Boosting Spectrum-Based Fault Localization using PageRank

Mengshi Zhang
Department of Electrical and Computer Engineering
University of Texas at Austin, USA
mengshi.zhang@utexas.edu

Xia Li
Department of Computer Science
University of Texas at Dallas, USA
xxl124730@utdallas.edu

Lingming Zhang
Department of Computer Science
University of Texas at Dallas, USA
lingming.zhang@utdallas.edu

Sarfraz Khurshid
Department of Electrical and Computer Engineering
University of Texas at Austin, USA
khurshid@ece.utexas.edu

ABSTRACT

Manual debugging is notoriously tedious and time-consuming. Therefore, various automated fault localization techniques have been proposed to help with manual debugging. Among the existing fault localization techniques, spectrum-based fault localization (SBFL) is one of the most widely studied techniques due to being lightweight. A focus of existing SBFL techniques is to consider how to differentiate program source code entities (i.e., one dimension in program spectra); indeed, this focus is aligned with the ultimate goal of finding the faulty lines of code. Our key insight is to enhance existing SBFL techniques by additionally considering how to differentiate tests (i.e., the other dimension in program spectra), which, to the best of our knowledge, has not been studied in prior work.

We present PRFL, a lightweight technique that boosts spectrum-based fault localization by differentiating tests using PageRank algorithm. Given the original program spectrum information, PRFL uses PageRank to *recompute* the spectrum information by considering the contributions of different tests. Then, traditional SBFL techniques can be applied on the recomputed spectrum information to achieve more effective fault localization. Although simple and lightweight, PRFL has been demonstrated to outperform state-of-the-art SBFL techniques significantly (e.g., ranking 42% more real faults within Top-1 compared with the most effective traditional SBFL technique) with low overhead (e.g., around 2 minute average extra overhead on real faults) on 357 real faults from 5 Defects4J projects and 30692 artificial (i.e., mutation) faults from 87 GitHub projects, demonstrating a promising future for considering the contributions of different tests during fault localization.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Software testing, Spectrum-based fault localization, PageRank

ACM Reference format:
Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting Spectrum-Based Fault Localization using PageRank. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 12 pages.
https://doi.org/10.1145/3092703.3092731

1 INTRODUCTION

Software debugging is an expensive and painful process that costs developers a lot of time and effort. For example, it has been reported that debugging can take up to 80% of the total software cost [55]. Thus, there is a pressing need for automated techniques that support debugging. In the last two decades, various fault localization approaches have been proposed to help developers locate the root cause of failures, e.g., spectrum-based [6, 15, 24, 40, 59, 63], slicing-based [36, 62], machine-learning-based [20, 61], and mutation-based [39, 45, 65] techniques. Recent survey by Wong et al. [58] shows more details about various fault localization approaches.

Among the existing fault localization approaches, spectrum-based fault localization (SBFL), is one of the most widely studied fault localization techniques in the literature [35, 40, 59, 63]. Despite that SBFL is a particularly lightweight approach, it has been shown to be competitive compared to other approaches [49]. SBFL techniques take as input a set of passing and failing tests, and analyze program execution traces (spectra) of successful and failed executions. The execution traces record the program entities (such as statements, basic blocks, and methods) executed by each test. Intuitively, a program entity covered by more failing tests but less passing tests is more likely to be faulty. Hence, SBFL applies a ranking formula to compute suspiciousness scores for each entity based on the program spectra. Suspiciousness scores reflect how likely it is for each program entity to be faulty, and can be used to sort program entities. Then, developers can follow the suspiciousness rank list (from the beginning to the end of the list) to manually inspect source code to diagnose the actual root cause of failures. Recently, SBFL techniques have also been utilized by various automated program repair techniques to localize potential patch locations [18, 32, 34, 37, 50, 60].

The advantage of *spectrum-based fault localization* is quite obvious – it is an extremely lightweight approach that is scalable and applicable for large-scale programs. An ideal fault localization

ISSTA'17, July 2017, Santa Barbara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092731

technique would always rank the faulty program entities at the top. However, in practice, although various SBFL techniques have been proposed (such as Jaccard/Ochiai [6], Op2 [40], and Tarantula [24]), no technique can always perform the best – the developers usually have to check various false-positive faults before finding the real one(s). We believe the current form of spectrum analysis is a key reason that limits the effectiveness of all existing SBFL techniques. Although different existing SBFL techniques use different formulae for suspiciousness computation, they all only consider how to differentiate program *source code entities (i.e., one dimension in program spectra)*. Our key insight is that a richer form of spectrum analysis that additionally considers how to differentiate *tests (i.e., the other dimension in program spectra)*, which, to the best of our knowledge, has not been studied in previous work, can provide more effective fault localization. For instance, consider two tests `t1` and `t2` that are both failing tests such that `t1` covers 100 program entities while `t2` only covers one. By our intuition, `t2` can be much more helpful than `t1` in fault localization since `t2` has a much smaller search space to localize the fault(s). However, the traditional SBFL techniques ignore this useful information and consider `t1` and `t2` as making the same contribution on SBFL, e.g., a program entity executed by `t1` or `t2` once will be treated the same regardless of the number of entities covered by the tests.

To overcome the limitations of existing spectrum-based fault localization techniques, we utilize the existing program spectra more effectively by explicitly considering the contributions of different tests. Based on our insight, we present PRFL, a lightweight PageRank-based technique that boosts spectrum-based fault localization by considering the additional test information via PageRank algorithm [43]. PRFL collects the connections between tests and source code entities (e.g., the traditional spectrum information) as well as the connections among source code entities (e.g., the static call graph information) via bytecode instrumentation and analysis. Then, PageRank is used to recompute the program spectrum information: (1) program entities connected with more important failing tests (which cover smaller number of program entities) may be more suspicious, and (2) program entities connected with more suspicious program entities may also be more suspicious since they may have propagated the error states to the connected entities. Finally, PRFL employs existing SBFL ranking formulae to compute the final suspiciousness score for each program entity. We have used our PRFL prototype to localize the faulty methods for 357 real faults in the Defects4J [25] benchmark. Since mutation faults have also been shown to be suitable for software testing experimentation [9, 26], to further validate the effectiveness of the proposed approach, we applied it to localize 30692 mutation faults generated from 87 GitHub Java projects. The experimental results demonstrate that our technique can outperform state-of-the-art SBFL techniques significantly (e.g., ranking 42%/55% more real/artificial faults within Top-1) with negligible overhead (e.g., around 2 minute average overhead on real faults). Overall, this paper makes the following contributions:

- **Simple Idea.** We propose a simple idea that considers the different contributions of different tests to further boost spectrum-based fault localization.
- **Lightweight Technique.** We implement the proposed idea as a lightweight fault localization technique, PRFL, that uses PageRank to consider the weights of different tests to enhance spectrum-based fault localization.
- **Extensive Evaluation.** We evaluate our PRFL on both real and artificial faults. Firstly, we evaluate our approach on 357 real faults from 5 projects in the Defects4J benchmark. To reduce the threats to external validity, we further evaluate PRFL on 30692 mutation faults of 87 GitHub projects. Both results demonstrate the effectiveness and efficiency of the proposed technique.

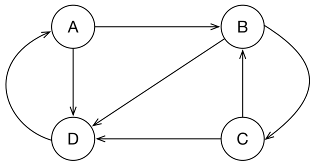
Table 1: Spectrum-based fault localization techniques and definitions

Tech	Defn	Tech	Defn
Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$	SBI	$1-\frac{e_p}{e_p+e_f}$
Ochiai	$\frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}}$	Jaccard	$\frac{e_f}{e_f+e_p+n_f}$
Ochiai2	$\frac{e_f n_p}{\sqrt{(e_f+e_p)(n_f+n_p)(e_f+n_p)(n_f+e_p)}}$	Kulczynski	$\frac{e_f}{n_f+e_p}$
Op2	$e_f-\frac{e_p}{e_p+n_p+1}$	Dstar2	$\frac{e_f^2}{e_p+n_f}$

2 BACKGROUND

2.1 Spectrum-Based Fault Localization

Spectrum-based fault localization techniques (SBFL) [6, 24, 40, 57] help developers identify the locations of faulty program entities (such as statements, basic blocks, and methods) based on observations of failing and passing test executions. A SBFL technique sorts all program entities by their suspiciousness scores and returns a rank list for manual checking. If a program entity is more likely to be faulty, it will be assigned a higher priority in the suspiciousness list. Therefore, an ideal SBFL technique should always rank the faulty entity with high suspiciousness score, which can significantly speed up the debugging process for finding the root causes of test failures. To compute suspiciousness scores of program entities, a SBFL technique firstly runs tests on the target program and records the program spectrum of each failing or passing test, i.e., the run-time profiles about which program entities are executed by each test. Then, based on the program spectra and test outcomes, various statistics can be extracted for suspiciousness computation, e.g., tuple (e_f, e_p, n_f, n_p), where e_f and e_p are the numbers of failing and passing tests executing the program entity e, while n_f and n_p are the numbers of failing and passing tests that do not execute the program entity e. Based on such tuples, various SBFL formulae have been proposed. The common intuition of these formulae is that a program entity executed by more failing tests and less passing tests is more likely to be faulty. This paper considers 8 well-studied SBFL techniques – Tarantula, Statistical Bug Isolation (SBI), Ochiai, Jaccard, Ochiai2, Kulczynski, Op2, and Dstar2 [6, 24, 33, 40, 57]. Tarantula, SBI, Ochiai and Jaccard [61, 62] are the most widely-used techniques for the evaluation of fault localization. Ochiai2 [40] is an extension version of Ochiai, which considers the impact of non-executed or passing test cases. Op2 [40] is the optimal SBFL technique for single-fault program, whereas Kulczynski and Dstar2 belong to the formula family Dstar [57], which is shown to be more effective than 38 other SBFL techniques. All their formulae are listed in Table 1.



	A	B	C	D
A	0	0	0	1
B	0.5	0	0.5	0
C	0	0.5	0	0
D	0.5	0.5	0.5	0

Figure 1: A small network and its transition matrix

2.2 PageRank Algorithm

PageRank [43] is a link analysis algorithm proposed by Larry Page and Sergey Brin for improving search quality and speed. PageRank views the World Wide Web as a set of linked nodes and ranks them based on their importance. The intuition behind PageRank is, for each node, if it is linked by important nodes, it should be more important than the ones linked by uninfluential nodes. Figure 1 presents a simple directed graph to describe a small network with four web pages, denoted by node A, B, C and D. The edges between two nodes denote that the starting node contains a hyperlink pointing to the ending node.

By our observation, the number of edges pointing to D is larger than others, so it should be more important than others. On the other hand, A is pointed by D and thus is also an important node according to the assumption of PageRank. Then, B is in turn pointed by A, and also should be assigned a high score to show the importance. Formally, the websites are described by a directed graph $G = \langle V, E \rangle$ with n nodes and m edges. Let $\boldsymbol{P}$ be the transition matrix of n by n elements. Then, each matrix element, P_{ij}, denotes the probability of transitioning from node j to i and its value is $\frac{1}{\text{Outbound Link Number of Node } j}$, as illustrated in Figure 1.

According to our intuition, the PageRank score of node i depends on the PageRank scores of the nodes with edges pointing to node i. Therefore, the PageRank score of node i can be computed as:

$$PR_i = \sum_{\forall j, j \to i} \frac{PR_j}{\text{Outbound Link Num of Node } j} \tag{1}$$

In order to make the equation more compact, we use PageRank vector $\vec{x}$ to present the PageRank score for each node and $\vec{x}$ is the solution of the eigenvalue equation:

$$\vec{x} = \boldsymbol{P} \cdot \vec{x} \tag{2}$$

In some cases, a node may have no outbound links and its PageRank score cannot be distributed to others. Considering these special nodes, an additional teleportation vector $\vec{v}$ weighted by the damping parameter d is attached to Equation(2):

$$\vec{x} = d \cdot \boldsymbol{P}\vec{x} + (1 - d) \cdot \vec{v} \tag{3}$$

where $\vec{v}$ is a positive vector and $\sum v_i$ is 1. When the network scale grows, it is harder to find the exact solution for the above equation in a reasonable time. Therefore, Page et al. [43] introduced an iterative approach to get the approximate solution. The equation for the k^{th} iteration is defined as:

$$\vec{x}^{(k)} = d \cdot \boldsymbol{P}\vec{x}^{(k-1)} + (1 - d) \cdot \vec{v} \tag{4}$$

and the initial value of $\vec{x}$ can be set as $\vec{v}$ or $\vec{0}$. For the example in Figure 1, if we use damping coefficient $d = 0.85$, vector $\vec{v} = [\frac{1}{n}, \frac{1}{n}, ..., \frac{1}{n}]^T$, and the initial PageRank vector $\vec{x}^{(0)} = \vec{v}$, after 25 iterations, the PageRank scores of nodes A, B, C and D are 0.3134, 0.2278, 0.1343 and 0.3246, respectively. These scores indicate the importance of each node. Recall that D is pointed by all others and becomes the most important node. A is the only node pointed by D, hence it is the second important node in the network. A and C together have two outbound links pointing to B, whose importance is lower than A. C's score is the lowest since it only has one inbound link from B.

Not only can PageRank rank web pages, but it also has been widely applied to various other domains. Gleich [21] surveyed the diversity of applications of PageRank and concluded that PageRank can be applied to Chemistry, Biology and Bioinformatics, Neuroscience, Bibliometrics, Databases and Knowledge Information Systems, Recommender Systems, Social Networks Web, i.e., twelve domains in total. Recently, PageRank-based techniques have also been proposed to analyze software systems. Chepelianskii [14] used PageRank to analyze function importance for Linux kernel. Kim et al. [27] proposed MonitorRank, a PageRank-based approach to find root causes of anomalies in service-oriented architectures. Bhattacharya et al. [11] proposed the notion of NodeRank based on PageRank to measure the importance of nodes on a static graph for software analysis and fault prediction. Later on, Mirshokraie et al. [38] proposed the notion of FunctionRank, a dynamic variant of PageRank, for ranking functions in terms of their relative importance, for mutation testing. To the best of our knowledge, this work is the first to apply the PageRank algorithm for fault localization.

3 MOTIVATING EXAMPLE

Spectrum-based fault localization techniques are designed based on program execution statistics, which include both test coverage and test outcomes. Execution statistics can be treated as the information source of SBFL's analysis, so that they determine the upper bound of SBFL's accuracy. When execution statistics is collected, SBFL will distribute them to construct program spectra, then apply various ranking formulae to compute suspiciousness score for each program entity. Program spectrum is a practical way to present execution statistics, however, it is risky since it loses useful information during construction. Here an example will be analyzed to show how program spectra affect the accuracy of fault localization.

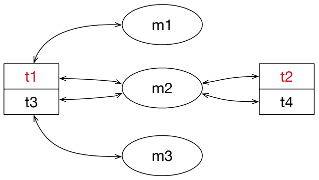
Shown in Figure 2, in example class `Code`, method `m2` is faulty since its conditional expression should be `(x > 10)` instead of `(x > 1)`. This fault leads `t1` and `t2` to fail. Based on the spectrum information in the left half of Table 2, the traditional Tarantula technique would compute all the suspiciousness scores of `m1`, `m2` and `m3` as the same, i.e., 0.5. This result is no better than random guess, and thus it is not quite helpful for fault localization. However, when we observe the detailed test coverage shown in Figure 3, we can directly find that `m2` is faulty since `t2` fails and only covers `m2`. This example shows that different tests have different capabilities to locate faults, and one limitation of the original spectrum information is that it only focuses on computing how many failing and passing tests cover the program entities but ignores the test differences. This observation inspires us that if test can be weighted based on their capabilities in localizing potential faults, the spectrum-based fault localization will be more accurate.

Here we just analyze failing tests. By our intuition, the test weight should be impacted by the test scope and the covered program entities. Firstly, if the failing test covers very few program

```
1  class Code{
2      static int m1(int x){
3          if (x >= 0)
4              return x;
5          else
6              return -x;
7      }
8      static int m2(int x){
9          if (x > 1)//buggy
10             return x;
11         else
12             return 0;
13     }
14     static int m3(int x){
15         return x * x;
16     }
17 }
```

```
public void t1() {
    int a = Code.m1(-2);
    int b = Code.m2(a);
    int c = Code.m3(b);
    assertEquals(0, c);
}
public void t2() {
    int a = Code.m2(5);
    assertEquals(0, a);
}
public void t3() {
    int a = Code.m2(15);
    int b = Code.m3(a);
    int c = Code.m1(b);
    assertEquals(225, c);
}
public void t4() {
    int a = Code.m2(30);
    assertEquals(30, a);
}
```

Figure 2: Example code and corresponding test suite



$$\left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 1/3 & 1 \\ 0 & 0 & 0 & 1/3 & 0 \\ \hline 1 & 1/2 & 1 & 0 & 0 \\ 0 & 1/2 & 0 & 0 & 0 \end{array}\right]$$

Figure 3: Test coverage graph of Code and the transition matrix of failing tests. Red means the tests that failed.

entities, then it has a small scope to infer faulty entities. Therefore its weight should be high. On the other side, if its covered entities are more likely faulty, it in turn also should get a higher weight. Similarly, if a program entity is covered by more highly weighted tests, it should also be more likely to contain faults.

All above analysis is constructed on the bi-directional test coverage graph, a kind of network, hence test weight analysis can be solved by PageRank. Note that the failing and passing tests cover different set of entities, PageRank analysis will be executed twice to generate the scores of entity importance for failing and passing tests. We term these scores *faultiness* and *successfulness* scores respectively. For example, when computing the failing test weights, we uses vector $\vec{x} = [m_1, m_2, m_3, t_1, t_2]^T$ to present the node values where m_1, m_2 and m_3 present the faultiness scores of m1, m2 and m3 and t_1, t_2 show the test weights of t1 and t2. The test scopes can be presented by teleportation vector $\vec{v} = [0, 0, 0, w_1, w_2]^T$, where the first three 0s denote the corresponding three source methods and w_1 and w_2 are the weights of t1 and t2. They can be computed by $w_i = \frac{c_i^{-1}}{\sum c_j^{-1}}$, where c_i is defined as the number of program entities covered by the ith test. For this example, c_1 and c_2 are 3 and 1 respectively and $\vec{v}$ is $[0, 0, 0, 0.25, 0.75]^T$. The construction of transition matrix has been introduced in Section 2.2 and the matrix $\boldsymbol{P}$ can be found in Figure 3. Assume that the damping factor d is 0.7, $\vec{x}^{(0)}$ is $\vec{0}$, based on Equation(4), we can get $\vec{x}$ as $[0.061, 0.290, 0.061, 0.262, 0.326]^T$, where m_2 is larger than m_1 and m_3, indicating that m2 is highly connected with failed tests and thus is more likely to be faulty, while w_1 is less than w_2, indicating that t2 is more effective to help with fault localization. This result reflects that PageRank analysis computes faultiness score for each method and distributes weights for tests in tandem. In the next step, we only utilize faultiness scores to construct weighted spectra since they already include the information from test weights. The weighted spectra can be computed by normalized faultiness score $\hat{m}_i = \frac{m_i}{max(\{m_j\})}$. In this example, only failing tests are considered, so only e_{fi} and n_{fi} need to be updated as $\hat{e}_{fi} = \hat{m}_i \cdot N_f$ and $\hat{n}_{fi} = N_f - \hat{e}_{fi}$, where N_f is the total number of failing tests. The passing tests can be analyzed in the similar way, whose details can be found in Section 4.2. The right half of Table 2 shows the weighted spectrum information and updated Tarantula scores for each Code method. According to the table, PRFL boosts Tarantula to rank m2 as the first, demonstrating the effectiveness of PageRank for fault localization.

Table 2: Original and weighted spectra of Code, T denotes Tarantula score.

Program Entity	Original Spectrum Info					Weighted Spectrum Info				
	e_f	e_p	n_f	n_p	T	e_f	e_p	n_f	n_p	T
m1	1	1	1	1	0.5	0.42	1	1.58	1	0.30
m2	2	2	0	0	0.5	2	2	0	0	**0.50**
m3	1	1	1	1	0.5	0.42	1	1.58	1	0.30

Actually, although PRFL can help with Tarantula with the above example, some other traditional formulae, e.g., Ochiai, actually can also rank the faulty method precisely. Therefore, we further show another example. Suppose method m1 is also faulty by changing Line 3 to if (x $\geq$ 5), and also t4 is modified as:

```
public void t4() {
    int a = Code.m1(4) + 5;
    int b = Code.m3(a);
    assertEquals(81, b);
}
```

Since now both m1 and m2 are faulty, tests t1, t2 and t4 failed and only t3 passed. This result leads all three methods to share the same traditional spectrum information, i.e., (e_f, e_p, n_f, n_p)=$(2, 1, 1, 0)$ – no matter which SBFL formula is applied, all the methods will be ranked with the same suspiciousness. However, as we analyzed before, m2 is clearly a fault since it is the only method covered by the failed t2. Based on our PRFL idea, we can get vector $\vec{x}$ as $[m_1, m_2, m_3, t_1, t_2, t_4]^T$=$[0.099, 0.213, 0.099, 0.198, 0.238, 0.151]^T$. This outcome shows that firstly, m2 is more likely to be faulty than m1 and m3 since 0.213 is greater 0.099. Secondly, as we expected, m1 and m3 has the same score(0.099) and the reason is that both of them are covered by the same tests. Thirdly, t2 is the most important (e.g., with the highest weight 0.238) test among t1, t2 and t4 since it only covers m2. Moreover, t1 is more important than t4 since it covers more faulty methods. This multiple-fault example further demonstrates the effectiveness of PageRank for fault localization.

4 APPROACH

Spectrum-based fault localization techniques [6, 24, 40, 57] utilize program spectra to record execution traces and test results, and use various ranking formulae to localize faults. However, as we discussed in Section 3, program spectra may lose useful information and negatively affect the effectiveness of SBFL. To alleviate this issue, we present a novel approach, PRFL, to refine spectrum information in order to improve the effectiveness of SBFL techniques. Note that PRFL is a lightweight technique that generates more informative spectrum and traditional SBFL techniques can be directly applied on top of the new program spectrum information. In theory, our PRFL idea can be applied to localize faults at different granularities. Previous studies demonstrated that statement-level fault localization may be too fine-grained and miss useful context information [48], while class-level fault localization is too coarse-grained and cannot help understand and fix the fault within a class [56]. Therefore, following recent work on fault localization [10, 31], we also focus on method-level fault localization, i.e., localizing faulty methods among all source code methods.

PRFL consists of three major phases: preparation, PageRank analysis and ranking. Similar with most approaches in traditional spectrum-based fault localization, the input of PRFL is a faulty program and its corresponding test suite (with failing test(s) revealing the fault(s)). In the *preparation* phase (Section 4.1), PRFL first collects execution traces and test results from the faulty program by running tests. Each trace records the test name, test result (failing or passing) and a series of executed methods of the faulty program. Besides the connections between tests and program methods, PRFL further applies static analysis to construct the connections between program methods, i.e., the call graph. In the *PageRank analysis* phase (Section 4.2), PRFL considers the connections between failing tests and methods (i.e., coverage for failing tests) as well as the connections among source methods (i.e., the call graph information), and applies PageRank to the connections to generate the new spectrum information for failing tests. Similarly, PRFL also updates the spectra for passing tests. In the final *ranking* phase (Section 4.3), PRFL takes as input the updated weighted spectra and uses existing ranking formulae to localize faulty methods.

4.1 Preparation Phase: Static and Dynamic Analysis

The preparation phase in PRFL is designed to collect graph data from both dynamic execution and source code for the PageRank analysis. In this phase, PRFL traces test coverage graph by dynamic analysis. When each test is running, the dynamic analysis performs code instrumentation to automatically record which methods are executed by the test. Note that different methods may be covered by the same failing and passing tests, which means they cannot be differentiated only by the coverage graph; hence, more information should be mined to alleviate this issue. Our heuristic is that if two methods have the same test coverage (e.g., the same method and test connections), the connections between the tied methods and other methods can help break the tie (e.g., a method connected with more fault-prone methods may also be fault-prone since it may have propagated the error states to the connected methods). Therefore, for each method, PRFL further applies static analysis to extract the static call graph to obtain the connections among source methods. The constructed call graph will then be combined with test coverage for the PageRank analysis.

4.2 Analysis Phase: PageRank Propagation

Here we first recall the original PageRank equation (Equation(3)). The PageRank equation consists of three elements, i.e., transition matrix $\mathbf{P}$, damping factor d, and teleportation vector $\vec{v}$. To be specific, damping factor d and teleportation vector $\vec{v}$ are parameters and transition matrix $\mathbf{P}$ is constructed based on coverage and call graph.

4.2.1 *Transition Matrix Construction.*

As we have illustrated in Section 3, given the test coverage information (i.e., the connections between tests and methods), the transition matrix can be partitioned as:

$$\boldsymbol{P} = \left[\begin{array}{c|c} \boldsymbol{0} & \boldsymbol{P_{TM}} \\ \hline \boldsymbol{P_{MT}} & \boldsymbol{0} \end{array}\right] \tag{5}$$

where $\boldsymbol{P_{MT}}$ and $\boldsymbol{P_{TM}}$ denote the transition matrix between methods and tests (based on the test coverage graph). Note that since the test coverage graph is bipartite, the sub-matrices on top left and bottom right are zero-matrices.

The matrix $\boldsymbol{P}$ can present the differences between methods based on test coverage; however, some methods may be covered by the same failing and passing tests and they cannot be differentiated only by $\boldsymbol{P}$. To alleviate this issue, we utilize the Class Hierarchy Analysis (CHA) call graph algorithm [17] to further distinguish these methods. The transition matrix of call graph is constructed in a different way. Firstly, we convert the call graph to adjacent matrix $\mathbf{A}$ as follows: suppose method m_i invokes method m_j, one edge would be added from m_i to m_j with weight 1. Also, m_j will return to m_i when finished and this return relation should also be considered. Heuristically, the calling edges may be more important than the return edges, thus we assign smaller weight, δ, on the return edge from m_j to m_i, i.e., A_{ij} and A_{ji} can be computed as $A_{ij} = 1$ and $A_{ji} = \delta$. Note that when m_j and m_i invoke each other (e.g., due to recursion), both A_{ij} and A_{ji} are equal to $1 + \delta$. Finally, $\mathbf{A}$ will be column-normalized to $\hat{\boldsymbol{A}}$ by:

$$\hat{A}_{ij} = \frac{A_{ij}}{\sum_k A_{ik}} \tag{6}$$

and the method-to-method matrix (i.e., the transition matrix of call graph), $\boldsymbol{P_{MM}}$, can be computed as $\boldsymbol{P_{MM}} = \hat{\boldsymbol{A}}^T$.

4.2.2 *Teleportation Vector Design.*

As we have discussed before, test weight is not only impacted by the faultiness or successfulness likelihood of covered methods, but also by the test scope. Hence, PRFL uses different teleportation vectors to present the impact of test scope. Teleportation vector $\vec{v}$ is composed by two sub-vectors: $\vec{v} = \begin{bmatrix}\vec{v}_m^T & \vec{v}_t^T\end{bmatrix}^T$, where $\vec{v}_m$ and $\vec{v}_t$ denote the teleportation vector of methods and tests respectively. For both failing and passing tests, $\vec{v}_m$ is $\vec{0}$. For failing tests, $\vec{v}_t$ is $[w_1, w_2, ..., w_m]^T$, where $w_i = \frac{c_i^{-1}}{\sum c_j^{-1}}$ and c_i denotes the number of methods covered by test i. This setting is based on the property that if a test failed, it covers at least one faulty method so that a failing test with smaller scope is more helpful to locate the faults. However, a passing test does not satisfy this property since it may cover a faulty method whose fault is not triggered. Therefore, the scope of a passing test does not depend on the number of covered methods and all test scopes share the same weight. Due to the normalization of weight w_i, $\vec{v}_t$ for passing tests is defined as $[\frac{1}{m}, \frac{1}{m}, ..., \frac{1}{m}]^T$, where m denotes the number of the passing tests.

4.2.3 Constrained PageRank Algorithm. In this work, we extended the standard PageRank algorithm [43] to analyze the integration of test coverage and call graph. The standard PageRank algorithm is applied to a graph whose nodes and edges are all congeneric. However, in our application scenario, this prerequisite is not satisfied since the edges in test coverage graph present the connections between methods and tests, which are not congeneric with the ones in call graph. Thus, the standard PageRank algorithm may be underperformed because of edge variance.

To overcome this issue, we propose a constrained PageRank algorithm to differentiate the edges based on their connection types. Our intuition is that the test coverage graph makes the main contribution for fault localization, so the edges in test coverage graph deserve more weights; on the other hand, the edges in call graph should be set less weights. We use parameter α to tune the weight of call graph, and the constrained transition matrix is:

$$\boldsymbol{P} = \left[\begin{array}{c|c} \alpha \boldsymbol{P_{MM}} & \boldsymbol{P_{TM}} \\ \hline \boldsymbol{P_{MT}} & \boldsymbol{0} \end{array}\right] \tag{7}$$

The PageRank vector $\vec{x}$ in Equation(3) can be decomposed as $\vec{x} = \left[\vec{x}_m^T \quad \vec{x}_t^T\right]^T$, where $\vec{x}_m$ denotes the method faultiness or successfulness scores and and $\vec{x}_t$ denotes the weights of tests. Then, the iterative Equation(3) can be updated as:

$$\vec{y}_m^{(k)} = d \cdot (\alpha \boldsymbol{P}_{MM} \cdot \vec{x}_m^{(k)} + \boldsymbol{P}_{TM} \cdot \vec{x}_t^{(k)}) \tag{8}$$

$$\vec{y}_t^{(k)} = d \cdot \boldsymbol{P}_{MT} \cdot \vec{x}_m^{(k)} + (1-d) \cdot \vec{v}_t \tag{9}$$

$$\vec{x}_m^{(k+1)} = \frac{\vec{y}_m^{(k)}}{max(\vec{y}_m^{(k)})} \tag{10}$$

$$\vec{x}_t^{(k+1)} = \frac{\vec{y}_t^{(k)}}{max(\vec{y}_t^{(k)})} \tag{11}$$

where the both initial settings of $\vec{x}_m$ and $\vec{x}_t$ are $\vec{0}$.

4.3 Ranking Phase: Weighted-Spectrum-based Fault Localization

The weighted spectrum can be constructed using faultiness, successfulness score and failing, passing test number. In the second phase , the *PageRank Analysis* is executed twice to generate vector $\vec{x}_{mf}$ and $\vec{x}_{ms}$, which consist of faultiness and successfulness scores of all covered methods. For each method m_i, its faultiness score s_{fi} and successfulness score s_{si} are extracted from $\vec{x}_{mf}$ and $\vec{x}_{ms}$ respectively, and the weighted spectrum can be computed as:

$$\begin{aligned} \hat{e}_{fi} &= s_{fi} \cdot N_f, \quad \hat{n}_{fi} = N_f - \hat{e}_{fi} \\ \hat{e}_{pi} &= s_{si} \cdot N_p, \quad \hat{n}_{pi} = N_p - \hat{e}_{pi} \end{aligned} \tag{12}$$

where N_f and N_p denote the total number of failing and passing tests, respectively. PRFL then applies SBFL formulae to compute suspiciousness scores on weighted spectra and ranks all method. The weighted spectra include information not only from test coverage, but also from test scope and call graph information. Therefore, the weighted spectra are more accurate to reflect the method faultiness and successfulness, and can consequently boost the effectiveness of SBFL techniques.

Table 3: Subject statistics

ID	Program	#Faults	LoC	#Tests
Chart	JFreeChart	26	96K	2,205
Closure	Closure Compiler	133	90K	7,927
Lang	Commons Lang	65	22K	2,245
Math	Commons Math	106	85K	3,602
Time	Joda-Time	27	28K	4,130
Real-Bug Total	5 Projects	357	321K	20,109
Mutation-Bug Total	87 Projects	30692	967K	10,364

5 EXPERIMENTAL SETUP

Our experimental study aims to answer the following research questions:

- **RQ1:** How does PRFL compare with traditional SBFL techniques in term of effectiveness and efficiency?
- **RQ2:** How do different configurations impact the effectiveness of PRFL?
- **RQ3:** How do different numbers of faults impact the effectiveness of PRFL?
- **RQ4:** How does the fault type (e.g., real or artificial faults) impact the effectiveness of PRFL?

5.1 Subjects

Real-Fault Subjects Defects4J [25] is a mature real fault dataset for testing experiments, and has been widely used in software testing research [10, 26, 29, 49, 52]. Defects4J includes 357 real faults from 5 open-source projects in August 2016: JFreeChart, Google Closure Compiler, Apache Commons Lang, Apache Commons Math and Joda-Time. For each fault, Defects4J provides the faulty program, the fixed program with minimum code change, the failing tests and modified source files. We identify the faulty methods in the following ways. Firstly, we compare the modified source files to collect code changes. If all changes are located in a single method, we label such method as a faulty method. However, in some other cases, the program changes are distributed in multiple methods which may not all be faulty. To precisely identify the actual fault-triggering methods, we then manually apply all the possible combinations of the modified methods to get the minimum change set that can pass all tests. Note that we used all the 357 Defects4J faults except the faults not within method bodies. Table 3 (except the last row) shows the statistics for the Defects4J subjects – Column 1 presents the subject IDs that will be used in the remaining text; Column 2 presents the full name for the subjects; Column 3 presents the number of faults for each subject; finally, Columns 4 and 5 present the LoC (i.e., Lines of Code) and test number information for the most recent version of each subject in Defects4J.

Artificial-Fault Dataset Although Defects4J is great for evaluating testing techniques, its projects and faults are rather limited, posing threats to external validity. Meanwhile, mutation faults have been shown to be suitable for software testing experimentation [9, 26]. Therefore, we further use the PIT mutation testing tool [5] to generates artificial faults for evaluating PRFL. To be specific, we start from the first 1000 most popular Java projects from GitHub [2]. 226 projects of those were built successfully with Maven and passed all tests. Then, 139 projects were further removed due since PIT crashed or it could not terminate within our time limit, i.e., 2 hours. Therefore, finally, we have 87 projects with mutation faults, ranging from 163 to 165067 lines of code. The last row of Table 3 presents the statistics of the mutation faults used in our study.

5.2 Implementation and Supporting Tools/Platform

Data Preparation We use ASM bytecode analysis framework [1] together with JavaAgent [3] to perform on-the-fly code instrumentation to capture the test coverage for each test. Furthermore, we also implement the static Class Hierarchy Analysis (CHA) call graph algorithm [17] based on ASM framework. Note that we ignore all the 3rd party libraries and Java internal libraries during the call graph analysis for time efficiency.
Data Analysis We use Numpy [4], one of the most popular scientific computing package in Python, to implement and evaluate PRFL and other traditional SBFL techniques. PRFL applies an iterative algorithm to compute the faultiness and successfulness scores. For our application scenario, the test number is relative small, which leads the sizes of transition matrices $\boldsymbol{P}_{TM}$ and $\boldsymbol{P}_{TM}$ limited. On the other hand, since the method invocations are not frequent, the transition matrix $\boldsymbol{P}_{MM}$ is sparse. This property makes PRFL execute fast, and for all following experiments, PRFL is iterated 25 times for both failing and passing tests.
Platform All our experiments were performed on a platform with 4-core Intel Core i7-6700 CPU (3.40 GHz) and 16 Gigabyte RAM on Ubuntu Linux 16.04.

5.3 Evaluation Metrics

We use the absolute wasted effort (AWE) and Top-N, two widely used metrics [10, 61, 62] to evaluate the effectiveness of the studied fault localization techniques. Note that all our metrics do not consider test code.

AWE: Given a faulty program and a ranking formula (such as Tarantula), AWE is defined as the ranking number of the faulty method. However, in some cases, there are more than one method sharing the same suspiciousness score with the faulty method, and AWE is defined as the average ranking of all the tied methods. The AWE is computed as:

$$AWE(b) = |\{m|susp(m) > susp(b)\}| + |\{m|susp(m) = susp(b)\}|/2 + 1/2$$

where b is the faulty method and m is any candidate method except b and $|\{\cdot\}|$ is the cardinality of a set. The range of AWE is from 1 to the total number of methods. A smaller AWE means the fault localization is more effective and the ideal value is 1.

Top-N: This metric counts the number of successfully localized faulty methods within the top-N (N=1, 3, 5) ranked results. If the faulty methods share the same score, we use the average position to present fault location. Higher Top-N denotes more effective fault localization. Note that this metric can be quite important in practice since developers usually only inspect top-ranked elements, e.g., over 70% developers only check Top-5 ranked elements [28].

All our experimental data and scripts are available at: https://bitbucket.org/zms0617/prfl.

Table 4: Fault localization results on all Defects4J bugs

	Tech	Top-1		Top-3		Top-5		AWE		
		S	P	S	P	S	P	S	P	Impr.
Chart	Tarantula	6	6	20	22	22	24	10.42	9.28	(10.89%)
	SBI	6	6	20	22	22	24	10.42	9.28	(10.89%)
	Ochiai	6	10	17	20	19	24	8.27	7.55	(8.66%)
	Jaccard	6	10	17	20	20	23	8.42	7.88	(6.42%)
	Ochiai2	6	11	17	20	21	23	8.54	8.26	(3.32%)
	Kulczynski	6	10	17	20	20	23	8.42	7.88	(6.42%)
	Dstar2	5	9	16	19	19	22	9.51	7.64	(19.74%)
	Op2	5	9	14	16	16	19	46.51	44.31	(4.74%)
Lang	Tarantula	20	26	47	53	61	61	5.20	4.93	(5.24%)
	SBI	20	26	47	53	61	61	5.20	4.93	(5.24%)
	Ochiai	23	31	48	55	59	62	4.82	4.45	(7.81%)
	Jaccard	22	29	48	54	60	61	4.86	4.58	(5.87%)
	Ochiai2	22	29	49	54	59	61	4.84	4.93	(-1.74%)
	Kulczynski	22	29	48	54	60	61	4.86	4.58	(5.87%)
	Dstar2	24	31	49	54	59	62	4.83	4.36	(9.81%)
	Op2	24	30	49	53	60	60	4.96	4.51	(9.03%)
Math	Tarantula	23	34	63	67	75	82	17.56	16.08	(8.45%)
	SBI	23	34	63	67	75	82	17.56	16.08	(8.45%)
	Ochiai	24	36	63	67	75	84	19.33	18.05	(6.63%)
	Jaccard	24	36	64	68	75	82	18.46	16.64	(9.85%)
	Ochiai2	24	36	64	69	75	82	17.94	16.43	(8.41%)
	Kulczynski	24	36	64	68	75	82	18.46	16.64	(9.85%)
	Dstar2	24	36	63	67	75	83	19.93	17.80	(10.69%)
	Op2	23	32	55	67	70	82	21.99	19.45	(11.55%)
Time	Tarantula	5	7	11	14	16	17	20.09	19.26	(4.10%)
	SBI	5	7	11	14	16	17	20.09	19.26	(4.10%)
	Ochiai	6	8	11	12	18	17	18.24	16.79	(7.90%)
	Jaccard	5	7	9	12	16	17	20.26	19.35	(4.50%)
	Ochiai2	5	7	11	14	16	17	20.21	19.29	(4.51%)
	Kulczynski	5	7	9	12	16	17	20.26	19.35	(4.50%)
	Dstar2	6	8	11	12	12	12	20.03	18.09	(9.69%)
	Op2	8	5	12	12	14	17	49.76	45.41	(8.75%)
Closure	Tarantula	12	15	31	38	41	49	126.57	100.19	(20.84%)
	SBI	12	15	31	38	41	49	126.57	100.19	(20.84%)
	Ochiai	14	19	33	41	44	57	116.88	85.80	(26.59%)
	Jaccard	13	17	31	40	42	49	125.74	99.37	(20.97%)
	Ochiai2	13	17	30	39	42	50	126.23	99.87	(20.88%)
	Kulczynski	13	17	31	40	42	49	125.74	99.37	(20.97%)
	Dstar2	14	20	32	41	44	56	116.49	85.33	(26.75%)
	Op2	17	13	36	23	46	43	124.61	97.45	(21.79%)
Overall	Tarantula	66	88	172	194	215	233	35.97	29.95	(16.74%)
	SBI	66	88	172	194	215	233	35.97	29.95	(16.73%)
	Ochiai	73	104	172	195	215	244	33.51	26.53	(20.83%)
	Jaccard	70	99	169	194	213	232	35.55	29.56	(16.83%)
	Ochiai2	70	100	171	196	213	233	35.55	29.76	(16.30%)
	Kulczynski	70	99	169	194	213	232	35.55	29.56	(16.83%)
	Dstar2	73	104	171	193	209	235	34.16	26.64	(22.00%)
	Op2	77	89	166	171	206	221	49.57	42.23	(14.81%)

6 RESULT ANALYSIS

6.1 RQ1: PRFL's Overall Effectiveness and Efficiency

To answer this RQ, we present the experimental results of PRFL using the default configuration (d=0.7, α=0.001 and δ=1.0) on all the real faults from the Defects4J dataset. Table 4 presents the overall results. In the table, different columns present different effectiveness metrics (for each metric, Column S represents the traditional spectrum-based techniques while Column P represents our PRFL) and different rows present the subjects and fault localization formulae used. Also, the bottom portion of the table presents the overall results for all the Defects4J subjects, e.g., the total Top-N values and the average AWE values. From the table, we can have the following observations. First, overall Ochiai and Dstar2 (marked in gray) are the two most effective SBFL techniques for all the faults in Defects4J. For example, they are the only two techniques with AWE values below 35.00. Also, both of them are able to localize 73 faults within

Table 5: Fault localization overheads

Sub	COV	CG	Analysis	Ranking	Total
Chart	35.18s	66.71s	0.42s	0.01s	102.32s
Closure	231.73s	431.71s	1.53s	0.01s	664.98s
Lang	23.85s	22.26s	0.14s	0.01s	46.26s
Math	268.32s	106.21s	0.82s	0.01s	375.36s
Time	21.56s	25.72s	0.34s	0.01s	47.63s
Avg.	116.13s	130.52s	0.65s	0.01s	247.31s

Top-1 and 170+ faults within Top-3. Second, in general, PRFL is able to boost all the studied traditional spectrum-based fault localization techniques. For example, the overall Top-1/3/5 and AWE values of all traditional techniques are all outperformed by the corresponding PRFL techniques. Third, interestingly, PRFL tends to boost more effective traditional spectrum-based fault localization techniques even more. For example, PRFL is able to boost the number of faulty methods ranked as Top-1 by Dstar2 from 73 to 104 (i.e., **42% more**), a higher improvement than the other inferior techniques.

We also recorded the overhead of our PRFL technique. Due to the space limitation, we only present the overhead for the first faulty version (i.e., the latest faulty version usually with the largest size) of each subject from Defects4J. Table 5 presents the overall results. In the table, each column presents the time spent in each phase of PRFL while the last column presents the total overhead; each row presents the overhead for each subject while the last row presents the average results for all the subjects. Note that coverage collection (i.e., Column `COV`) and suspiciousness computation/ranking (i.e., Column `Ranking`) are also required by traditional spectrum-based fault localization techniques. Therefore, only the call graph and PageRank analysis time (i.e., Column `CG` and Column `Analysis` colored in gray) is the extra overhead incurred by PRFL. Based on the table, the PRFL technique is very lightweight and can finish within 5 minutes for the studied medium- and large-sized subjects on average. Furthermore, the average extra overhead incurred by PRFL (i.e., the call graph and PageRank analysis time) is also quite low, e.g., 130 seconds for call graph analysis (which is close to the coverage collection time) and less than 1 second for the PageRank analysis. Therefore, our PRFL is a light-weight technique that can be rather efficient for real-world projects.

6.2 RQ2: Configuration Impacts

In this section, we extend our experiments with different configurations to investigate the influence of internal factors of PRFL so as to learn how to make PRFL achieve better performance. Figure 4 presents the impacts of different damping factors (i.e., d) on the effectiveness of PRFL using the default α=0.001 and δ=1.0. In the figure, the x axis presents various damping factor values, while the y axis presents the AWE improvements of PRFL techniques over the original pure spectrum-based techniques (different formulae are represented using different lines). From the figure, we have the following observations. First, the damping factor does not impact the PRFL effectiveness much. For example, for all the formulae on all the subjects, the largest improvement difference among different damping factors is only 4%. Second, for the majority cases, when the damping factor increases, the improvement rates slightly decrease. This observation is as expected. The reason is that when damping factor increases, the test scope will be distributed a smaller weight, causing it to make less contributions in localizing the faults.

Figure 5 shows the impact of the call graph weights (i.e., α) using the default d=0.7 and δ=1.0. Similar with Figure 4, the x axis presents different call graph weights, the y axis presents the improvement rates of PRFL over pure spectrum-based techniques (different line represents different formulae). From the figure, we have the following observations. First, on all the subjects, the improvement rates of PRFL dramatically increase at the very beginning, but then slowly increase or even decrease for some formulae. We think the reason to be that when call graph weight is 0, PRFL degrades to use only the test coverage information and cannot differentiate the methods with same test coverage, hence the improvement rates are relative low. Note that even without call graph information (i.e., α=0), PRFL is still able to outperform pure spectrum-based techniques for the majority cases. Second, the call graph weight has different impacts on different formulae. For example, the improvement rate of PRFL over Op2 eventually decreases dramatically for subjects Time and Closure, while the improvement rates keep stable or increasing for the other formulae on the most subjects. Also, the impact of call graph weight is similar for Tarantula and SBI, as well as Jaccard and Kulczynski due to the similarities in the formula definition.

Figure 6 shows the impact of the return edge weights (i.e., δ) using d=0.7 and α=0.001. Similar with Figure 4 and 5 , the x axis presents different return edge weights, while the y axis presents the improvement rates of PRFL over pure spectrum-based techniques. From this figure, we observe that the improvement rates of PRFL (especially on Op2) on all the subjects are quite low and sometimes even below zero when δ=0.0. We found the potential reason to be that when δ=0.0, all return edges are ignored in call graph, and the invoked methods are assigned with too much faultiness or successfulness scores. Especially, faultiness score plays a leading role in Op2 and can make callee methods more suspicious than the caller methods, thus decreasing the effectiveness of PRFL. Furthermore, interestingly, we also observe that is that the return edge weight does not impact the PRFL effectiveness much when $\delta \geqslant$0.1, e.g., the largest improvement difference among different return edge weights is no more than 5% for all the formulae on all the subjects.

6.3 RQ3: Impact of Fault Number

The real faults from Defects4J include both single-fault and multi-fault versions. To further investigate the impacts of fault number, we split the overall fault localization results of PRFL on real faults into single-fault and multi-fault results. Table 6 presents the main results. In the table, the left half presents the results on single-fault versions while the right half presents the results on the multi-fault versions. From the table, we have the following findings. First, we find that the traditional techniques perform differently on single and multiple faults. For example, Dstar2 and Op2 (marked in gray in the left half) are the two overall most effective techniques for single faults, while Tarantula and SBI (marked in gray in the right half) are the two overall most effective techniques for multiple faults. In particular, Op2 performs the best on single faults (also confirmed by prior work [40]), e.g., with the highest Top-1 value (i.e., 72) and the lowest AWE value (i.e., 24.28), but performs the worst on multiple faults, e.g., with the lowest Top-1 value (i.e., 5) and the highest AWE value (i.e., 93.00). We found the reason to be that Op2 is specifically designed and also shown to be optimal for single-fault

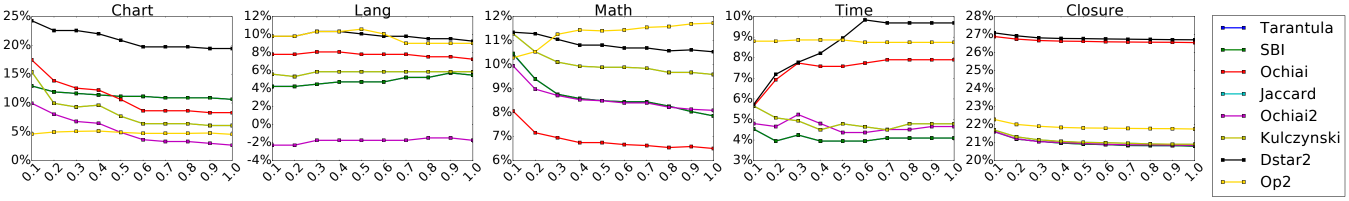


Figure 4: Impact of damping factor

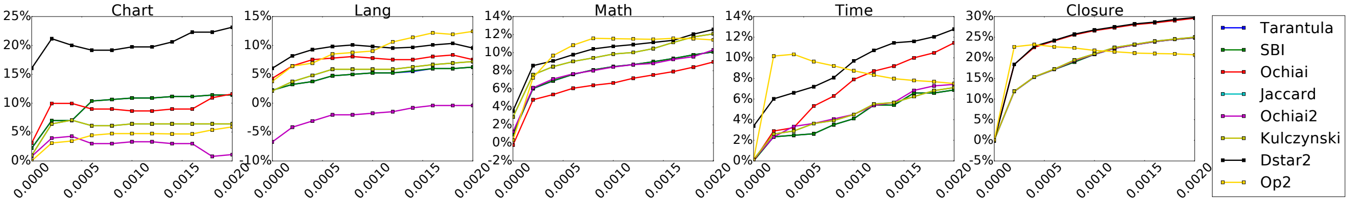


Figure 5: Impact of call graph weight

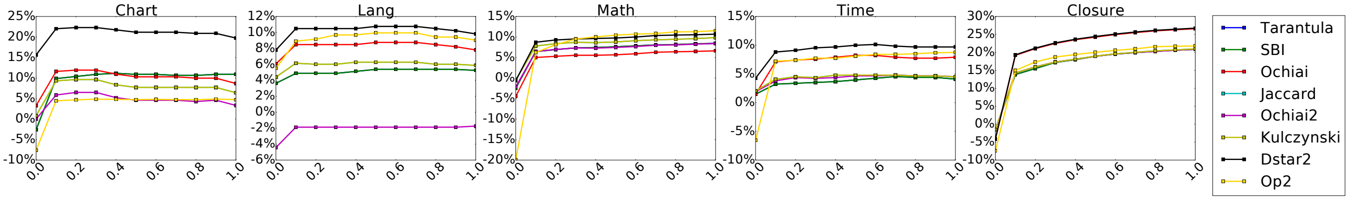


Figure 6: Impact of return weight

Table 6: Overall fault localization results on single and multiple Defects4J bugs

Tech	Single-bug versions									Multi-bug versions									
	Top-1		**Top-3**		**Top-5**		**AWE**			**Top-1**		**Top-3**		**Top-5**		**AWE**		
	S	P	S	P	S	P	S	P	Impr.	S	P	S	P	S	P	S	P	Impr.
Tarantula	57	72	121	135	150	161	31.25	25.41	(18.69%)	9	16	51	59	65	72	46.54	39.93	(14.20%)
SBI	57	72	121	135	150	161	31.25	25.41	(18.69%)	9	16	51	59	65	72	46.54	39.93	(14.20%)
Ochiai	64	84	127	140	156	171	26.40	19.95	(24.44%)	9	20	45	55	59	73	48.13	40.10	(16.68%)
Jaccard	61	79	123	137	151	162	29.95	24.38	(18.59%)	9	20	46	57	62	70	47.16	40.30	(14.55%)
Ochiai2	61	78	122	136	151	162	30.21	24.70	(18.24%)	9	22	49	60	62	71	46.79	40.33	(13.81%)
Kulczynski	61	79	123	137	151	162	29.95	24.38	(18.59%)	9	20	46	57	62	70	47.16	40.30	(14.55%)
Dstar2	65	85	128	141	156	171	26.30	19.66	(25.25%)	8	19	43	52	53	64	50.07	40.81	(18.48%)
Op2	72	77	135	143	163	182	24.28	18.00	(25.84%)	5	12	31	28	43	39	93.00	83.25	(10.49%)

programs [40], but it cannot perform well for multi-fault programs. Second, we find that despite the fact that various techniques perform differently on single or multiple fault programs, PRFL is able to boost all the studied techniques similarly on both single and multiple fault programs. For example, the Top-1 value improvement for Dstar2 is 31% (from 65 to 85) on single-fault programs and 138% (from 8 to 19) on multi-fault programs. Finally, PRFL tends to boost more effective techniques more for single faults. For example, the AWE improvement for Op2 (i.e., 25.84%) is the highest for single faults. For multiple faults, this rule does not hold anymore – the AWE improvement for two optimal techniques, Tarantula and SBI, is 14.20%, which is lower than that of Dstar2 (i.e., 18.48%).

Table 7: Fault localization results on mutation bugs

Tech	**Top-1**		**Top-3**		**Top-5**		**AWE**		
	S	P	S	P	S	P	S	P	Impr.
Tarantula	2712	3012	7751	8052	10982	11316	21.69	21.36	(1.50%)
SBI	2711	3010	7750	8048	10982	11313	21.74	21.41	(1.48%)
Ochiai	6570	10067	14710	16925	18036	19949	11.67	9.79	(16.06%)
Jaccard	6470	9934	14518	16673	17815	19638	12.04	10.26	(14.84%)
Ochiai2	6356	9564	14242	16036	17444	18980	13.24	11.82	(10.69%)
Kulczynski	6470	9934	14518	16673	17815	19638	12.04	10.26	(14.83%)
Dstar2	6700	10352	14987	17365	18293	20394	11.40	9.42	(17.44%)
Op2	7216	9590	15851	17352	19253	20670	10.37	8.61	(16.93%)

6.4 RQ4: Impact of Fault Type

So far we have studied the effectiveness of PRFL on real faults from the Defects4J dataset. In this section, we further study the effectiveness of PRFL on artificial mutation faults from other projects to reduce the threats to external validity. Table 7 presents our results on 30692 mutation faults from 87 GitHub projects. In the table, different rows present the results for different fault localization formulae, while different columns present the different metrics used. According to the table, we find that most effective techniques are the same with those for single real faults on Defects4J (shown in Table 6), i.e., Dstar2 and Op2 (marked in gray in the table), demonstrating the result consistency on real and mutation single faults (Note that mutation faults are all single faults since mutation testing generates one syntactic change for each mutant). Furthermore, PRFL also boosts the original most effective formulae to the most effective PRFL techniques. With the PRFL supports, Dstar2 and Op2 are again the most effective technique. For example, PRFL is able to boost the number of faults ranked as Top-1 by Dstar2 from 6700 to 10352 (e.g., 55% more), and reduce the AWE value of Op2 from 10.37 to 8.61 (16.93% more precise). These findings demonstrate that the effectiveness of PRFL is not impacted much by the fault type and subject programs used.

7 RELATED WORK

To the best of our knowledge, this paper is the first work to improve spectrum-based fault localization using PageRank. We list the related work in fault localization as follows.

Spectrum-Based Fault Localization Various formulae have been proposed for computing suspiciousness scores of program entities based on passing and failing test cases. Jones et al. [24] proposed the first foundational ranking formula, Tarantula, which is based on the intuition that program entries which are frequently executed by failing test cases and infrequently executed by passing test cases are more likely to be faulty. Dallmeier et al. [15] proposed Ample, an Eclipse plug-in for identifying faulty classes in Java software. Abreu et al. [6] designed Ochiai, which is also widely-studied and state-of-the-art ranking formula. Naish et al. [40] proposed the theoretical-best ranking formulae for single faults, Op and Op2, and empirically analyzed the existing ranking formulae on C programs. Yoo [63] generated a group of ranking formulae using genetic programming (GP). Xie et al. [59] summarized existing ranking formulae and theoretically compared them, finding that formulae of two families are optimal, including Op, Op2 and four GP-generated formulae. Lucia et al. [35] investigated the effectiveness of existing work and concluded that there is no best single ranking formula for all cases. Similarly, Steimann et al. [54] studied the threats to the validity in SBFL on ten open-source programs and showed that well-known fault locators do not uniformly perform better.

Machine-Learning-Based Fault Localization Several existing work has applied machine learning techniques to improve the accuracy of SBFL. Nath et al. [41] proposed TFLMs, a Relational Sum-Product Network model for fault localization. TFLMs could be learned from a corpus of faulty programs and localized faults in a new context more accurately. Feng et al. [20] proposed Error Flow Graph (EFG), a Bayesian Network to predict fault locations. EFG is constructed from the dynamic dependency graphs of the programs and then standard inference algorithms are employed to compute the probability of each executed statement being faulty. Xuan et al. [61] proposed Multric, which applied RankBoost, a pairwise learning-to-rank algorithm to combine 25 existing formulae. Roychowdhury et al. [51] utilized feature selection for fault localization and Le et al. [30] extended a standard feature selection to identify program entities which capture important characteristics of failing tests. Actually, our work can also be treated as an unsupervised-learning-based fault localization technique.

Mutation-Based Fault Localization Besides spectrum-based and machine-learning-based fault localization, there was one category of approach utilizing mutation analysis [44, 46, 47]. Papadakis et al. [45] firstly applied mutation testing to traditional fault localization. Zhang et al. [65] firstly applied mutation testing to localize faults during regression testing. Later on, Moon et al. [39] proposed MUSE, a mutation-based fault localization technique by analyzing mutant impacts on faulty and correct program entities. Hong et al. [23] developed new mutation operators as well as traditional operators to improve fault localization in real-world multilingual programs. There are also empirical studies evaluating mutation-based fault localization techniques [13, 49].

Slicing-Based Fault Localization Slicing technique is also widely used in fault localization [68, 69]. Zhang et al. [67] proposed a forward and a bidirectional dynamic slicing techniques for improving fault localization. Alves et al. [8] used dynamic slicing technique and change-impact analysis to prune irrelative code statements to improve Tarantula SBFL. Sinha et al. [53] focused on the fault localization of Java Runtime Exceptions. They combined dynamic analysis and static backward data-flow analysis to detect source statements which lead to exceptions. Xuan et al. [62] proposed to use program slicing to trim test cases into minimal fractions to achieve more precise spectrum-based fault localization. Gupta et al. [22] combined delta debugging which can identify a minimal failure-inducing input with forward and backward dynamic program slicing to narrow down probably faulty code for improving fault localization. Ocariza et al. [42] also proposed an automated technique to improve fault localization for Javascript code via backward slicing.

Other Fault Localization Techniques Campos et al. [12] applied entropy theory in the fitness function to extend existing test suites with new test cases in order to improve fault localization. Alipour et al. [7] extracted extended invariants such as execution features to improve fault localization. Zhang et al. [66] identified the causes of faults by switching predicates' outcome at runtime and altering the control flow. Yu et al. [64] introduced multiple kinds of spectrum types such as control and data dependences to build fault localization model. Le et al. [31] and Dao et al. [16] combined SBFL with information retrieval based fault localization, which recommends a set of program entities with similar contents of bug reports. Le et al. [10] employed likely invariants and suspiciousness scores to locate faults. In this work, they used Daikon's Invariant Diff [19] tool to mine changes in invariant sets between failing and passing program executions. They then applied learning-to-rank algorithm to predict fault positions.

8 CONCLUSION

Manual debugging remains costly and painful. Researchers have developed various techniques to automate debugging. A particularly well-studied class of techniques is spectrum-based fault localization (SBFL), which help developers infer the positions of faulty program entities. Despite the research progress, current SBFL techniques fail to deliver the promise they hold. In this paper, we propose PRFL, a novel approach to boost the accuracy of spectrum-based fault localization. PRFL uses PageRank algorithm to analyze the importance of each test, and then ranks methods by considering the corresponding test importance. We evaluated our approach on 357 real bugs and 30692 mutation bugs. The experimental results showed that PRFL outperforms existing state-of-the-art SBFL techniques significantly (e.g., ranking 42%/55% more real/artificial bugs within Top-1 compared with the most effective traditional SBFL technique), with low overhead.

9 ACKNOWLEDGMENT

This work is supported in part by NSF Grant Nos. CCF-1319688, CCF-1566589, CNS-1239498, UT Dallas start-up fund, and Google Faculty Research Award. We thank the anonymous reviewers and Nima Dini for the valuable comments.

REFERENCES

[1] 2017. ASM Java bytecode manipulation and analysis framework. (2017). http://asm.ow2.org/ Accessed: 01-30-2017.

[2] 2017. Github a web-based version control repository and Internet hosting service. (2017). https://github.com/ Accessed: 01-30-2017.

[3] 2017. Java programming language agents. (2017). https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html Accessed: 01-30-2017.

[4] 2017. Numpy package for scientific computing with Python. (2017). http://www.numpy.org/ Accessed: 01-30-2017.

[5] 2017. PIT mutation testing tool. (2017). http://http://pitest.org/ Accessed: 01-30-2017.

[6] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007.* IEEE, 89–98.

[7] Mohammad Amin Alipour and Alex Groce. 2012. Extended program invariants: applications in testing and fault localization. In *Proceedings of the Ninth International Workshop on Dynamic Analysis.* ACM, 7–11.

[8] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. 2011. Fault-localization using dynamic slicing and change impact analysis. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society, 520–523.

[9] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering.* ACM, 402–411.

[10] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ACM, 177–188.

[11] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. 2012. Graph-based analysis and prediction for software evolution. In *Software Engineering (ICSE), 2012 34th International Conference on.* 419–429.

[12] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. 2013. Entropy-based test generation for improved fault localization. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on.* IEEE, 257–267.

[13] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2016. Assessing and Comparing Mutation-based Fault Localization Techniques. *arXiv preprint arXiv:1607.05512* (2016).

[14] Alexei D Chepelianskii. 2010. Towards physical laws for software architecture. *arXiv preprint arXiv:1003.5455* (2010).

[15] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight bug localization with AMPLE. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging.* ACM, 99–104.

[16] Tung Dao, Lingming Zhang, and Na Meng. 2017. How Does Execution Information Help with Information-Retrieval Based Bug Localization?. In *ICPC*. To appear.

[17] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming.* Springer, 77–101.

[18] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis.* ACM, 30–39.

[19] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45.

[20] Min Feng and Rajiv Gupta. 2010. Learning universal probabilistic models for fault localization. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* ACM, 81–88.

[21] David F Gleich. 2015. PageRank beyond the Web. *SIAM Rev.* 57, 3 (2015), 321–363.

[22] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.* ACM, 263–272.

[23] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.* IEEE, 464–475.

[24] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering.* ACM, 467–477.

[25] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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.* ACM, 437–440.

[26] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 654–665.

[27] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. 2013. Root cause detection in a service-oriented architecture. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 41. ACM, 93–104.

[28] 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.* 165–176.

[29] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. 2016. Fine-tuning spectrum based fault localisation with frequent method item sets. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 274–285.

[30] Tien-Duy B Le, David Lo, and Ming Li. 2015. Constrained feature selection for localizing faults. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on.* IEEE, 501–505.

[31] 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.* ACM, 579–590.

[32] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[33] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *ACM SIGPLAN Notices* 40, 6 (2005), 15–26.

[34] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ACM, 166–178.

[35] Lucia 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 (2014), 172–219.

[36] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89 (2014), 51–62.

[37] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2016. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering* (2016), 1–29.

[38] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering* 41, 5 (2015), 429–444.

[39] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on.* IEEE, 153–162.

[40] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 11.

[41] Aniruddh Nath and Pedro Domingos. 2016. Learning tractable probabilistic models for fault localization. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence.* AAAI Press, 1294–1301.

[42] Frolin S Ocariza Jr, Karthik Pattabiraman, and Ali Mesbah. 2012. AutoFLox: An automatic fault localizer for client-side JavaScript. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* IEEE, 31–40.

[43] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[44] Mike Papadakis, Marcio E Delamaro, and Yves Le Traon. 2013. Proteum/fl: A tool for localizing faults using mutation analysis. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on.* IEEE, 94–99.

[45] Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate" unknown" faults. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* IEEE, 691–700.

[46] Mike Papadakis and Yves Le Traon. 2014. Effective fault localization via mutation analysis: A selective mutation approach. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing.* ACM, 1293–1300.

[47] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[48] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis.* ACM, 199–209.

[49] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating & improving fault localization techniques. In *ICSE'17, Proceedings of the 39th International Conference on Software Engineering.* Buenos Aires, Argentina. To appear.

[50] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, 254–265.

[51] Shounak Roychowdhury and Sarfraz Khurshid. 2011. Software fault localization using feature selection. In *Proceedings of the International Workshop on Machine*

Learning Technologies in Software Engineering. ACM, 11–18.

[52] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 201–211.

[53] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. 2009. Fault localization and repair for Java runtime exceptions. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 153–164.

[54] 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. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 314–324.

[55] Gregory Tassey. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project* 7007, 011 (2002).

[56] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 1–11.

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

[58] 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 (2016), 707–740.

[59] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. 2013. Provably optimal and human-competitive results in sbse for spectrum based fault localization. In *International Symposium on Search Based Software Engineering*. Springer, 224–238.

[60] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.

[61] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 191–200.

[62] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 52–63.

[63] Shin Yoo. 2012. Evolving human competitive spectra-based fault localization techniques. In *International Symposium on Search Based Software Engineering*. Springer, 244–258.

[64] Kai Yu, Mengxiang Lin, Qing Gao, Hui Zhang, and Xiangyu Zhang. 2011. Locating faults using multiple spectra-specific models. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 1404–1410.

[65] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*. 765–784.

[66] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*. ACM, 272–281.

[67] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2007. Locating faulty code by multiple points slicing. *Software: Practice and Experience* 37, 9 (2007), 935–961.

[68] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2007. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering* 12, 2 (2007), 143–160.

[69] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. 2005. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 33–42.