How to Evaluate Statistical Fault Localization

Tim A. D. Henderson
Google Inc.
Mountain View, California
tadh@google.com

1 INTRODUCTION

Automatic fault localization is a software engineering technique to assist a programmer during the debugging process by suggesting suspicious locations that may be related to the root cause of the bug. The big idea behind behind automatic fault localization (or just fault localization) is by pointing the programmer towards the right area of the program the programmer will find the cause of the bug more quickly.

One approach to fault localization is *Spectrum Based Fault Localization* which is also known as *Coverage Based Statistical Fault Localization* (CBSFL) [22, 29, 41]. This approach uses test coverage information to rank the statements from most "suspicious" to least suspicious. To perform CBSFL, test cases are run through an instrumented program. The instrumentation collects coverage profiles which report each statement[1] executed during the test run. A *test oracle* is used to label each execution profile with whether or not the test passed or failed. Such oracles can be either automatic or manual (i.e. a human). The labeled execution profiles are referred to as the *coverage spectra*.

CBSFL techniques score each program element (location in the program) by its "statistical suspiciousness" such that the most suspicious element has the highest score. The scores are computed by *suspiciousness metrics* which attempt to quantify the relationship between the execution each program element and the occurrence of program failure.

There have been a great many statistical fault localization suspiciousness metrics proposed [29] since the idea was first proposed by Jones, Harrold and Stasko in 2002 [22]. The majority of the metrics are computed from just a few values: the number of tests n, the number of passing tests p, the number failing tests f, the number of test runs an element e was executed in n_e, the number of passing test runs an element was executed in p_e, and the number of failing test runs an element was executed in f_e. For instance, using these simple statistics one can estimate the conditional probability of program failure F given that a particular element e was executed:

$$\Pr[F|e] = \frac{\Pr[F \cap e]}{\Pr[e]} \approx \frac{\frac{f_e}{n}}{\frac{n_e}{n}} = \frac{f_e}{n_e} \tag{1}$$

While many of the studies in statistical fault localization use more complex formulas [29, 41] (with various technical motivations) most of the metrics are measures statistical association as in the equation above.

The runtime information used to compute the above statistics is commonly referred to as the *coverage spectra* of the program. Coverage spectra is a matrix $\mathbf{C}$ where $\mathbf{C}_{i,j} > 0$ indicates that the program element $\mathcal{E}_i$ was executed at least once during test $\mathcal{T}_j$. Additionally, there is an additional "test status" vector $\mathbf{S}$ where $\mathbf{S}_j$ = "F" if test $\mathcal{T}_j$ failed and $\mathbf{S}_j$ = "P" if test j passed.

To collect the coverage spectra the program is instrumented to collect a runtime profile of the executed elements. This instrumentation can be done (in principle) at any granularity including: expression, statement, basic block[2], function, class, file, or package. The statistical methods which make use of spectra are agnostic to their granularity. While the granularity does not effect the statistical computation it changes how the localization results are perceived by the programmer. In the past, programmers have indicated a desire for finer grained results: at statement, basic block, or function level [24] over very coarse grained results at the class, file, or package level.

Some statistical fault localization techniques use additional information to either improve accuracy or provide more explainable results. For instance, work on *Causal Fault Localization* uses additional static and dynamic information to control for statistical confounding [11]. In contrast *Suspicious Behavior Based Fault Localization* (SBBFL) uses runtime control flow information (the behavior) to identify groups of collaborating suspicious elements [20]. These techniques leverage data mining techniques [5] such as frequent [4, 7, 47] or significant pattern mining [20, 46]. When significant patterns are mined metrics (such as statistical fault localization suspiciousness metrics) are used to identify the most significant patterns [12, 20].

Finally, a variety of non-statistical (or mixed methods) techniques for fault localization have been explored [1–3, 44]. These range from delta debugging [49] to nearest neighbor queries [38] to program slicing [30, 42] to information retrieval [26, 31, 51] to test case generation [10, 37, 39]. Despite differences in the technical and theoretical approach of these alternate methods they also suggest locations (or groups of locations) for the programmer to consider when debugging.

2 EVALUATION METHODS

Some of the earliest papers in fault localization do not provide a quantitative method for evaluating performance (as is seen in later papers [36]). For instance, in the earliest CBSFL paper [22] (by Jones *et al.*'s) the technique is evaluated using a qualitative visualization. At the time, this was entirely appropriate as Jones was proposing a technique for visualizing test coverage for assisting the debugging process. The test coverage visualization was driven by what is now called a statistical fault localization metric (Tarantula). The evaluation visualization aggregated the visualizations all of the programs included in the study.

While, the evaluation method used in the Jones paper effectively communicated the potential of CBSFL (and got many researchers

[1] The coverage can be collected at other levels as well. For instance there has been work which collects it at the class, method, and basic block levels.

[2] A basic block is a sequence of sequential instructions, always entered from the first instruction and exited from the last [8].

excited about the idea) it was not good way to compare multiple fault localization techniques. In 2005 Jones and Harrold [23] conducted a study which compared their Tarantula technique to 3 other techniques: Set Union and Intersection [6], Nearest Neighbor [38], and Cause-Transitions [14]. These techniques all took unique approaches toward the fault localization problem and were originally evaluated in different ways. Jones and Harrold re-evaluated all 5 methods under a new common evaluation framework.

In the 2005 paper, Jones and Harrold evaluate the effectiveness of each technique by using the technique to rank the statements in the subject programs. Each technique ranked the statements from most likely to be the cause of the fault to least likely. For Tarantula, the statements are ranked using the Tarantula suspiciousness score:[3]

Definition 1 (Tarantula Rank Score [23])**.** Given a set of locations L with their suspiciousness scores $s(l)$ for $l \in L$ the Rank Score for a location $l \in L$ is:

$$|\{x \ : \ x \in L \wedge s(x) > s(l)\}| + |\{x \ : \ x \in L \wedge s(x) = s(l)\}|$$

For Set Union and Intersection, Nearest Neighbor and Cause-Transitions the statements are ranked using a System Dependence Graph (SDG) [21] technique from Renieres and Reiss [38] who first suggested the ranking idea. The ranks are then used to calculate the Tarantula Rank Score.

In the Jones and Harrold evaluation the authors do not use the Tarantula Rank Score directly but instead use a version normalized by program size:

Definition 2 (Tarantula Effectiveness Score (Expense) [23])**.** The percentage of program elements that do not need to be examined to find the fault when the elements are arranged according to their rank. Formally: let n be the total number of program elements, and let $r(f)$ be the Tarantula Rank Score of the faulty element f then the score is:

$$\frac{n - r(f)}{n}$$

Using the normalized effectiveness score Jones and Harrold directly compare the fault localization effectiveness of each of the considered methods. They did this in two ways. First, they presented a table (Table 2) which bucketed all the buggy versions form all the programs by the percentage given by the Tarantula Effectiveness Score. Second, they presented a figure (Figure 2) which showed the data in Table 2 as a cumulative curve.

The basic evaluation method presented by Jones and Harrold has become the standard evaluation method. Faulty statements are scored, ranked, rank-scored, normalized, and then aggregated over all versions and programs to provide an overall representation of the fault localization method's performance (a few examples: [25, 29, 40, 41, 43, 50]). While the basic method has stayed fairly consistent, there has been some innovation in the scoring (both the Rank Score and the Effectiveness Scores).

For instance, Wong *et al.* [43] introduced the most commonly used Effectiveness Score the $\mathcal{EXAM}$ score. This score is essentially the same as the Expense score except it gives the percentage of elements which need to be examined rather than those avoided.

Definition 3 ($\mathcal{EXAM}$ Score [43])**.** The percentage of program elements that need to be examined to find the fault when the elements are arranged according to their rank. Formally: let n be the total number of program elements, and let $r(f)$ be the Tarantula Rank Score of the faulty element f then the score is:

$$\frac{r(f)}{n}$$

Ali *et al.* [9] identified an important problem with the Jones and Harrold evaluation: some fault localization metrics and algorithms rank statements equally. This is captured in the second term in the definition for the Tarantula Rank Score. However, Ali points out that this introduces bias towards algorithms that always assign unique scores (that are close together) rather than those that would score the same group of statement equally. The fix is to instead compute the expected number of statements the programmer would examine if they chose the next equally scored element at random.

Definition 4 (Rank Score)**.** Gives the expected number of locations a programmer would inspect before finding the bug. Formally, given a set of locations L with their suspiciousness scores $s(l)$ for $l \in L$ the Rank Score for a location $l \in L$ is [9]:

$$|\{x \ : \ x \in L \wedge s(x) > s(l)\}| + \frac{|\{x \ : \ x \in L \wedge s(x) = s(l)\}|}{2}$$

Following Ali, we recommend utilizing the above definition for Rank Score over the Tarantula definition.

Parin and Orso [35] conducted a user study which looked at the programmer experience of using a statistical fault localization tool (Tarantula [22]). Among their findings they found that programmers would not look deeply through the list of locations and would instead only consider the first few items. As a result they encouraged studies to no longer report scores as percentages. While some studies still report the percentages most studies are now reporting the absolute (non-percentage) rank scores. Reporting as absolute scores is important for another reason, if percentage ranks are reported larger programs can have much larger absolute ranks for the same percentage rank. This biases the evaluation toward large programs even when the actual localization result is poor.

Steimann *et al.* [40] identified a number of threats to validity in CBSFL studies including: heterogeneous subject programs, poor test suites, small sample sizes, unclear sample spaces, flaky tests, total number of faults, and masked faults. For evaluation they used the Rank Score modified to deal with k faults tied at the same rank.

Definition 5 (Steimann Rank Score)**.** Gives the expected number of locations a programmer would inspect before finding the bug when multiple faulty statements have the same rank. Formally, given a set of locations L with their suspiciousness scores $s(l)$ for $l \in L$ the Rank Score for a location $l \in L$ is [40]:

$$|\{x \ : \ x \in L \wedge s(x) > s(l)\}| + \frac{|\{x \ : \ x \in L \wedge s(x) = s(l)\}| + 1}{|\{x \ : \ x \in L \wedge s(x) = s(l) \wedge x \text{ is a faulty location}\}| + 1}$$

Moon *et al.* [32] proposed Locality Information Loss (LIL) as an alternative evaluation framework. LIL models the localization result

[3] It is ahistorical to call Tarantula metric a suspiciousness score when referring to the 2002 paper [22]. Jones introduced the term suspiciousness score in the 2005 paper [23] for the purpose of ranking the statements. However, the term is now in common use and it was explained above.

as a probability distribution constructed from the suspiciousness scores:

Definition 6 (LIL Probability Distribution)**.** Let τ be a suspicious metric normalized to the $[0, 1]$ range of reals. Let n be the number statements in the program. Let S be the set of statements. For all $1 \leq i \leq n$ let $s_i \in S$. The constructed probability distribution is:

$$P_\tau(s_i) = \frac{\tau(s_i)}{\sum_{j=1}^{n} \tau(s_j)}$$

LIL uses a measure of distribution divergence (Kullback-Leibler) to compute a score of how different the constructed distribution is from the "perfect" expected distribution. The advantage of the LIL framework is it does not depend on a list of ranked statements and can be applied to non-statistical methods (using a synthetic τ). The disadvantage of LIL is it does not indicate programmer effort (as indicated by the Rank Score). However, it may be a better metric to use when evaluating fault localization systems as a component for automated bug repair systems.

Pearson *et al.* [36] re-evaluated a number of previous results using new real world subject programs with real defects and test suites. In contrast to previous work they made use of statistical hypothesis testing and confidence intervals to test the significance of the results. To evaluate the performance of each technique under study they used the $\mathcal{EXAM}$ score reporting best, average, and worst case results for multi-statement faults.

T-Score [27] is designed for non-statistical fault localization methods which produce a small set of suspicious statements in the program. To evaluate how helpful these reports are *T-Score* uses the Program Dependence Graph (PDG) [19, 21] to compute a set of vertices in the graph that must be examined in order to reach any faulty vertex. This set is computed via a breadth first search from the set of vertices in the report. Finally, the score is computed as the percentage of examined vertices out of the total number of vertices in the graph.

3 MULTIPLE FAULT EVALUATIONS

With the exception of LIL, the evaluation methods discussed so far are generally defined to operate with a single faulty location. However, there may be multiple faults or multiple locations associated with a single fault or both. Multiple faults can interact [16] and interfere with the performance of the fault localizer. For evaluation purposes one of the most popular methods is to take either best result [45], the average result [2, 34], or the worst result [45].

4 EVALUATING OTHER TECHNIQUES

One of the challenges with the methods presented so far is they may not work well for evaluating alternate fault localization methods. For instance, information retrieval based localization methods do not necessarily score and rank every program location. Instead they produce a report of associated regions. Jones and Harrold [23] used a synthetic ranking system [38] based on the SDG [21] which in principle could be used in such situation. However, like the *T-Score* it uses an arbitrary method (minimal dependence spheres) to compute the number of SDG nodes which must be examined.

The LIL method could also potentially be used to evaluate alternative methods. It does not rely on ranking but instead on the suspiciousness scores which it converts into a probability distribution. To support evaluating report based localization the reports are converted to a probability distribution with all locations in the report set to a equal high probability and all locations not in the report set to a tiny probability.

Suspicious Behavior Based Fault Localization [20] requires particular care. These methods are produce a ranked set of "behaviors" which are structured groups of interacting program locations. The structure could be a call invocation structure [13, 15, 17, 28, 48], a general control flow structure [12, 20, 33], or even an information flow structure [18]. The structures are scored and ranked similar to CBSFL. However, unlike in CBSFL all program locations are not necessarily included. In the past, studies have used a variety of techniques to evaluated the effectiveness including precision and recall [12] and scores based off of the $\mathcal{EXAM}$ score [20].

Another subtle special case involves comparing statistical techniques which operate on different granularity levels. As mentioned previously, coverage can be collected at any granularity level: expression, statement, basic block, method or function, class, file, and even non-structural elements such as paths. Any of the CBSFL metrics can be used with any of these granularities. However, a single study using any of the previous evaluation methods must keep the granularity consistent. This makes it impossible to compare across granularity levels. This is makes it particularly difficult to accurately compare method level behavioral approaches [28] to CBSFL.

5 ASSUMPTIONS

The biggest assumption that all evaluation models make is so-called *perfect bug understanding* which assumes programmers will recognize a bug as soon as they "examine" the faulty location. This assumption is obviously false [35]. However, it continues to be a useful simplifying assumption for evaluation purposes of the localization algorithms. From the standpoint of automated fault localization there are really two tasks: 1) finding the fault and 2) explaining the fault. Assuming perfect bug understanding is reasonable for evaluating a tools performance on task 1. However, their is the important caveat that programmers need more assistance at task 2. As a research community we do not currently have a standard method for evaluating our algorithmic performance on task 2.

The second assumption is that programmers will follow the rank list or suspiciousness scores when debugging a program using a fault localization tool. This assumption is obviously false as well [35]. A programmer may follow the list for the very first item and even the second but where they go from there is likely only partially influenced by the list. The bigger influence will be from the conclusions they are drawing from what they learn upon inspecting each location. Our new model does not require this assumption.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J C van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035

[2] Rui Abreu, Peter Zoeteweij, and Arjan Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46. https://doi.org/10.1109/PRDC.2006.18

[3] Pragya Agarwal and Arun Prakash Agrawal. 2014. Fault-localization Techniques for Software Systems: A Literature Review. *SIGSOFT Softw. Eng. Notes* 39, 5 (sep 2014), 1–8. https://doi.org/10.1145/2659118.2659125

[4] Charu C. Aggarwal, Mansurul A. Bhuiyan, and Mohammad Al Hasan. 2014. Frequent Pattern Mining Algorithms: A Survey. In *Frequent Pattern Mining*. Springer International Publishing, Cham, 19–64. https://doi.org/10.1007/978-3-319-07821-2_2

[5] Charu C. Aggarwal and Jiawei Han (Eds.). 2014. *Frequent Pattern Mining*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-07821-2

[6] Hiralal Agrawal, J.R. Horgan, Saul London, and W.E. Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95.* IEEE Computer Society, 143–151. https://doi.org/10.1109/ISSRE.1995.497652

[7] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. *ACM SIGMOD Record* 22, 2 (jun 1993), 207–216. https://doi.org/10.1145/170036.170072

[8] Alfred Aho, Ravi Sethi, Monica S. Lam, and Jeffery D. Ullman. 2007. *Compilers: principles, techniques, and tools.*

[9] Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, and Wantao Wang. 2009. Evaluating the Accuracy of Fault Localization Techniques. *2009 IEEE/ACM International Conference on Automated Software Engineering* (2009), 76–87. https://doi.org/10.1109/ASE.2009.89

[10] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed Test Generation for Effective Fault Localization. In *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA '10)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/1831708.1831715

[11] G.K. George K Baah, Andy Podgurski, and Mary Jean M.J. Harrold. 2010. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA '10)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/1831708.1831717

[12] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. 2009. Identifying Bug Signatures Using Discriminative Graph Mining. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/1572272.1572290

[13] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. 2009. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 34–44. https://doi.org/10.1109/ICSE.2009.5070506

[14] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. *Proceedings of the 27th international conference on Software engineering - ICSE '05* (2005), 342. https://doi.org/10.1145/1062455.1062522

[15] Themistoklis Diamantopoulos and Andreas Symeonidis. 2014. Localizing Software Bugs using the Edit Distance of Call Traces. *International Journal on Advances in Software* 7, 1 & 2 (2014), 277–288.

[16] Nicholas DiGiuseppe and James A. Jones. 2011. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*. ACM, 210. https://doi.org/10.1145/2001420.2001446

[17] Frank Eichinger, Klemens Böhm, and Matthias Huber. 2008. Mining Edge-Weighted Call Graphs to Localise Software Bugs. In *European Conference Machine Learning and Knowledge Discovery in Databases*, Walter Daelemans, Bart Goethals, and Katharina Morik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 333–348. https://doi.org/10.1007/978-3-540-87479-9_40

[18] Frank Eichinger, Klaus Krogmann, Roland Klug, and Klemens Böhm. 2010. Software-defect Localisation by Mining Dataflow-enabled Call Graphs. In *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part I (ECML PKDD'10)*. Springer-Verlag, Berlin, Heidelberg, 425–441.

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. , 319–349 pages. https://doi.org/10.1145/24039.24041

[20] Tim A D Henderson and Andy Podgurski. 2018. Behavioral Fault Localization by Sampling Suspicious Dynamic Control Flow Subgraphs. In *IEEE Conference on Software Testing, Validation and Verification*. IEEE, Västerås, Sweden.

[21] Susan Horwitz. 1990. Identifying the Semantic and Textual Differences Between Two Versions of a Program. *SIGPLAN Not.* 25, 6 (jun 1990), 234–245. https://doi.org/10.1145/93548.93574

[22] J.a. Jones, M.J. Harrold, and J. Stasko. 2002. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002* (2002). https://doi.org/10.1145/581339.581397

[23] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949

[24] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, New York, New York, USA, 165–176. https://doi.org/10.1145/2931037.2931051

[25] David Landsberg, Hana Chockler, Daniel Kroening, and Matt Lewis. 2015. Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme. In *International Conference on Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Alexander Egyed and Ina Schaefer (Eds.), Vol. 9033. Springer Berlin Heidelberg, Berlin, Heidelberg, 115–129. https://doi.org/10.1007/978-3-662-46675-9

[26] Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, New York, New York, USA, 579–590. https://doi.org/10.1145/2786805.2786880

[27] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*. ACM, New York, NY, USA, 872–881. https://doi.org/10.1145/1150402.1150522

[28] Chao Liu, Hwanjo Yu, Philip S Yu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S Yu. 2005. Mining Behavior Graphs for âĂIJBacktraceâĂİ of Noncrashing Bugs. In *Proceedings of the 2005 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 286–297. https://doi.org/10.1137/1.9781611972757.26

[29] Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. 2014. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process* 26, 2 (feb 2014), 172–219. https://doi.org/10.1002/smr.1616

[30] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89, 1 (2014), 51–62. https://doi.org/10.1016/j.jss.2013.08.031

[31] Andrian Marcus, Andrey Sergeyev, Václav Rajlich, and Jonathan I. Maletic. 2004. An information retrieval approach to concept location in source code. *Proceedings - Working Conference on Reverse Engineering, WCRE* (2004), 214–223. https://doi.org/10.1109/WCRE.2004.10

[32] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating faulty programs for fault localization. *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014* (2014), 153–162. https://doi.org/10.1109/ICST.2014.28

[33] Zaynab Mousavian, Mojtaba Vahidi-Asl, and Saeed Parsa. 2011. Scalable Graph Analyzing Approach for Software Fault-localization. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 15–21. https://doi.org/10.1145/1982595.1982599

[34] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 20, 3 (2011), 1–32. https://doi.org/10.1145/2000791.2000795

[35] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *ISSTA*. ISSTA, 199–209.

[36] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 609–620. https://doi.org/10.1109/ICSE.2017.62

[37] Alexandre Perez, Rui Abreu, and André Riboira. 2014. A Dynamic Code Coverage Approach to Maximize Fault Localization Efficiency. *J. Syst. Softw.* 90 (apr 2014), 18–28. https://doi.org/10.1016/j.jss.2013.12.036

[38] Manos Renieres and S.P. Reiss. 2003. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE Comput. Soc, 30–39. https://doi.org/10.1109/ASE.2003.1240292

[39] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using likely invariants for automated software fault localization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, Vol. 41. ACM Press, New York, New York, USA, 139. https://doi.org/10.1145/2451116.2451131

[40] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013* (2013), 314. https://doi.org/10.1145/2483760.2483767

[41] Shih-Feng Sun and Andy Podgurski. 2016. Properties of Effective Metrics for Coverage-Based Statistical Fault Localization. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 124–134. https://doi.org/10.1109/ICST.2016.31

[42] Frank Tip. 1995. A survey of program slicing techniques. *Journal of programming languages* 3, 3 (1995), 121–189.

[43] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. 2008. A Crosstab-based Statistical Method for Effective Fault Localization. In *2008 International Conference on Software Testing, Verification, and Validation*. IEEE, 42–51. https://doi.org/10.1109/ICST.2008.65

[44] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (aug 2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[45] W. Eric Wong, Yu Qi, Lei Zhao, and Kai Yuan Cai. 2007. Effective fault localization using code coverage. *Proceedings - International Computer Software and Applications Conference* 1, Compsac (2007), 449–456. https://doi.org/10.1109/COMPSAC.2007.109

[46] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S Yu. 2008. Mining Significant Graph Patterns by Leap Search. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 433–444. https://doi.org/10.1145/1376616.1376662

[47] Xifeng Yan and Jiawei Han. 2002. gSpan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE Comput. Soc, 721–724. https://doi.org/10.1109/ICDM.2002.1184038

[48] Anis Yousefi and Alan Wassyng. 2013. A Call Graph Mining and Matching Based Defect Localization Technique. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops.* IEEE, 86–95. https://doi.org/10.1109/ICSTW.2013.17

[49] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why? *SIGSOFT Softw. Eng. Notes* 24, 6 (oct 1999), 253–267. https://doi.org/10.1145/318774.318946

[50] Yan Zheng, Zan Wang, Xiangyu Fan, Xiang Chen, and Zijiang Yang. 2018. Localizing multiple software faults based on evolution algorithm. *Journal of Systems and Software* 139 (2018), 107–123. https://doi.org/10.1016/j.jss.2018.02.001

[51] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. *Proceedings - International Conference on Software Engineering* (2012), 14–24. https://doi.org/10.1109/ICSE.2012.6227210