CMHFL: A new Fault Localization technique based on Cochran–Mantel–Haenszel method

Rutwik Dharanappagoudar
Department of CSE
National Institute of Technology
Warangal, Telangana, India
rd921872@student.nitw.ac.in

Palash Gupta
Department of CSE
National Institute of Technology
Warangal, Telangana, India
pg841822@student.nitw.ac.in

Sangharatna Godboley
Department of CSE
National Institute of Technology
Warangal, Telangana, India
sanghu@nitw.ac.in

***Abstract*—As modern software systems are large and complex, manually debugging software has become excessively expensive, time-consuming, boring, and error-prone. We present a modified Cochran–Mantel–Haenszel statistical method that utilizes test results along with statement coverage information in order to compute the suspiciousness score (likelihood of containing a fault) of each statement and generate a ranked list of statements. Results show that on an average, our proposed approach outperforms in contrast to other Fault Localisation (FL) techniques, such as Zoltar, Dstar, Ochiai, Barinel, and Ample by 38.994%.**

***Index Terms*—Fault Localization, SBFL, Program Spectra, Debugging Aids**

I. Introduction

Software Testing and Debugging is an important and expensive phase in Software Development [11]–[14], [16] . Fault localization (FL) technique is the most difficult, time-taking, and costly activity in program debugging task for discovering the location of faults in a program [2]. Locating faults manually when failures occur is becoming impractical. As a result, there is a huge demand for FL techniques that can locate program flaws with minimal human interaction.
Numerous automated FL systems have been documented in past [4], [6], [8], [15], [19], [22], [25] to reduce the time and expenses required to locate problems. The majority of existing FL approaches rely on heuristics. These techniques take information about line coverage and test execution outcome as input and produce a prioritized list of lines based on their likelihood of containing a bug. Static, dynamic, and execution slice-based approaches, statistics-based methods, model-based methods, spectrum-based methods, and so on are the basic categories of fault localization techniques. These models share a similar goal, i.e., finding the location of faults. SBFL has grown in popularity as a result of its independence from the system model and ease of implementation. Its diagnostic accuracy, on the other hand, is fundamentally limited.
Current fault localization techniques currently entail that 40-50% of the code be examined, regardless of program size. Additionally, these methods rely on intuitive guesses or developer knowledge. Inspired by FTFL [7], we apply the concept of a contingency table [9], [17] to compute the suspiciousness for every statement, as opposed to existing intuitive guesses or heuristics-based FL approaches. To establish the correlation between the execution of a statement and the test execution results (i.e., pass or fail), we use Cochran–Mantel–Haenszel (CMH) [20] test statistics. A ranked list is then generated, where the statements are ranked according to the suspiciousness score previously calculated.
The rest of the paper is categorised as follows: Section 2 presents the details of CMHFL. Section 3 shows the experimental study followed by comparison with related work in Section 4. Finally, we conclude with insight future work in Section 5.

978-1-6654-4175-9/21/$31.00 ©2021 IEEE

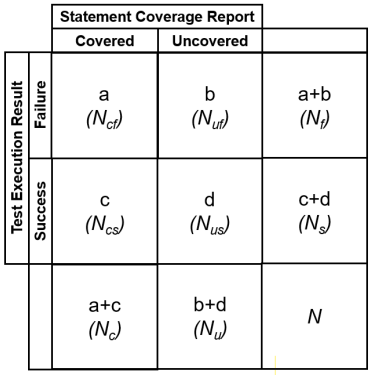


Figure 1: Contingency table in the context of fault localisation

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

N	#TCs
N_s	#Passed TCs
N_f	#Failed TCs
$N_c(s)$	#TCs executing s
$N_u(s)$	#TCs not executing s
$N_{cp}(s)$	#Passed TCs executing s
$N_{cf}(s)$	#Failed TCs executing s
$N_{up}(s)$	#Passed TCs not executing s
$N_{uf}(s)$	#Failed TCs not executing s

II. Proposed Approach

Assume that s is a statement in a program P, which is executable. Our objective is to compute the suspiciousness

score for s which depicts the likelihood of it containing a bug. The meanings of the symbols that are utilized for our technique are listed in Table I

A. Overview

For every statement in the program, a contingency table is built as shown in Fig. 1. This two-dimensional (columns) table show the values that indicate the number of successful and failed test cases. The two-dimensional (rows) table values indicate the number of statements that are covered and uncovered by the corresponding successful and failed test cases. We perform a hypothesis test for each contingency table to establish a relationship for program execution and the coverage results for s. We establish a null hypothesis that states "The execution result of program is independent of the invocation or coverage of an executable statement". The CMH Test, stated in Eq. 1 [20] to define whether the hypothesis is accepted or rejected.

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

For a significance level of α, if the value of $CMH(s)$ is greater than α, the null hypothesis will be rejected, this implies that the program execution results are dependent on the coverage results. However, if the value of $CMH(s)$ is less that α, it indicates that the execution results are independent of the coverage results. Our main purpose is to compute the degree of association for execution and coverage results of s, instead of finding dependency of the two variables. We later use this degree of association as a score for each s in the faulty program and assign a rank to the statement based on its computed suspiciousness score.

B. Explanation

These constituent units are presented in Fig. 2. We discuss the three important units that concern with the implementation of CMHFL technique namely Output Comparator, Program Spectra Generator and Statement Ranking Generator.

1) Output Comparator: The flow of implementation starts with providing the test suite to the Oracle (source or fault-free program) and the faulty version (mutant) of the same program. Both of them will then generate the outputs for the provided test suites. The Output Comparator takes the output results obtained from the Oracle (Expected Output) and the faulty program (Actual Output) as inputs. The test case is successful, if both outputs of faulty or original versions of program matches. Otherwise, the test case is known as a failed test case. This constitutes the Test Execution Results which is the output of the Output Comparator.

2) Program Spectra Generator: At the time of generating the actual output for the Output Comparator, the faulty program also generates the statement coverage report for the provided test suite. The Program Spectra Generator takes the statement coverage results obtained from the mutant and the test execution results obtained from the Output Comparator

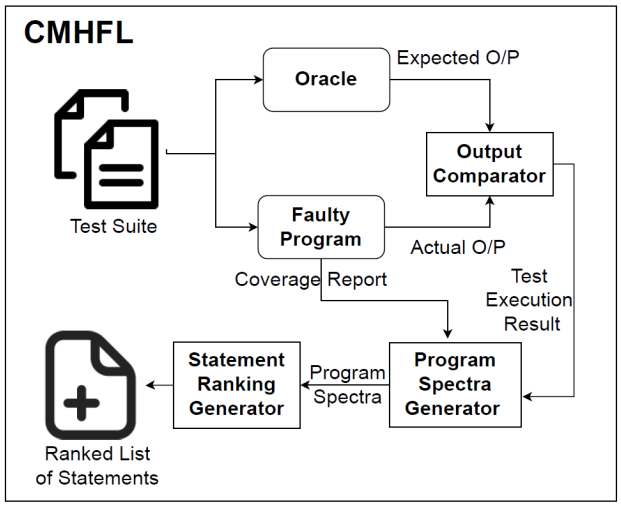


Figure 2: Schematic representation of CMHFL

as inputs. It generates the contingency table metric discussed in 1 for each s. The Spectra thus obtained would contain information regarding the variables viz. N_{cs}, N_{cf}, N_{us} and N_{uf} that lay the foundation for computing the CMH test score. The output is then forwarded to the Statement Ranking Generator.

3) Statement Ranking Generator: Statement Ranking Generator utilizes the Program Spectra information to assign a suspiciousness score to all executable statements of the faulty program. This score for a statement s is equivalent to the CMH test score which is computed using the formula mentioned in equation 1. More the suspiciousness score for a statement, the more is its likelihood of containing a fault. The Statement Ranking Generator assigns a rank to every s in the input faulty program and thus derives a list with rankings of Statements which is the final output obtained from the CMHFL technique. The developers can thus use the Ranked List obtained which would therefore help them in localizing the fault in their buggy code.

III. Experimental Study

In this section, we discuss experimental study in detail.

A. The Set-Up

We used 64-bit Ubuntu virtual machine with 10GB RAM. The benchmark programs are in C format.GCOV[1] is an open-source code-coverage tool and it is used in conjunction with GCC to test code coverage in the subject C programs. We have implemented our prototype tool which named as CMHFL tool.

B. Benchmarks

To show the effectiveness of CMHFL, we performed experiments for total eleven programs. The first five programs are taken from the NTS repository[2] and the subsequent

[1]https://man7.org/linux/man-pages/man1/gcov-tool.1.html
[2]https://github.com/ArpitaDutta/NTS_Repository

Table II: Program Characteristics

S.No.	Program	#Flty. Ver.	LOCs	#Exec. Lines	#Test Cases
1.1	Print_Tokens	7	565	195	4130
1.2	Print_Tokens2	10	510	200	4115
1.3	Schedule	9	412	152	2650
1.4	Schedule2	10	307	128	2710
1.5	Tcas	41	173	65	1608
1.6	Tot_Info	23	406	122	1052
2.1	quick_sort	6	99	59	128
2.2	cfg_test	25	93	46	55
2.3	merge2BSTree	8	226	93	197
2.4	nextDate	14	204	81	378
2.5	Problem1	24	431	298	3906

Table III: Types of Mutants with scores

Program	ROR	LOR	AOR	CM	CC	CA	VC	SI	DC	BOR	PVR
Print_Tokens	0	0	0	49.1	32.2	47.6	0	0	0	0	0
Print_Tokens2	0	0	0	29.4	41.6	71.4	0	0	0	0	0
Schedule	0	0	0	26.5	36.1	37.7	60.6	0	0	0	0
Schedule2	10	0	0	67.3	0	7.4	8	0	0	0	0
Tcas	23.6	13.7	3.7	19.5	18	9.9	12.2	0	0	0	0
Tot_Info	18.1	0	5	32	22.6	8.3	0	1.6	1.2	0	0
cfg_test	42.8	0	0	37.7	0	0	12.4	1.1	0	0	0
merge2BSTree	12.5	0	0	16.2	0	0	21.9	0	0	0	0
nextDate	15.3	9	19.4	8	18.6	0	0	1.7	0	0	0
Problem1	4.3	72.7	0	0	37.6	0	0	0	0	0	0
quick_sort	66.6	0	0	0	0	0	2.7	0	0	0	0

six programs are taken from the Siemens suite which is downloaded from the SIR Repository[3]. Table II displays the detailed characteristics of the programs, such as Program Name, number of fault versions, total executable lines of code and the number of test cases can be found here. The Siemens suite is a standard benchmark to investigate on fault localization techniques [7], [19].

NTS repository consists of five C-programs [7]. The cfg_test program is taken from the CREST [3] benchmark suite. Problem1 is taken from the RERS 2018 [24].

Table III shows the types of mutants / faults used in our work. It presents a percentage of mutants for a specific type of mutant in the faulty versions. Information regarding Data-type Change (DC), Sign Inversion (SI), Arithmetic Operator Replacement (AOR), Code Addition (CA), Constant Change (CC), Pointer Value Replacement (PVR), Code Missing (CM), Logical Operator Replacement (LOR), Variable Change (VC), Relational Operator Replacement (ROR), and Bitwise Operator Replacement (BOR) faults are presented.

C. Evaluation Metric

In this section, we show the useful metrics used in CMHFL.

EXAM Score: EXAM Score [23] to compute the effectiveness of fault localization. It is computed using Eq. 2. It computes the score of statements that need to be examined to localize the fault.

$$Exam\_Score = \frac{|S_{examined}|}{|S_{Total}|} * 100 \tag{2}$$

where, $|S_{examined}|$ indicates the number of statements examined to localize the fault while $|S_{total}|$ indicates the total number of executable statements. If EXAM Score is lower, it means the technique is more effective.

Average Improvement: The Average improvement achieved for one FL technique over another technique is measured using Eq. 3

$$IA_{A,B} = \frac{avgES_B - avgES_A}{avgES_A} * 100 \tag{3}$$

where $IA_{A,B}$ indicates the improvement achieved for technique A over technique B. $avgES_A$ and $avgES_B$ present the average EXAM_Score computed by techniques A and B respectively. If EXAM_Score is less then it means the technique is better.

As previously stated, a Spectrum-Based FL technique gives a suspiciousness score to each statement of the program. The score of a statement is calculated by a function involving four variables, viz. N_{cf}, N_{cs}, N_{uf} and N_{us} as discussed in Table I. However, the sore assigned can be identical for two or more statements. This demands for the computation of two different effectiveness scores for any SBFL technique, namely the best case effectiveness and the worst case effectiveness. If the faulty statement is examined first among all the statements that have identical score, then it is termed as best case effectiveness. Similarly, the worst case effectiveness is the case if the faulty statement is examined last among the statements with identical scores. Hence, to measure the effectiveness of CMHFL, we use the best case and the worst case effectiveness.

D. Results

We compare the performance results of CMHFL with other FL techniques. By comparing the metric scores we show the effectiveness of CMHFL with other FL techniques, followed by the discussion of the cases where our proposed approach is likely to fail. We also discuss the results of CMHFL with five other FL techniques. All the five techniques are taken from the SBFL family, viz. DStar [26], Zoltar [4] Ochiai [21], Barinel [1] and Ample [1]. DStar [26] is considered to be the best fault localization technique and a state-of-the-art technique. Ochiai [21] has been reported to be more effective than many other SBFL and non-SBFL techniques. Zoltar, Barinel and Ample are other widely used and well established SBFL techniques.

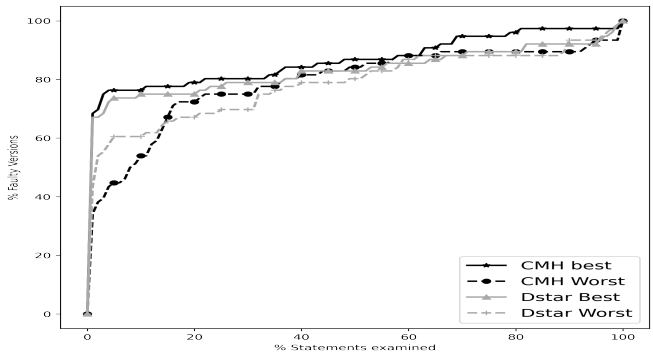
Figures 3a to 7b highlight the performance of CMHFL and other FL techniques on the basis of EXAM_Score and the Average Improvement metrics. For each graph, the x-axis represents the percentage of statements examined which are executable and y-axis represents the percentage of faulty versions detected. Any coordinate point (x,y) in the graph indicates that y% of faulty versions are localized after examining code less than or equal to x% of the total executable statements. Each graph compares one of the five techniques we have taken into consideration with our CMHFL technique using the best case and worst case (effectiveness) of approach. Fig. 3a presents the comparative result between CMH

[3]https://sir.csc.ncsu.edu/portal/index.php

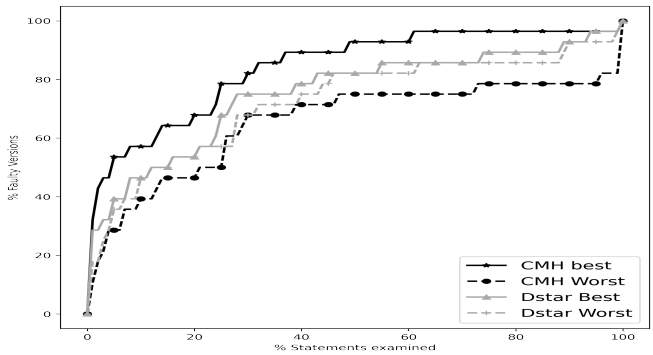
and DStar using NTS program suite. We observed that CMH(Best) achieves 5% of code examination to localise 76% of faulty versions. On the otherhand DStar(Best) achieves 72%. On average, CMH(Best) and CMH(Worst) produce an average improvement of 14.49% and 0.77% in contrast to DStar(Best) and DStar(Worst) respectively. Fig. 3b presents the comparative result between CMH and DStar using Siemens test suite. We observed that CMH(Best) achieves 20% of code examination to localise 67.86% of faulty versions. On the otherhand DStar(Best) localises 53.57%. On average, CMH(Best) produces an average improvement of 53.03% in contrast to DStar(Best). However, CMH(Worst) turns out to be less effective than DStar(Worst) by 23.73%. Overall effectiveness of CMH is concluded higher than the DStar.

Fig. 4a presents the comparative result between CMH and Zoltar using NTS program suite. We observed that CMH(Best) and CMH(Worst) achieve 20% of code examination to localise 78.94% and 72.36% of faulty versions respectively. On the otherhand Zoltar(Best) and Zoltar(Worst) localize the bugs in 72.36% and 63.15% respectively. An average improvement of 29.96% and 10.34% over the Zoltar(Best) and Zoltar(Worst), respectively. Fig. 4b presents the comparative result between CMH and Zoltar using Siemens test suite. We observed that CMH(Best) and CMH(Worst) achieve 25% of code examination to localise 78.57% and 60.71% of faulty versions respectively. On the otherhand Zoltar(best) and Zoltar(Worst) localise 75.00% and 60.71% respectively. On average, CMH(Best) is 44.07% more effective than Zoltar(Best) while it is 27.69% less effective than the Zoltar(Worst).

Figures 5a, 6a, and 7a, respectively, present the comparative results of CMH with Ochiai, Barinel and Ample for the NTS program suite. CMH(Best) produces an average improvement of 31.25%, 101.1% and 165.06%, respectively, over the Ochiai(Best), Barinel(Best) and Ample(Best), while CMH(Worst) produces an average improvement of 11.23%, 68.79% and 114.50% was noticed over the Ochiai(Worst), Barinel(Worst) and Ample(Worst) respectively. Figures 5b and 6b present the comparative results for CMH using Ochiai and Barinel, respectively, for the Siemens program suite. The observed improvement of CMH(Best) over Ochiai(Best) and Barinel(best) is 56.88% and 78.42%, respectively. On the other hand, CMH(Worst) proves to be less effective than Ochiai(Worst) and Barinel(Worst) by 22.01% and 12.39%, respectively. As per above discussion, the significant difference between best and worst cases result for a overall better performance for CMH over Ochiai and Barinel techniques. Figure 7b presents the comparative results for CMH and Ample using Siemens suite. The average improvement of CMH(Best) and CMH(Worst) over Ample(Best) and Ample(worst), respectively, is 21.79% and 9.77% respectively. Notice that CMH(Worst) is more effective than the Ample(Worst) unlike all the results observed above for the Siemens suite. Studying the results derived from Figures 3a, 4a, 5a, 6a, and 7a, for the NTS program suite, marginal improvements were noticed



(a) NTS test suite



(b) Siemens test suite

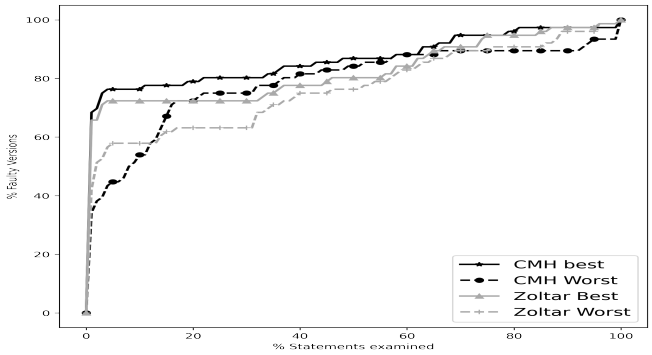
Figure 3: Effectiveness of CMHFL and Dstar

over the DStar technique but a higher percentage could be noticed over the Zoltar, Ochiai, Barinel and Ample techniques confirming the high effectiveness of the CMH technique.

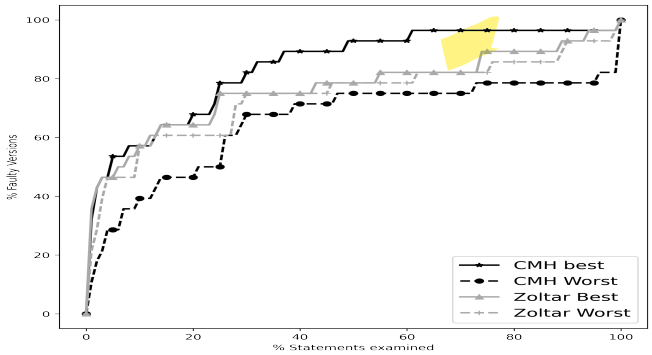
Studying the results derived from figure 3b, 4b, 5b, 6b, and 7b, using Siemens suite, we can deduce that a higher percentage of improvement was observed for the best case ranking, while the worst case ranking predominantly produced unfavorable results for our technique. However, the overall effectiveness of CMH still remains better than all the five techniques mentioned previously, because the improvements observed in the best case approach were significantly higher than the negative results in the worst case approach. As per our result discussion we conclude that for all the considered programs, CMHFL yields better results as compared to DStar, Zoltar, Ochiai, Barinel and Ample.

IV. Comparison with related work

A wide range of slice-based techniques for FL have been proposed in the past [4], [21], [26]. Since slices are lengthy and complex to understand, these FL techniques are not much useful. Most of the times a slice may not contain the fault, so examining those statement in not effective. Ranks are not being assigned to the statements in case of Slice-based techniques. On the contrary, CMHFL assigns a suspiciousness score to all the executable statements in the program and ranks all of them ruling out the possibility of missing the bug as in the case of slice-based techniques.

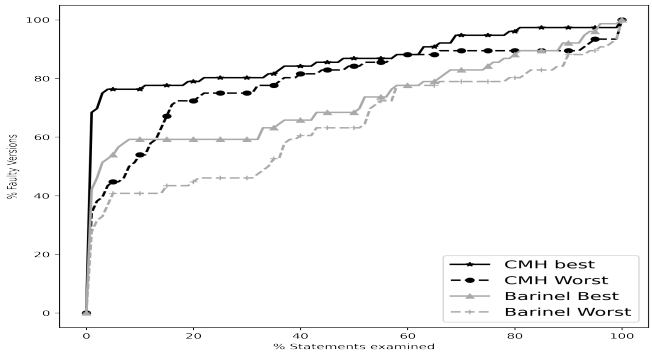


(a) NTS test suite

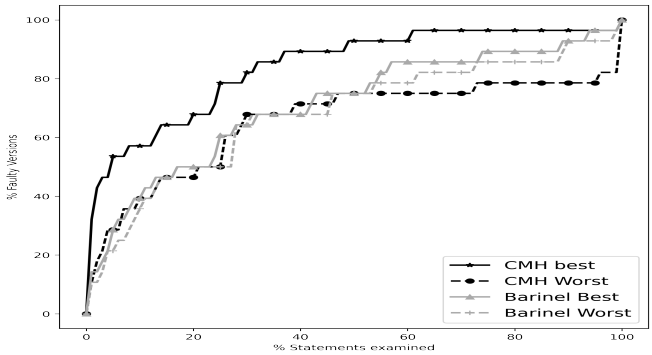


(b) Siemens test suite

Figure 4: Effectiveness of CMHFL and Zoltar

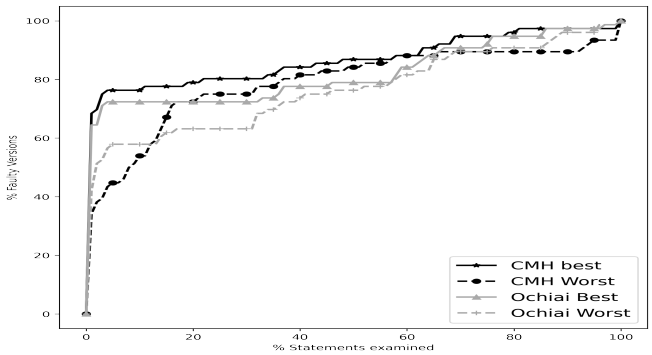


(a) NTS test suite

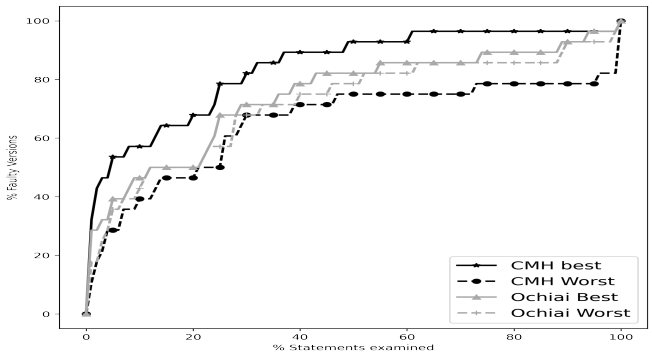


(b) Siemens test suite

Figure 6: Effectiveness of CMHFL and Barinel

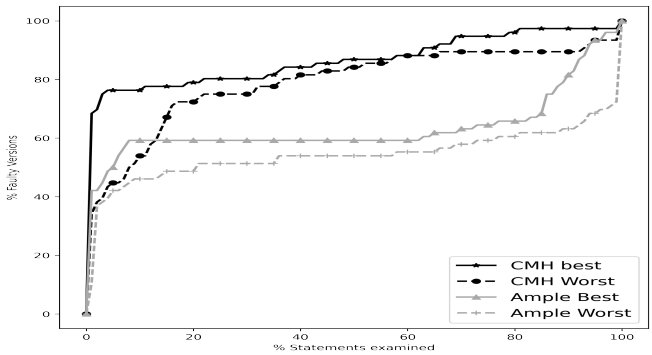


(a) NTS test suite

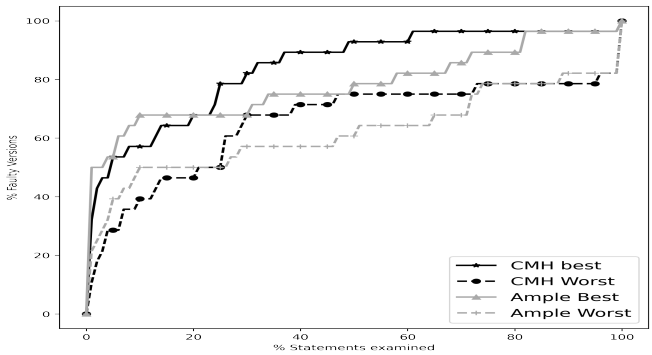


(b) Siemens test suite

Figure 5: Effectiveness of CMHFL and Ochiai



(a) NTS test suite



(b) Siemens test suite

Figure 7: Effectiveness of CMHFL and Ample

We have already compared CMHFL with other Spectrum-based FL techniques and presented the results in Section III-D. DStar [26] is considered to be the state-of-the-art for FL techniques and our results outperform the effectiveness achieved by DStar. Ochiai has been reported to be more effective than Tarantula [18], [19] which is another Spectrum-based FL technique. Tarantula is a highly established FL technique and it is known to outperform cause-transition [5], set intersection [18], set union and nearest neighbor [23] techniques of FL. It is known that CMH achieves better effectiveness than Ochiai, which implies that CMH outperforms Tarantula as well.

BPNN [28] and RBFNN [27] are eminent neural network based techniques. The run-time for any Spectrum-based FL technique including CMH is less expense where the time consumption is very less (order of seconds), whereas NN based approaches consume time (order of minutes). For each faulty version of the program, NN based methods demand additional time for training the model. As per above, CMHFL is much more efficient than NN based techniques because of the massive differences in the execution time of both the techniques. Especially for large-sized programs, CMH is faster which makes it more applicable as compared to BPNN and RBFNN.

V. Conclusions and Future Work

Our new technique makes use of the program execution results and the code coverage report to generate Spectra information which is later utilized to produce a list showing the rankings of statements according to their computed suspiciousness score. Through the use of EXAM_Score and Average improvement metrics, we have compared CMH with five other prominent SBFL techniques and observed an improvement of 38.994% on average.

In future the effectiveness of CMHFL can be improved by combining it with Predicate Rank-Based Fault Localization (PRFL) [10] which is an existing FL technique that ranks the predicates in a buggy program.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, 2009.

[2] Hiralal Agrawal, Richard A. De Millo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[3] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008.

[4] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, page 378–381, New York, NY, USA, 2012. Association for Computing Machinery.

[5] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 342–351. IEEE, 2005.

[6] Arpita Dutta and Sangharatna Godboley. Msfl: A model for fault localization using mutation-spectra technique. In *International Conference on Lean and Agile Software Development*, pages 156–173. Springer, 2021.

[7] Arpita Dutta, Krishna Kunal, Saksham Sahai Srivastava, Shubham Shankar, and Rajib Mall. Ftfl: A fisher's test-based approach for fault localization. *Innovations in Systems and Software Engineering*, 17(4):381–405, Dec 2021.

[8] Arpita Dutta, Saksham Sahai Srivastava, Sangharatna Godboley, and Durga Prasad Mohapatra. Combi-fl: Neural network and sbfl based fault localization using mutation analysis. *Journal of Computer Languages*, 66:101064, 2021.

[9] Brian S Everitt. *The analysis of contingency tables*. CRC Press, 1992.

[10] Sangharatna Godboley and Arpita Dutta. Prfl: Predicate rank based fault localization. In *2021 IEEE 18th India Council International Conference (INDICON)*, pages 1–6, 2021.

[11] Sangharatna Godboley, Arpita Dutta, Durga Prasad Mohapatra, Avijit Das, and Rajib Mall. Making a concolic tester achieve increased mc/dc. *Innovations in systems and software engineering*, 12(4):319–332, 2016.

[12] Sangharatna Godboley, Arpita Dutta, Durga Prasad Mohapatra, and Rajib Mall. J3 model: a novel framework for improved modified condition/decision coverage analysis. *Computer Standards & Interfaces*, 50:1–17, 2017.

[13] Sangharatna Godboley, Arpita Dutta, Durga Prasad Mohapatra, and Rajib Mall. Gecojap: A novel source-code preprocessing technique to improve code coverage. *Computer Standards & Interfaces*, 55:27–46, 2018.

[14] Sangharatna Godboley, Arpita Dutta, Durga Prasad Mohapatra, and Rajib Mall. Scaling modified condition/decision coverage using distributed concolic testing for java programs. *Computer Standards & Interfaces*, 59:61–86, 2018.

[15] Sangharatna Godboley, Joxan Jaffar, Rasool Maghareh, and Arpita Dutta. Toward optimal mc/dc test case generation. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 505–516, 2021.

[16] Sangharatna Godboley, Durga Prasad Mohapatra, Avijit Das, and Rajib Mall. An improved distributed concolic testing approach. *Software: Practice and Experience*, 47(2):311–342, 2017.

[17] Leo A Goodman, Clogg C Clifford, and Clifford C Clogg. *The analysis of cross-classified data having ordered categories*. Harvard University Press, 1984.

[18] J.A. Jones, M.J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477, 2002.

[19] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[20] The CMH Method. https://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704-ep713_confounding-em/BS704-EP713_Confounding-EM7.html, 2016.

[21] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3), aug 2011.

[22] Golla Monika Rani and Sangharatna Godboley. Poster: A gcov based new profiler, gmcov, for mc/dc and sc-mcc. In *15th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 469–472, 2022.

[23] Manos Renieres and Steven P Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE, 2003.

[24] Rigorous examination of reactive systems (rers-2018): Sequential training problems for rers 2018, 2018.

[25] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[26] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.

[27] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2012.

[28] W. ERIC WONG and YU QI. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.