Fault Localization with DNN-based Test Case Learning and Ablated Execution Traces

Takuma Ikeda[1,†], Kozo Okano[1,†], Shinpei Ogata[1,†] and Shin Nakajima[2,‡]

[1]*Shinshu University, Nagano, Japan*
[2]*National Institute of Informatics, Tokyo, Japan*

Abstract

Automatic fault localization is a technique that helps reduce the costly task of program debugging. Among the existing approaches, Spectrum-based fault localization (SFL) shows promising results in terms of scalability. SFL calculates the suspiciousness scores of each spectrum (statement or expression of codes) from the coverage information obtained with test cases. This paper considers the fault localization problem from a new perspective. Our key idea is to examine the impact of missing spectrum from which we obtain useful information for locating faults, while SFL basically relies on the information extracted from executed spectrums. Executing programs with a certain spectrum removed requires a novel method of emulating the execution of incomplete programs. We adopt a machine learning test-case classifier that classifies test execution results into either Pass or Fail; the classifier is able to work on executions of either complete or incomplete programs. Evaluation experiments were conducted using open-source programs of different characteristics from three projects available on Defects4j. The paper includes discussions on the pros and cons of the proposed method by analyzing the experimental results for programs with different fault characteristics.

Keywords

machine learning classifier, test-case learning, Word2Vec, LSTM, attention

1. Introduction

In software development, fault localization is a costly task. Testing and debugging are reported to account for up to 75% of the development cost[1]. Automatic fault localization is an effective technique to reduce the cost of program debugging. Among the existing methods, Spectrum-based fault localization (SFL) has shown promising results in terms of scalability, light processing, and language independence[2][3][4].

Ochiai[5] and Tarantula[6] are representative SFL techniques. These calculate a failure suspiciousness score for each statement based on code coverages at test runtime. Statements with high scores are considered highly suspicious of failure, and developers investigate the program to locate faults starting with the high-scored statements.

Ochiai and Tarantula calculate a failure suspiciousness score based on the frequency with which each statement is executed in the Fail or Pass test cases. A statement that is frequently executed in a Fail test and rarely executed in a Pass test has a higher suspicion score than other

2nd International Workshop on Intelligent Software Engineering, 2023
22w2008a@shinshu-u.ac.jp (T. Ikeda); okano@cs.shinshu-u.ac.jp (K. Okano); ogata@cs.shinshu-u.ac.jp (S. Ogata); nkjm@nii.ac.jp (S. Nakajima)


statements. Therefore, if the fault is a statement for which the number of executions does not differ between Pass and Fail test runs, these techniques are unable to give a high suspiciousness score to the fault location. Indeed, these techniques do not give high suspiciousness scores to statements that are executed regardless of Pass or Fail, such as assignment statements. In this paper, we propose an approach effective for the types of faults for which conventional SFL techniques such as Ochiai cannot give a high suspiciousness score.

The first step in our approach is test case learning, which trains a DNN model to classify test executions into Pass or Fail. In order to improve the time for processing execution traces, we extended the test case learning method reported in Tsimpourlas et al[7]. The trained DNN model classifies every execution trace as Pass or Fail. The execution trace is represented as a set of executed statements generated from the code coverage information so that the DNN model learns execution patterns of statements in each test case. The DNN model takes as input a set of executed statements and outputs a single value of type float a confidence level indicating that the test result is Pass. Second, an ablated trace, which is a trace that systematically removes a statement from an execution trace, is input to the trained DNN model. If the removed statement is indeed a fault, the inference result of the DNN model is supposed to change because the execution trace does not contain any information about the fault execution. If the removed statement is not a fault, the information about the fault statement remains in the execution trace and thus does not significantly affect the inference result; the DNN model outputs a confidence value to indicate how much it believes the input is categorized as Pass. If a highly suspicious fault statement is removed in the ablated trace, then the DNN's output value becomes much higher than the case with the trace before the ablation.

We evaluated the proposed method on three projects (Math, Lang, Chart) available on Defects4j[8]. Then, Wilcoxon Signed-Rank test confirmed that the proposed method is able to identify faults with higher accuracy than the SFL techniques for programs with multiple faults. In addition, the proposed method achieved higher accuracy than the SFL approach when the fault is a statement such as an assignment statement, for which there is little difference in the number of execution frequencies between the Pass and Fail test cases. In comparison to Ochiai and Tarantula, our method reduced the amount of code investigation required to identify faults by up to 23 percentage points or more.

The rest of the paper is organized as follows. Section 2 describes the proposed method. Section 3 describes the experiment setup and Section 4 discusses the experiment results. Section 5 describes the related work. We conclude in Section 6.

2. Proposed Approach

The proposed fault localization method consists of four steps explained below in order.

1. Prepare execution traces, **2.** Train DNN model, **3.** Ablate execution traces, **4.** Identify faulty statements with the trained DNN model.

2.1. Prepare execution traces

The execution traces used in the proposed method are generated from code coverage information. We collected the code coverages of the SUT using OpenClover[9]. Figure 1 shows the procedure

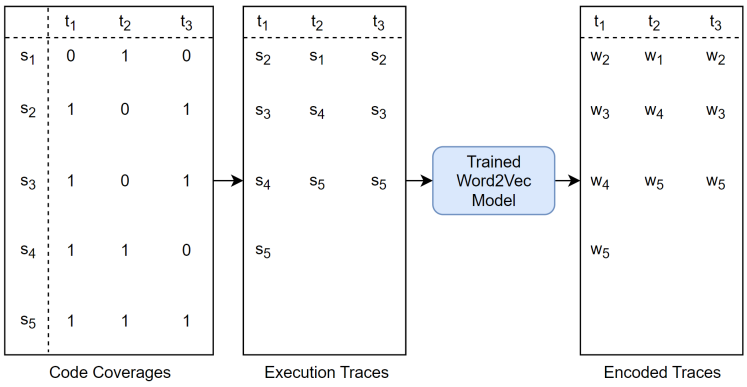

Figure 1: Create Execution Traces and Encoded Traces

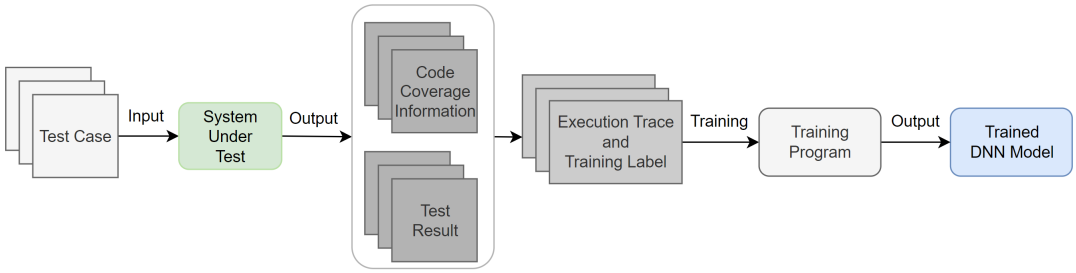

Figure 2: Overview of DNN Model Learning

for generating execution traces, which are sets of statements, and the procedure for encoding the execution traces. In Figure 1, t_1, t_2, t_3 are the test cases and $s_1, s_2, ..., s_5$ are statements in the source code. And $w_1, w_2, ..., w_5$ is the distributed representation in which each statement is encoded with Word2Vec.

In Figure 1, we identify the statements executed in each test case from the collected code coverage information and arrange each statement according to its line number in the source code. Next, Word2Vec[10] is used to encode the execution traces of $s_1, s_2, ..., s_5$. The statements in each test case $(s_1, s_2, ...s_5)$ are considered "words" in natural language, and their sequences are encoded as "sentences".

2.2. Train the DNN model

Figure 2 shows an overview of the DNN model training. The DNN model is trained using the collected execution traces as input and the test results as labels. It is assumed that the test results have been collected by the developer using a test oracle derived from a requirements document. Once training is complete, the DNN model can classify the test results of the input execution traces. Our DNN model architecture and its input are shown in Figure 3. The test results given to the DNN model are 0 for Fail and 1 for Pass; the higher value of the output of the DNN model, the greater the confidence that the classification result is Pass.

The DNN model of the existing approach[7] uses LSTM to encode both the statement in-

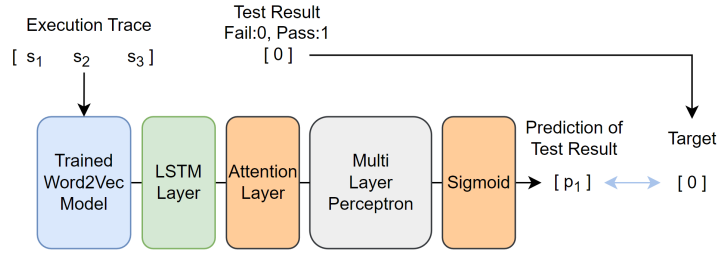

Figure 3: DNN Model Architecture

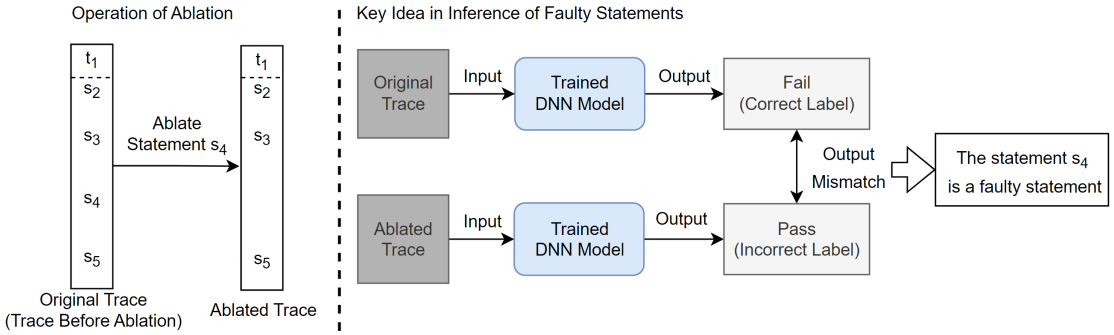

Figure 4: Operation of Ablation and Key Idea in Our Approach

formation and the execution traces, which requires an enormous computing time to process a large number of execution traces. We improved the processing time of execution traces without losing the classification performance. Our method uses Word2Vec[10] and LSTM together for encoding execution traces. Word2Vec is used to encode each statement information executed in the test case (s_1, s_2, and s_3 in Figure 3), and the distributed representation of Word2Vec is input to LSTM to encode the test case execution traces. We also used Attention[11] to improve classification performance.

2.3. Ablate execution traces

We define ablation as the removal of information about a particular statement from an execution trace. The operation of ablation is shown in Figure 4, the statement s_4 is targeted and the information about this statement is ablated. In our approach, ablated traces are incomplete program execution traces that skip the execution of a particular statement.

2.4. Identify faulty statements with the trained DNN model

The key idea of our approach is shown in Figure 4. An execution trace for which the execution result is Fail is selected as the target of ablation. If all statements executed in the Fail test are not included in the selected trace, we target all statements for ablation using other execution traces that result in Fail. In the case of Figure 4, the removal of information about s_4 from the original trace changes the classification result of the DNN model. Therefore, s_4 is considered to

have important information for the original trace to be classified as Fail, suggesting that s_4 is a statement of suspected fault.

2.5. Rationale

We summarize here the rationale of the proposed fault localization method. Our approach is to train a DNN model that inputs execution traces and classifies the test results. We expect that the DNN model learns the execution patterns and orders of statements in classifying Pass or Fail. We pick up a specific statement and remove information about this statement to observe how confident the predicted result is. If the ablated statement is a fault, the output of the DNN model is expected to be closer to Pass than the result before the ablation. We choose every statement that appeared in the Fail execution as a candidate for the ablation. The ablated traces are input to the DNN model one by one. The statement that makes the DNN model output closest to Pass (i.e., the output is larger) is the most suspicious. We used the descending order of the output values of the ablated traces as a rank of fault suspicion in order to evaluate the accuracy of our method in locating faults.

3. EVALUATION EXPERIMENT

3.1. Research Question

The following research questions are investigated in the evaluation experiment.

RQ 1. Comparison of the proposed method and existing approaches[5][6] in terms of fault localization accuracy.

RQ 2. Consideration of the fault types that the proposed method can detect.

We evaluate the effectiveness of the proposed method by comparing it with Ochiai[5] and Tarantula[6]. Since Ochiai and Tarantula are representative SFL methods and are used as benchmarks in several techniques[2], we chose these methods for the comparison. Our method is based on a DNN model trained on the set of statements executed in each test case, thus it is expected to be effective for faults that are difficult to identify using execution frequency information only. We discuss our experiment results in terms of fault types.

The TopN % is used as a measure of the accuracy of fault localization. The TopN % represents that a bug is classified into the top N % of the total, and a smaller N value indicates a better fault localization performance. In the case of multiple bugs, the largest TopN % is used as such a metric.

3.2. Subject Programs

We conducted evaluation experiments using bugs and fixes data provided by Defects4j[8], a database of actual bugs identified in Java projects. Lang, Math, and Chart from Defects4j are selected as the projects for the experiments. We choose bug cases that meet the following conditions.

- Bug fixes only with code addition are excluded. If the bug is fixed by code addition only, the original buggy source code does not have any defects to be pointed out.

Table 1
Details of Target Programs

Project	Number of Versions	LOC	Number of Tests
Lang	30	58389	54987
Math	29	23623	83364
Chart	15	10094	27036

- The fault statement is executed at least once in each of the fail and pass tests: the SFL approaches require bugs to be executed in those tests.

Execution coverages are collected using OpenClover[9]. Because execution is not recorded for class member variable definitions, etc., due to OpenClover's specifications, we excluded from the fault set the parts of the program that are not recorded as code coverage information. The number of lines (LOC) and number of tests are shown in Table 1.

3.3. Setup for The DNN Model

In performing supervised training, the weight parameters of LSTM and Multi-Layer Perceptrons (MLP) are initialized with random values. The size of the vector representation of words encoded with Word2Vec is set to 128. Further increasing the size of the distributed representation did not significantly affect classification performance. The size of the output vector of the LSTM network is set to 256. The MLP is three hidden layers of $256, 128$ and 64. The parameters for each layer are chosen according to the results from the existing methods[7] in terms of computation time and classification performance. Adam optimizer is used, and the learning rate is adjusted according to the size of the LOC of the SUT under the experiment. The learning rate was selected between $1e-06$ and $5e-05$. A learning rate of less than $1e-06$ increases the computation time required for convergence, while a value greater than $5e-05$ adversely affects the learning results. PyTorch (ver. 1.10.1) is used as the machine learning framework. In our evaluation experiments, we trained ten DNN models with each SUT, and take their averages as the final experiment result.

4. RESULTS AND DISCUSSIONS

Results. Figure 5 shows the SFL performance in terms of TopN % for each approach. The horizontal axis represents the TopN % and indicates the amount of source code examined by the developer. The vertical axis shows the percentage of identified faults, and 100% means that all faults are identified. For example, the vertical axis plot with a Top 50% is more than 90%, indicating that more than 90% of the faults are identified by investigating half (50%) of the source code. Figure 5 contains three plots, with gray and blue indicating Ochiai and Tarantula, respectively, and the orange plot showing the performance of the proposed method.

Table 2 shows the results of adapting the Wilcoxon Signed-Rank test to the experiment results. The null hypothesis indicates that there is no significant difference between the two groups,

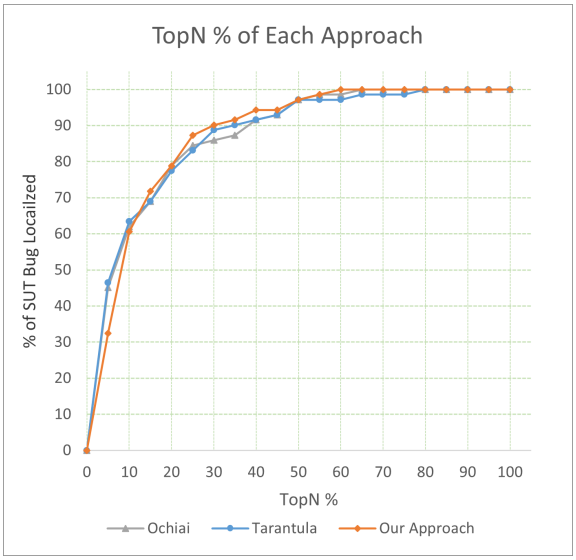

Figure 5: SFL Performance in Terms of TopN %

Table 2
Test Results in Evaluation Experiments

All Programs	1-tailed (left)
Our Approach vs Ochiai	6.258E-01
Our Approach vs Tarantula	6.149E-01
Multi Fault Programs	1-tailed (left)
Our Approach vs Ochiai	1.494E-02
Our Approach vs Tarantula	3.937E-02

while the alternative hypothesis indicates that there is a significant difference between the two groups. The alternative hypothesis is described below.

1-tailed (left): The proposed method has a smaller TopN % value than the existing approach.

Since a smaller N value indicates higher accuracy, if the left-tailed Wilcoxon Signed-Rank test is accepted, the proposed method is significantly better than the existing approach in fault localization performance. In this paper, a p-value less than 0.05 is considered statistically significant.

Discussions of RQ1. In Figure 5, which shows the results for all programs in the experiment SUT, the proposed method is able to identify the same or slightly more faults than the existing approach in all plots except the Top 5% and 10%. However, the tests shown in Table 2 did not confirm that the proposed method is more accurate than the existing approach ($p = 0.626, p = 0.615$). Analyzing the experiment results, we found that the proposed method is more accurate than the existing approach in only about half of the programs. To examine the types of faults for which the proposed method is effective, we conducted a statistical test on programs only with multiple faults. Those programs accounted for about 20% of the total number of programs. Table 2 shows the test results for multiple faults programs only, and indicates that the proposed method

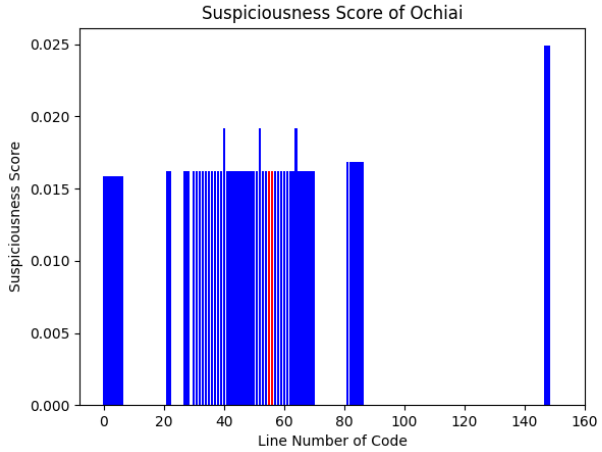

Figure 6: Suspiciousness Score of Ochiai in Each Statement

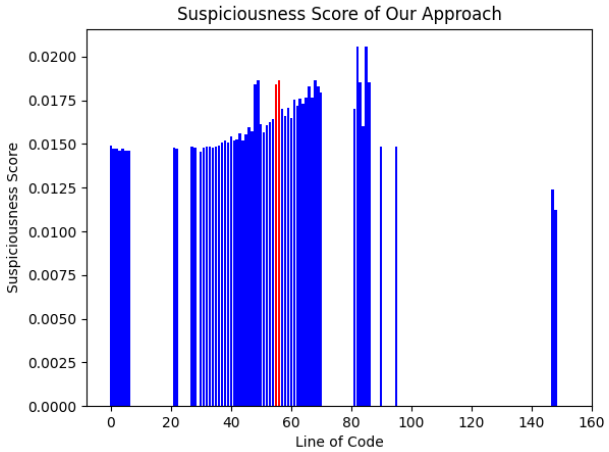

Figure 7: Suspiciousness Score of Our Method in Each Statement

is significantly more accurate than the two existing approaches ($p = 0.0149, p = 0.0394$). From this result, we consider that the proposed method can achieve higher accuracy than the existing approaches for programs with multiple faults. Indeed, SFL are not good at dealing with those multiple faults programs, because each fault statement is not necessarily executed in all Fail test cases, and thus it is difficult to identify faults based on execution frequency information only.

Discussions of RQ2. We examined in detail fault types for single-fault programs. In Defects4j[8], each program fault is assigned a Repair Actions tag, which indicates the specific work to be done to fix bugs. We used the Repair Actions tag to investigate the type of faults in our experiment results for programs with a single fault. Programs with no significant difference in accuracy between the existing and the proposed approaches are excluded from here, and programs with

an accuracy difference of 15 points or more from Ochiai in the TopN % are included in the discussion. Since there is little difference in accuracy between Tarantula and Ochiai, there is no impact on the discussion even when programs with large accuracy differences from Tarantula are included in the discussion.

As a result of investigating the types of faults, tags such as changing the type of variables and modifying assignment expression are identified in the programs that the proposed method is more accurate than the existing approach. Figure 6 shows Ochiai's suspiciousness scores for each statement in Chart version 7, the vertical axis means the score of each statement, and the red plot means a faulty statement. Ochiai is unable to give the fault statement a high suspicion score, giving it the same score as many of the other statements. Figure 7 shows the suspiciousness scores in our method. The fault in Chart version 7 is one of the faults for which our method worked well, and our method gives the fault statement the third highest suspiciousness score. The Chart version 7 has faulty assignment statements, and conventional SFL does not give a high suspiciousness score to the statement executed in all the test cases regardless of pass or fail, thus it is difficult to identify the fault. Our approach is to use DNN models learned from the combination of executed statements in each test case to identify the faults. Thus, we can deal with the case in which the fault is a statement that SFL are hard to deal with the existing approach.

Besides, we investigated the error types that the program outputs. The percentage of error types that terminated abnormally, such as Exception, is larger in programs in which the proposed method is less accurate than the existing approach, and the programs that the proposed method has higher accuracy than the existing approaches, all of the error types are AssertionFailedError. Future work is needed to clarify for which errors our method is effective.

5. Related Work

Several approaches[12][13] have been proposed to compute suspiciousness scores similar to Tarantula and Ochiai. These approaches use only the number of executions of each statement collected from code coverages for SFL. The proposed method uses a combination of executed statements in each test case, thus is different from the information used in the SFL methods.

Existing function-level fault localization techniques[14][15] use function coverage or statement coverage to compute the suspiciousness values. Murtaza et al.[14] uses decision trees to identify patterns of function calls related to failures. Sohn et al[15]. proposed the approach to rank faulty methods higher using genetic programming (GP) and linear rank-supported vector machines (SVM). These approaches are function-level SFL approaches, which are different in granularity from the statement granularity SFL approaches discussed in this paper.

As one of the recent techniques for dynamic analysis using machine learning, Li et al. proposed DeepRL4FL, which identifies buggy codes by treating fault localization as an image pattern recognition problem[16]. Li et al.'s approach requires marking the statements that are faulty as training data. Therefore, the training data used in Li et al.'s approach is more informative than our approach's training data.

Wang et al. proposed a method to generate a passed execution from a failed execution[17]. In Wang et al.'s approach, a passed execution is generated by toggling the results of conditional

branch instances in a failed execution. It differs from our idea that an incomplete execution with a missing spectrum affects test results.

The other recent related researches to this paper are approaches[18][19] that use virtual coverage to locate faults. These approaches are similar to our method in the idea of using the output of a DNN model that classifies test cases to identify faults, but differ from the proposed method to use missing spectrum in that they use virtual coverage, which virtually represents that only one specific line is executed.

6. Conclusion

This paper proposes a new approach to fault localization using supervised test case learning. The proposed method is to identify faulty statements using output values of DNN model resulting from ablated execution traces. This approach is evaluated using three different SUTs. Evaluation experiments show that the proposed method achieves higher accuracy than existing approaches when there are multiple faults and when the faulty statement is an assignment to a variable. In the future, we plan to compare the proposed method with other DNN-based SFL methods and discuss the pros and cons of the proposed method.

Acknowledgments

The research is also being partially conducted as Grant-in-Aid for Scientific Research C (21K11826).

References

[1] G. Tassey, The economic impacts of inadequate infrastructure for software testing, 2002.

[2] H. A. de Souza, M. L. Chaim, F. Kon, Spectrum-based software fault localization: A survey of techniques, advances, and challenges (2016). arXiv:1607.04347.

[3] A. Perez, R. Abreu, A qualitative reasoning approach to spectrum-based fault localization, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 372–373. doi:10.1145/3183440.3195015.

[4] Q. I. Sarhan, A. Beszédes, A survey of challenges in spectrum-based software fault localization, IEEE Access 10 (2022) 10618–10639. doi:10.1109/ACCESS.2022.3144079.

[5] R. Abreu, P. Zoeteweij, A. J. Van Gemund, An evaluation of similarity coefficients for software fault localization, in: 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06), 2006, pp. 39–46. doi:10.1109/PRDC.2006.18.

[6] J. Jones, M. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, 2002, pp. 467–477. doi:10.1145/581396.581397.

[7] F. Tsimpourlas, A. Rajan, M. Allamanis, Supervised learning over test executions as a test oracle, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing,

SAC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 1521–1531. doi:10.1145/3412841.3442027.

[8] R. Just, D. Jalali, M. D. Ernst, Defects4j: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, Association for Computing Machinery, New York, NY, USA, 2014, pp. 437–440. doi:10.1145/2610384.2628055.

[9] OpenClover, https://openclover.org/, (Accessed 18 October 2023).

[10] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, CoRR abs/1310.4546 (2013). arXiv:1310.4546.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, 2023. arXiv:1706.03762.

[12] R. Abreu, P. Zoeteweij, A. J. van Gemund, On the accuracy of spectrum-based fault localization, in: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), 2007, pp. 89–98. doi:10.1109/TAIC.PART.2007.13.

[13] W. E. Wong, V. Debroy, R. Gao, Y. Li, The dstar method for effective software fault localization, IEEE Transactions on Reliability 63 (2014) 290–308. doi:10.1109/TR.2013.2285319.

[14] S. Murtaza, N. Madhavji, M. Gittens, A. Hamou-Lhadj, Identifying recurring faulty functions in field traces of a large industrial software system, Reliability, IEEE Transactions on 64 (2015) 269–283. doi:10.1109/TR.2014.2366274.

[15] J. Sohn, 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, ISSTA 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 273–283. URL: https://doi.org/10.1145/3092703.3092717. doi:10.1145/3092703.3092717.

[16] Y. Li, S. Wang, T. N. Nguyen, Fault localization with code coverage representation learning, in: Proceedings of the 43rd International Conference on Software Engineering, ICSE '21, IEEE Press, 2021, pp. 661–673. doi:10.1109/ICSE43902.2021.00067.

[17] T. Wang, A. Roychoudhury, Automated path generation for software fault localization, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 347–351. URL: https://doi.org/10.1145/1101908.1101966. doi:10.1145/1101908.1101966.

[18] W. E. Wong, V. Debroy, R. Golden, X. Xu, B. Thuraisingham, Effective software fault localization using an rbf neural network, IEEE Transactions on Reliability 61 (2012) 149–169. doi:10.1109/TR.2011.2172031.

[19] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, X. Zhang, A study of effectiveness of deep learning in locating real faults, Information and Software Technology 131 (2021) 106486. doi:https://doi.org/10.1016/j.infsof.2020.106486.