFL techniques may encounter ties when many statements have identical suspiciousness scores, and they rely solely on test results without distinguishing their contributions.

2.3. Machine Learning-based Fault Localization

Machine learning (ML) techniques, including Back Propagation Neural Networks (BPNN), Radial Basis Function Neural Networks (RBFNN), and Convolutional Neural Networks (CNN), have been used for fault localization in software systems. These algorithms learn patterns between program features and fault labels to predict the likelihood of a statement containing a fault. BPNN, RBFNN, and CNN have all been used successfully in fault localization studies, demonstrating their effectiveness in real-world applications. Overall, machine learning-based fault localization shows promise for improving the efficiency and effectiveness of fault localization.

2.4. Mutation-based Fault Localization

Mutation-based fault localization [31] is a technique that involves introducing small changes to the code and then running a set of test cases to detect any resulting failures. By comparing the execution profiles of the original and mutated versions of the code, mutation-based fault localization can pinpoint which parts of the code are most likely to contain faults. The MUSE technique is an extension of traditional mutation-based fault localization that incorporates a broader range of mutation operators and prioritizes the use of mutations that are more likely to expose faults. These techniques are powerful tools for software developers looking to improve the quality and reliability of their code.

2.5. Miscellaneous Fault Localization Techniques

Some of the miscellaneous fault localization methods include:

- Multiple Fault Localization (MFL) in the refined technique of software fault localization (SFL). This presents a challenging yet increasingly pertinent domain [36]. Literature survey, reveals a growing interest in MFL, particularly over the last five years, marked by a stable expansion. Among the MFL debugging approaches, the One-bug-at-a-time debugging approach (OBA), parallel debugging approach, and multiple-bug-at-a-time debugging approach (MBA) our good. The OBA being the most widely used technique.
- Singh et al. [35] provide a concise overview of significant fault localization techniques employing soft computing methodologies. It becomes evident that more promising results can be achieved through the integration of machine learning techniques, accompanied by a reduction in time constraints. The main objective of this study is to explore fault localization techniques in conjunction with soft computing approaches. It aims of minimizing time and space complexities, thereby enhancing usability and effectiveness.
- DeepRL4FL [38] is a fault localization technique that uses convolutional neural networks for bug localization at the method and statement levels, significantly enhancing accuracy over other techniques. All three techniques use different approaches to improve the accuracy and efficiency of software debugging.
- Traceability-based Fault Localization [37] is the technique that leverages traceability information between requirements, test cases, and code to aid in fault localization. It explores how code changes relate to specific requirements and test cases, helping developers pinpoint potential sources of faults.

3. Proposed Approach

We have introduced a fault localization method called Predicate Ranking Cochran–Mantel–Haenszel Fault Localisation (PRCMHFL), which combines the strengths of Predicate Ranking [2] and the CMH method [1]. This approach enables accurate fault identification while reducing the amount of code that needs to be examined. Our goal is to determine the suspiciousness score of a given statement 's' in a program 'P,' representing the probability of it containing an error. We provide a detailed description of PRCMHFL in this section. The meanings of the symbols that are utilized for our technique are listed in Table I

3.1. Overview

Fig. 1 illustrates the flow of PRCMHFL. We employ several components that take different inputs and, ultimately, generate a ranked list of statements. In this study, we construct a contingency table for each statement in the program under

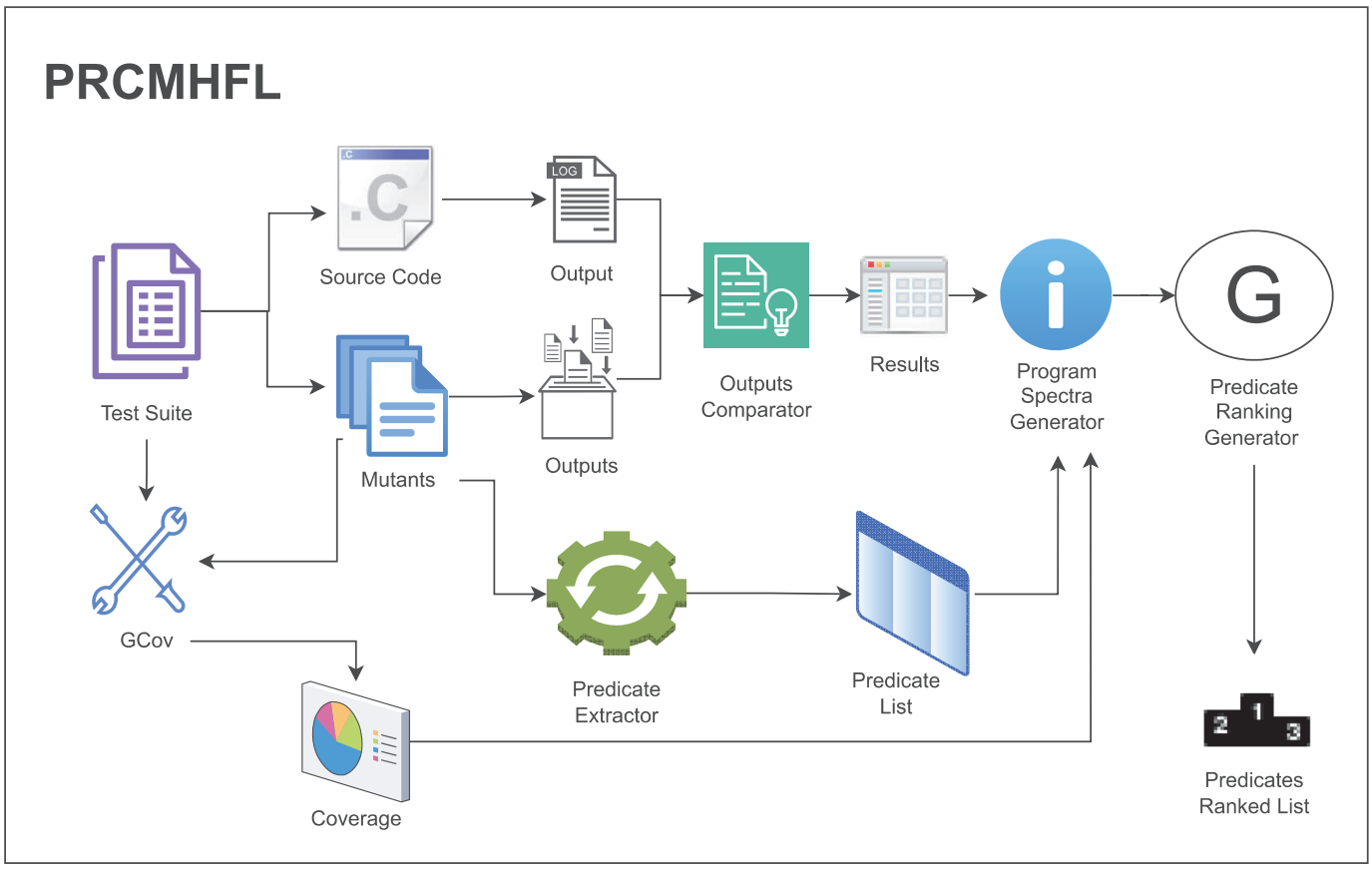


Figure 1: Schematic representation of PRCMHFL

TABLE I: Notations used in our proposed approach. Note: TCs: Test Cases and s: Statement

N	#TCs
N_s	#Success TCs
N_f	#Failed TCs
$N_c(s)$	#TCs covered s
$N_u(s)$	#TCs not covered s
$N_{cs}(s)$	#Success TCs covered s
$N_{cf}(s)$	#Failed TCs covered s
$N_{us}(s)$	#Success TCs not covered s
$N_{uf}(s)$	#Failed TCs not covered s

analysis. The table is two-dimensional, with columns representing the number of successful and failed test cases and rows representing the number of statements covered and uncovered by those test cases. We then conduct a hypothesis test on each contingency table to determine the relationship between program execution and coverage results for the corresponding statement, based on the CMH test. To rank the statements according to their suspiciousness scores accurately, we follow the rules as defined in PRFL [2]. We used gcov tool [39] in our work. It is a static coverage tool used to compute the execution of the code during a program's runtime.

3.2. Algorithmic Description

Algorithm 1 shows the Program Spectra Generator (PSG). The program accepts inputs: 'statement,' 'predicate_list,' and 'test_cases.' and produces four values: 'Ncf,' 'Nus,' 'Ncs,' and 'Nuf,'. These values representing specific statistics related to test cases and program statements. Initially the values are set to 0. For each 'test_case' in the 'test_cases' it evaluates whether the 'statement' is a predicate or not. If it is a statement, the algorithm recursively calls itself ('PSG') with the next statement along with the 'predicate_list' as inputs. Next, in case the 'statement' is not a decision / predicate but it is inside a block, then it checks if it is the first statement of the block or not. Now, whether the 'test_case' passed or failed and whether it covered or uncovered the statement, the values for 'Ncs,' 'Nus,' 'Ncf,' or 'Nuf' are updated accordingly. In case the 'statement' does not meet the above conditions, it will recursively calls itself with the previous statement and the 'predicate_list' as inputs. At last, it returns the values of 'Ncf,' 'Nus,' 'Ncs,' and 'Nuf.' This algorithm plays a important role in our proposed framework.

3.3. Detailed Description

In this section, we discuss on our work in detailed. The **Output Comparator** targets to generate program output using a predefined set of test cases and mutants. It executes original and mutated programs on all the test cases to produce the files. The success of a test case is considered by matching the outputs of the original and the mutated version of the program.

In case the outputs differ, then the test case is considered successful otherwise, it is a failed test case.

The **Predicate Extractor** processes a C-program as input. It produces a list of predicates using logical and relational operators. Then it extracts lines / statements that meet the specified conditions for the analysis.

Algorithm 1: Program Spectra Generator

Data: statement, predicate_list, test_cases
Result: Ncf, Nus, Ncs, Nuf
PSG($statement, predicate\_list$)
Ncf←0 ;
Nuf←0 ;
Ncs←0 ;
Nus←0 ;
for *test_case in test_cases* **do**
  if *statement is predicate* **then**
    Ncf, Nus, Ncs, $Nuf \leftarrow PSG(next\_statement, predicate\_list)$;
  else if *statement inside block* **then**
    if *first statement of block* **then**
      if *test_case passed and covered* **then**
        $Ncs \leftarrow Ncs + 1$
      else if *test_case passed and uncovered* **then**
        $Nus \leftarrow Nus + 1$
      else if *test_case failed and covered* **then**
        $Ncf \leftarrow Ncf + 1$
      else if *test_case failed and uncovered* **then**
        $Nuf \leftarrow Nuf + 1$
    else
      Ncf, Nus, Ncs, $Nuf \leftarrow PSG(prev\_statement, predicate\_list)$;
  else
    Ncf, Nus, Ncs, $Nuf \leftarrow PSG(statement, predicate\_list)$;
return Ncf, Nus, Ncs, Nuf

The **GCOV Generator** program employs the 'gcov' tool [39] to generate in-depth reports on code execution. It takes as input a C program and a set of test cases, compiles the C program with the 'gcov' option enabled, and executes each test case to collect coverage data. This process enables us to determine which sections of the program are being executed by each test case and, consequently, identifying areas of the program that may be susceptible to errors.

The **Program Spectra Generator** is a crucial component in our proposed system. It receives input from three sources: test results from the Output Comparator, a statement coverage report from the GCOV Generator, and a predicate list from the Predicate Extractor. It then assesses the statements to calculate the parameter values used in the contingency table [1] to determine the CMH score [1]. In CHMFL [1], the predicates could not be assessed correctly as some of them had 100% coverage. But, according to PRFL [2], conditions are more susceptible to bugs, and thus, predicates as a whole entity require a correct evaluation. Thus, we took the list from the Predicate Extractor and valued the whole block (including the predicate) using the score allotted to the first statement in the block, reducing complexity and allotting the correct score to the predicate as well.

The **Statement Ranking Generator** module takes input from the Program Spectra and the result of passed and failed test cases from the Output Comparator. It calculates the suspiciousness score for each statement, using the rules specified in PRFL, giving higher preference to predicates as referenced in section. This generates a ranked list of statements from which a fault can be identified by examining a certain number of statements.

4. Experimental Setup

In this section, we provide a detailed account of the experiments conducted and the results obtained.

4.1. Experimental Environment

Our experiments were conducted on a 64-bit Linux system with 3GB of RAM. The programs in our dataset were implemented in the C programming language. We leveraged GCOV, a code coverage tool, in conjunction with GCC, for the analysis of static code coverage in the subject C programs. All other modules were developed using the Python programming language.

4.2. Dataset

To showcase the effectiveness of PRCMHFL, we carried out experiments on a dataset consisting of eleven C programs. The initial six programs were obtained from the Siemens suite, while the remaining five programs were sourced from the NTS repository, which is available for download from the SIR Repository[1]. Table II provides comprehensive information about the C programs. The Siemens suite is widely recognized as a benchmark for evaluating fault localization techniques.

4.3. Evaluation Metric

In this section, we introduce a valuable metric utilized for evaluating PRCMHFL. Our study employs the EXAM Score to assess the quality of PRCMHFL, which is calculated using Eq. 1.

$$EXAM\_Score = \frac{|S_{examined}|}{|S_{total}|} \times 100 \tag{1}$$

Here, $|S_{examined}|$ represents the rank of the statement in the list, while $|S_{total}|$ indicates the number of lines on which the test case ran. A lower EXAM score signifies a better technique. As previously mentioned, a Spectrum-Based Fault Localization (SBFL) technique provides a ranked list of statements. The CMH method is employed as a criterion to calculate the score for all statements. The CMH[2] Test, stated in Eq. 2 to

[1]https://sir.csc.ncsu.edu/portal/index.php
[2]https://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704-ep713_confounding-em/BS704-EP713_Confounding-EM7.html

TABLE II: Data set characteristics

Programs	#Mutants	LOCs	#Exec.Lines	#Tests
PrintTokens	7	565	195	4140
PrintTokens2	10	510	200	4140
Schedule	9	412	152	2650
Schedule2	10	307	128	2710
Tcas	41	173	65	1608
TotInfo	23	406	122	1026
quickSort	6	99	59	128
cfgTest	25	93	46	55
merge2BSTree	8	226	93	197
nextDate	14	204	81	378
Problem1	24	431	298	3906

define whether the hypothesis is accepted or rejected.

$$CMH(s) = \frac{N_{cf} - \frac{N_f N_c}{N}}{\left(\frac{N_f N_s N_c N_u}{N^2(N-1)}\right)} \quad (2)$$

However, it is possible for two or more statements to be assigned the same score. This necessitates the calculation of two different effectiveness scores for any spectrum-based fault localization technique: best-case effectiveness and worst-case effectiveness. The best-case effectiveness occurs when the faulty statement is examined first among all statements with identical scores. Conversely, the worst-case effectiveness occurs when the scores are identical, and the faulty statement is identified last among the statements.

To evaluate the effectiveness of PRCMHFL, we consider both of these effectiveness measures.

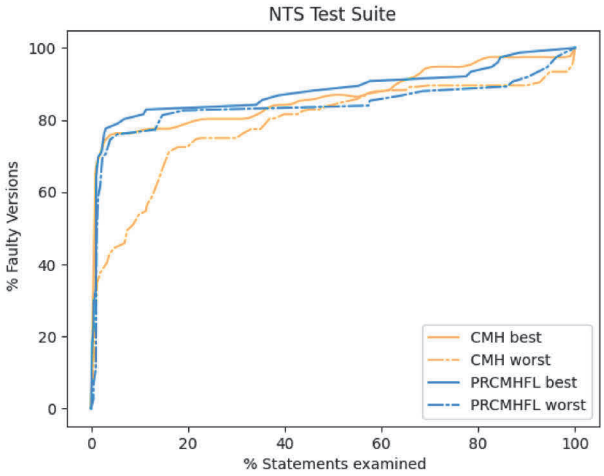
4.4. Results

To evaluate the accuracy and effectiveness of PRCMHFL, we conducted a performance comparison with other techniques. We employed the EXAM Score metric to highlight the strengths and weaknesses of our proposed approach. Since PRCMHFL represents an advancement in Spectrum-Based Fault Localization (SBFL) techniques, we compared it with five well-established techniques: Dstar, Zoltar, Barinel, and Ample. For additional information about these techniques, please refer to section 1 of this paper.

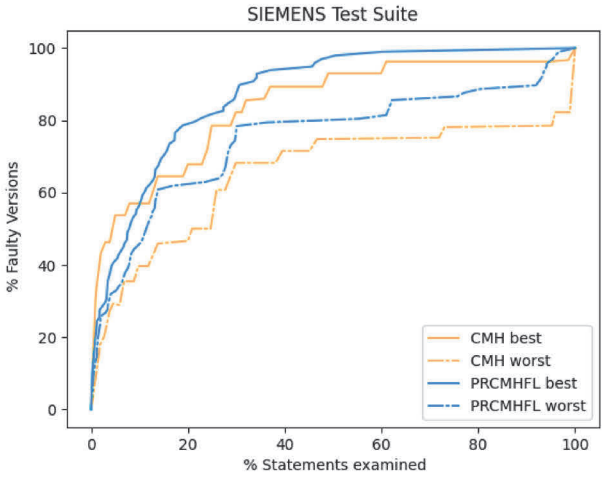
The performance of PRCMHFL, as measured by the EXAM Score metric in Eq. 1, is illustrated in Figures 2 to 6. In each graph, the horizontal axis represents the percentage of executable statements examined, while the vertical axis represents the percentage of detected faulty versions.

A point (x, y) on the graph signifies that y% of faulty versions are identified after examining code equivalent to x% of the executable statements in the program. The graphs compare our technique with an existing technique by showcasing both the best and worst effectiveness of our technique.

Specifically, PRCMHFL (best) required only 5% of code examination to localize 79% of faulty versions, while CMHFL (best) achieved 76%, as shown in Fig. 2. This represents a significant improvement in the efficiency of our technique.



(a)

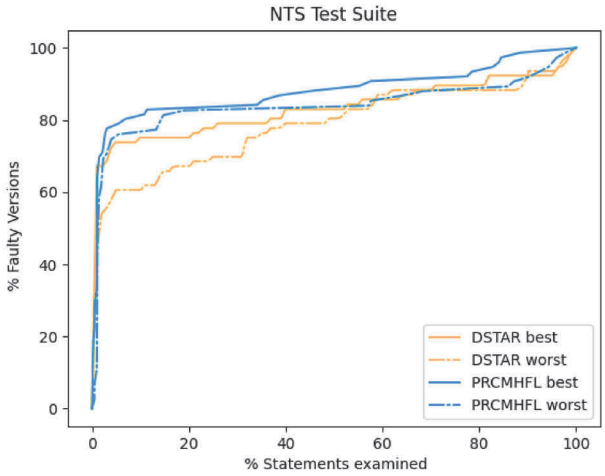


(b)

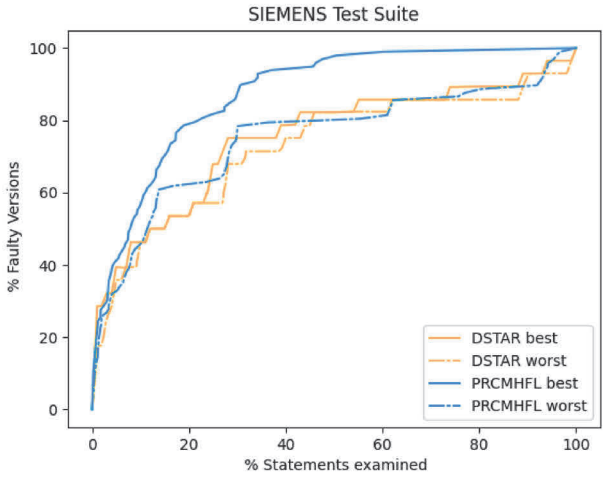
Figure 2: Effectiveness comparison of CMHFL and PRCMHFL

Furthermore, we observed a substantial improvement in the worst-case scenario of PRCMHFL compared to CMHFL[1]. PRCMHFL (worst) was able to localize 82.6% of faulty versions with only 20% of code examination, while CMHFL[1] (worst) only covered 72.1% of faulty versions. It is also noteworthy that our technique exhibits a smaller gap between the worst and best-case scenarios and that our worst-case scenario outperforms the best-case scenario of CMHFL[1] in certain cases.

In another comparative study using the NTS test suite, we found that PRCMHFL also outperformed Dstar and Barinel in localizing faulty versions. PRCMHFL (best) was able to localize 80% of faulty versions with only 6.1% examination of code, while Dstar (best) required 39% of code examination for the same result. Dstar (worst) accurately localize 63.1% of faulty versions with 20% of code examination. In comparison,

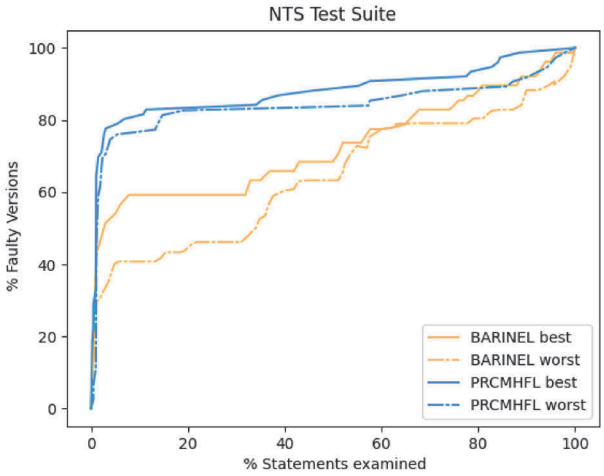


(a)

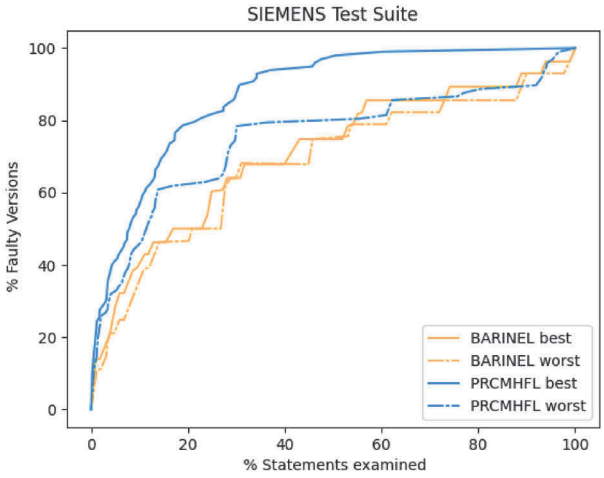


(b)

Figure 3: Effectiveness comparison of Dstar and PRCMHFL



(a)



(b)

Figure 4: Effectiveness comparison of Barinel and PRCMHFL

Barinel (best) and Barinel (worst) were able to localize 59% and 44.3% of faulty versions, respectively, while Ample (best) and Ample (worst) achieved 59% and 44.9%, respectively.

In addition to the NTS test suite, we also conducted experiments using the Siemens suite (as depicted in Fig. 4 to Fig. 5 SIEMENS test suite graphs). Our results demonstrated that our technique's best-case scenario outperformed the best cases of most existing techniques. Furthermore, due to the smaller gap between our best and worst-case scenarios, our technique consistently produced better results than most other techniques, even when compared to their best cases. This highlights the robustness and reliability of our technique in localizing faulty versions.

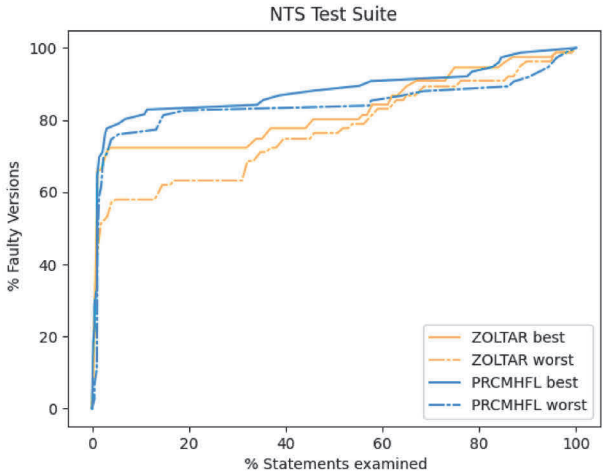
Overall, our results showcase the improved effectiveness and efficiency of PRCMHFL in localizing faulty versions compared to other techniques such as CMHFL[1], Dstar[17], Barinel[6], Zoltar[23] and Ample[21].

5. Comparison with Related Work

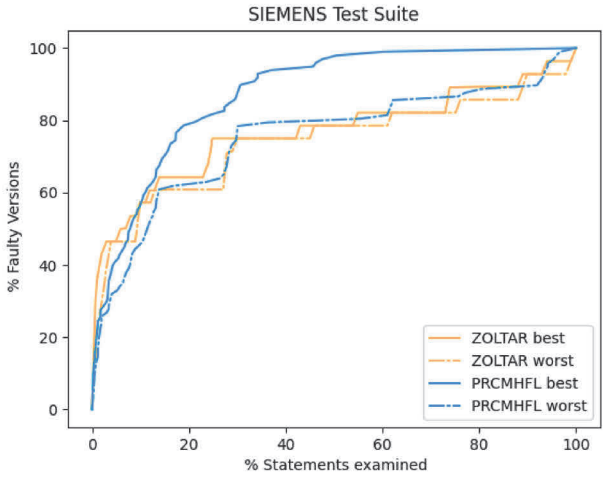
There are several existing slice-based Fault Localization techniques [9, 15, 24]. These techniques aim to reduce the size of the program by selecting only those statements that directly or indirectly affect the value of a variable at a given program point. They then attempt to localize the most prone target slice. However, this approach has limitations as it may not provide a complete view of the program code.

In contrast, our method, PRCMHFL, assigns a suspiciousness score to each and every executable statement in the program. This provides a more comprehensive and detailed view of the program code, allowing for more accurate fault localization.

We discussed about PRCMHFL and other spectrum-based fault localization techniques previously. Our findings show that our technique is good as compared to others. Dstar is a well-known technique in this domain. Our results establish PRCMHFL as a stronger technique among spectrum-based

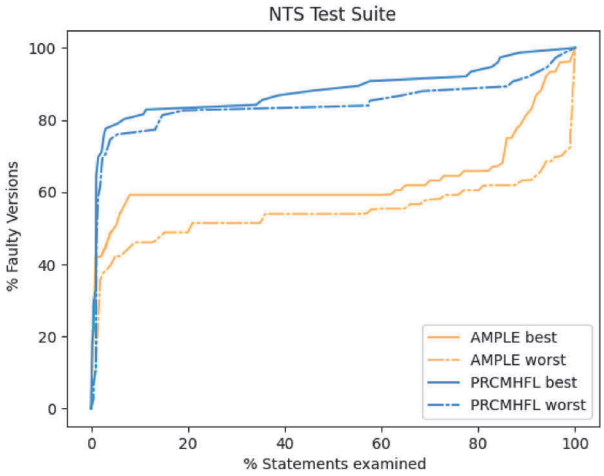


(a)

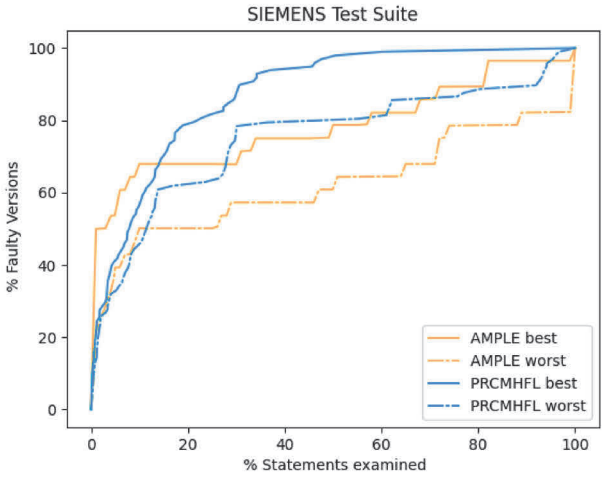


(b)

Figure 5: Effectiveness comparison of Zoltar and PRCMHFL



(a)



(b)

Figure 6: Effectiveness comparison of Ample and PRCMHFL

approaches and befit for standard for fault localization.
Neural Network-based techniques [18, 19, 26, 27, 28, 29, 30] have the potential to be effective in fault localisation. They have few drawbacks, including the need for extensive training time, parameter tuning, and the complexity of weight updates. On the other hand, our work PRCMHFL is efficient and effective solution. It requires significantly less time to execute when compared to Neural Network-based techniques. Also, it is a more practical and reliable option for Fault Localization.

6. Discussion

We proposed a statistical fault localization technique. This amalgamates the strengths of two existing methods, PRFL and CMHFL. Our aim was to enhance the efficiency of fault localization by reducing the amount of code examination required. PRFL and CMHFL have established themselves as valuable techniques in this context, and our approach builds upon their foundations to achieve even more promising results. Through extensive experimentation and evaluation, we have demonstrated the effectiveness of PRCMHFL in comparison to various well-established fault localization techniques. PRCMHFL consistently outperforms most of these methods, including Dstar, Zoltar, Ochiai, Barinel, and Ample, in terms of accuracy and efficiency. The ability of PRCMHFL to achieve high fault localization accuracy with a reduced code examination percentage sets it apart as a leading technique in the field.

However, our research does not end here, and there are several future insights and avenues for further exploration:

1. **Enhancing Scalability**: While PRCMHFL exhibits impressive performance on the datasets tested, further research can focus on making it even more scalable to accommodate larger software projects. This would require optimizing the algorithm to handle extensive codebases efficiently.

2. **Integration with Machine Learning**: Considering the suc-

cess of machine learning-based fault localization techniques, future work may involve integrating machine learning models into PRCMHFL to harness the power of predictive algorithms for even better results.

3. **Real-world Applications**: Expanding the evaluation to real-world software projects and scenarios would provide valuable insights into the practicality and adaptability of PRCMHFL in industrial settings.

4. **User-Friendly Tools**: Developing user-friendly tools and interfaces that implement PRCMHFL can facilitate its adoption by developers and testing teams, making it a more accessible and valuable asset in the software development process.

7. Conclusion

In conclusion, PRCMHFL represents a significant advancement in the domain of fault localization. Its ability to accurately pinpoint faults with reduced code examination and its potential for further improvements make it a promising choice for software developers seeking efficient and effective debugging solutions. As we continue to explore and refine this technique, we envision it playing a pivotal role in enhancing software quality and reliability in the future.

References

[1] Dharanappagoudar, R., Gupta, P., & Godboley, S. (2022, November). CMHFL: A new Fault Localization technique based on Cochran–Mantel–Haenszel method. In 2022 IEEE 19th India Council International Conference (INDICON) (pp. 1-7). IEEE.

[2] Godboley, S., & Dutta, A. (2021, December). PRFL: Predicate Rank based Fault Localization. In 2021 IEEE 18th India Council International Conference (INDICON) (pp. 1-6). IEEE.

[3] Godboley, S., Dutta, A., Mohapatra, D. P., Das, A., & Mall, R. (2016). Making a concolic tester achieve increased MC/DC. Innovations in systems and software engineering, 12(4), 319-332.

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

[5] Rani, G. M., & Godboley, S. (2022, April). Poster: A gCov based new profiler, gMCov, for MC/DC and SC-MCC. In 2022 IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 469-472). IEEE.

[6] Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2009, November). Spectrum-based multiple fault localization. In 2009 IEEE/ACM International Conference on Automated Software Engineering (pp. 88-99). IEEE.

[7] Agrawal, H., De Millo, R. A., & Spafford, E. H. (1991). An execution-backtracking approach to debugging. IEEE Software, 8(3), 21-26.

[8] Burnim, J., & Sen, K. (2008, September). Heuristics for scalable dynamic test generation. In 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (pp. 443-446). IEEE.

[9] Campos, J., Riboira, A., Perez, A., & Abreu, R. (2012, September). Gzoltar: an eclipse plug-in for testing and debugging. In Proceedings of the 27th IEEE/ACM international conference on automated software engineering (pp. 378-381).

[10] Choi, S. S., Cha, S. H., & Tappert, C. C. (2010). A survey of binary similarity and distance measures. Journal of systemics, cybernetics and informatics, 8(1), 43-48.

[11] Cleve, H., & Zeller, A. (2005, May). Locating causes of program failures. In Proceedings of the 27th international conference on Software engineering (pp. 342-351).

[12] Goodman, L. A., & Clogg, C. C. (1984). The analysis of cross-classified data having ordered categories. Harvard University Press.

[13] Jones, J. A., & Harrold, M. J. (2005, November). Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (pp. 273-282).

[14] Jones, J. A., Harrold, M. J., & Stasko, J. (2002, May). Visualization of test information to assist fault localization. In Proceedings of the 24th international conference on Software engineering (pp. 467-477).

[15] Naish, L., Lee, H. J., & Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. ACM Transactions on software engineering and methodology (TOSEM), 20(3), 1-32.

[16] Renieres, M., & Reiss, S. P. (2003, October). Fault localization with nearest neighbor queries. In 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings. (pp. 30-39). IEEE.

[17] Wong, W. E., Debroy, V., Gao, R., & Li, Y. (2013). The DStar method for effective software fault localization. IEEE Transactions on Reliability, 63(1), 290-308.

[18] Wong, W. E., Debroy, V., Golden, R., Xu, X., & Thuraisingham, B. (2011). Effective software fault localization using an RBF neural network. IEEE Transactions on Reliability, 61(1), 149-169.

[19] Wong, W. E., & Qi, Y. (2009). BP neural network-based effective fault localization. International Journal of Software Engineering and Knowledge Engineering, 19(04), 573-597.

[20] Xuan, J., & Monperrus, M. (2014, September). Learning to combine multiple ranking metrics for fault localization. In 2014 IEEE International Conference on Software Maintenance and Evolution (pp. 191-200) IEEE.

[21] V. Dallmeier, C. Lindig and A. Zeller, "Lightweight bug localization with ample", Proceedings of the sixth international symposium on Automated analysis-driven debugging, pp. 99-104, 2005.

[22] R. Abreu, P. Zoeteweij and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization", Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION 2007. TAICPART-MUTATION 2007, pp. 89-98, 2007.

[23] T. Janssen, R. Abreu and A. J. C. van Gemund,

"Zoltar: A Toolset for Automatic Fault Localization," 2009 IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, 2009, pp. 662-664, doi: 10.1109/ASE.2009.27.

[24] Eric Wong, Tingting Wei, Yu Qi and Lei Zhao, "A crosstab-based statistical method for effective fault localization", 2008 1st international conference on software testing verification and validation, pp. 42-51, 2008.

[25] Jiang, J., Wang, R., Xiong, Y., Chen, X., & Zhang, L. (2019, November). Combining spectrum-based fault localization and statistical debugging: An empirical study. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 502-514). IEEE.

[26] Zhang, Z., Lei, Y., Tan, Q., Mao, X., Zeng, P., & Chang, X. (2017). Deep learning-based fault localization with contextual information. IEICE TRANSACTIONS on Information and Systems, 100(12), 3027-3031.

[27] Tan, P. N., Steinbach, M., & Kumar, V. (2016). Introduction to data mining. Pearson Education India.

[28] W. Eric Wong, Yu Qi, BP neural network-based effective fault localization, Int. J. Softw. Eng. Knowl. Eng. 19 (04) (2009) 573–597.

[29] Zheng, W., Hu, D., & Wang, J. (2016). Fault localization analysis based on deep neural network. Mathematical Problems in Engineering, 2016.

[30] Dutta, A., Manral, R., Mitra, P., & Mall, R. (2019). Hierarchically localizing software faults using DNN. IEEE Transactions on Reliability, 69(4), 1267-1292.

[31] Hong, S., Lee, B., Kwak, T., Jeon, Y., Ko, B., Kim, Y., & Kim, M. (2015, November). Mutation-based fault localization for real-world multilingual programs (T). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 464-475). IEEE.

[32] Weiser, M. (1984). Program slicing. IEEE Transactions on software engineering, (4), 352-357.

[33] Lyle, R. (1987). Automatic program bug location by program slicing. In Proceedings 2nd international conference on computers and applications (pp. 877-883).

[34] Korel, B., & Laski, J. (1988). Dynamic program slicing. Information processing letters, 29(3), 155-163.

[35] P. K. Singh, S. Garg, M. Kaur, M. S. Bajwa and Y. Kumar, "Fault localization in software testing using soft computing approaches," 2017 4th International Conference on Signal Processing, Computing and Control (ISPCC), Solan, India, 2017, pp. 627-631, doi: 10.1109/ISPCC.2017.8269753.

[36] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, Rasheed Abubakar Rasheed, Multiple fault localization of software programs: A systematic literature review, Information and Software Technology, Volume 124, 2020, 106312, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2020.106312.

[37] Aranega, Vincent, Jean-Marie Mottu, Anne Etien, and Jean-Luc Dekeyser. "Traceability Mechanism for Error Localization in Model Transformation." In ICSOFT (1), pp. 66-73. 2009.

[38] Y. Li, S. Wang and T. Nguyen, "Fault Localization with Code Coverage Representation Learning," 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, ES, 2021, pp. 661-673, doi: 10.1109/ICSE43902.2021.00067.

[39] GCC, "GNU gcov - a Test Coverage Program." https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Accessed October 5, 2023.

[40] Godboley, S., Mohapatra, D.P. (2022). Towards Agile Mutation Testing Using Branch Coverage Based Prioritization Technique. In: Przybyłek, A., Jarzębowicz, A., Luković, I., Ng, Y.Y. (eds) Lean and Agile Software Development. LASD 2022. Lecture Notes in Business Information Processing, vol 438. Springer, Cham. https://doi.org/10.1007/978-3-030-94238-0_9

[41] Agarwal, S., Godboley, S., Krishna, P.R. (2022). Cyclomatic Complexity Analysis for Smart Contract Using Control Flow Graph. In: Panda, S.K., Rout, R.R., Sadam, R.C., Rayanoothala, B.V.S., Li, KC., Buyya, R. (eds) Computing, Communication and Learning. CoCoLe 2022. Communications in Computer and Information Science, vol 1729. Springer, Cham. https://doi.org/10.1007/978-3-031-21750-0_6

[42] Godboley S. and Krishna P. (2023). SmartMuVerf: A Mutant Verifier for Smart Contracts. In Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE, ISBN 978-989-758-647-7, SciTePress, pages 346-353. DOI: 10.52200011822200003464

[43] Monika Rani Golla and Sangharatna Godboley. 2023. GMutant: A gCov based Mutation Testing Analyser. In Proceedings of the 16th Innovations in Software Engineering Conference (ISEC '23). Association for Computing Machinery, New York, NY, USA, Article 22, 1–5. https://doi.org/10.1145/3578527.3578546