Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach

Yiling Lou*
HCST, CS, Peking University
Beijing, China
yiling.lou@pku.edu.cn

Ali Ghanbari
Xia Li
Lingming Zhang†
University of Texas at Dallas
Texas, USA
{Ali.Ghanbari,Xia.Li3,lingming.zhang}@utdallas.edu

Haotian Zhang
Ant Financial Services Group
Hangzhou, China
jingyun.zht@antfin.com

Dan Hao†
Lu Zhang
HCST, CS, Peking University
Beijing, China
{haodan,zhanglucs}@pku.edu.cn

ABSTRACT

A large body of research efforts have been dedicated to automated software debugging, including both automated fault localization and program repair. However, existing fault localization techniques have limited effectiveness on real-world software systems while even the most advanced program repair techniques can only fix a small ratio of real-world bugs. Although fault localization and program repair are inherently connected, their only existing connection in the literature is that program repair techniques usually use off-the-shelf fault localization techniques (e.g., Ochiai) to determine the potential candidate statements/elements for patching. In this work, we propose the *unified debugging* approach to unify the two areas in the other direction for the first time, i.e., *can program repair in turn help with fault localization?* In this way, we not only open a new dimension for more powerful fault localization, but also extend the application scope of program repair to all possible bugs (not only the bugs that can be directly automatically fixed). We have designed ProFL to leverage patch-execution results (from program repair) as the feedback information for fault localization. The experimental results on the widely used Defects4J benchmark show that the basic ProFL can already at least localize 37.61% more bugs within Top-1 than state-of-the-art spectrum and mutation based fault localization. Furthermore, ProFL can boost state-of-the-art fault localization via both unsupervised and supervised learning. Meanwhile, we have demonstrated ProFL's effectiveness under different settings and through a case study within Alipay, a popular online payment system with over 1 billion global users.

CCS CONCEPTS

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

KEYWORDS

Automated Program Repair, Fault Localization, Unified Debugging

ACM Reference Format:
Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach . In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Los Angeles/Virtual, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397351

1 INTRODUCTION

Software bugs (also called software faults, errors, defects, flaws, or failures [74]) are prevalent in modern software systems, and have been widely recognized as notoriously costly and disastrous. For example, in 2017, `Tricentis.com` investigated software failures impacting 3.7 Billion users and $1.7 Trillion assets, and reported that this is just scratching the surface – *there can be far more software bugs in the world than we will likely ever know about* [70]. In practice, software debugging is widely adopted for removing software bugs. However, manual debugging can be extremely tedious, challenging, and time-consuming due to the increasing complexity of modern software systems [68]. Therefore, a large body of research efforts have been dedicated to automated debugging [8, 34, 52, 59, 68].

There are two key questions in software debugging: (1) *how to automatically localize software bugs to facilitate manual repair?* (2) *how to automatically repair software bugs without human intervention?* To address them, researchers have proposed two categories of techniques, *fault localization* [5, 14, 30, 42, 51, 80, 81] and *program repair* [32, 36, 38, 44, 45, 60, 61, 72]. For example, pioneering

*This work was mainly done when Yiling Lou was a visiting student in UT Dallas. HCST is short for Key Lab of High Confidence Software Technologies, MoE, China.
†Corresponding authors.

ISSTA '20, July 18–22, 2020, Los Angeles/Virtual, CA, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397351

spectrum-based fault localization (SBFL) techniques [5, 14, 30] compute the code elements covered by more failed tests or less passed tests as more suspicious, and pioneering mutation-based fault localization (MBFL) techniques [51, 55, 81] inject code changes (e.g., changing > into >=) based on mutation testing [17, 26] to each code element to check its impact on test outcomes; meanwhile, pioneering search-based program repair techniques (e.g., GenProg [38]) tentatively change program elements based on certain rules (e.g., deleting/changing/adding program elements) and use the original test suite as the oracle to validate the generated patches. Please refer to the recent surveys on automated software debugging for more details [50, 76]. To date, unfortunately, although debugging has been extensively studied and even has drawn attention from industry (e.g., FaceBook [47, 66] and Fujitsu [65]), *we still lack practical automated debugging techniques*: (1) existing fault localization techniques have been shown to have limited effectiveness in practice [56, 77]; (2) existing program repair techniques can only fix a small ratio of real bugs [20, 29, 73] or specific types of bugs [47].

In this work, we aim to revisit the connection between program repair and fault localization for more powerful debugging. We observe that the current existing connection between fault localization and program repair is that program repair techniques usually use off-the-shelf fault localization techniques to identify potential buggy locations for patching, e.g., the Ochiai [5] SBFL technique is leveraged in many recent program repair techniques [20, 29, 73]. In this work, we propose a new *unified debugging* approach to unify program repair and fault localization in the reversed way, and explore the following question, *can program repair in turn help with fault localization?* Our basic insight is that the patch execution information during program repair can provide useful feedbacks and guidelines for powerful fault localization. For example, if a patch passes some originally failing test(s), the patched location is very likely to be closely related to the real buggy location (e.g., sharing the same method or even same line), since otherwise the patch cannot mute the bug impacts to pass the failing test(s). Based on this insight, we designed ProFL (**Pro**gram Repair for **F**ault **L**ocalization), a simplistic feedback-driven fault localization approach that leverages patch-execution information from state-of-the-art PraPR [20] repair tool for rearranging fault localization results computed by off-the-shelf fault localization techniques. Note that even state-of-the-art program repair techniques can only fix a small ratio of real bugs (i.e., <20% for Defects4J [20, 29, 73]) fully automatically and were simply aborted for the vast majority of unfixed bugs, while our approach extends the application scope of program repair to all possible bugs – *program repair techniques can also provide useful fault localization information to help with manual repair even for the bugs that are hard to fix automatically.*

We have evaluated our ProFL on the Defects4J (V1.2.0) benchmark, which includes 395 real-world bugs from six open-source Java projects and has been widely used for evaluating both fault localization and program repair techniques [20, 29, 40, 67, 73]. Our experimental results show that ProFL can localize 161 bugs within Top-1, while state-of-the-art spectrum and mutation based fault localization techniques can at most localize 117 bugs within Top-1. We further investigate the impacts of various experimental configurations: (1) we investigate the finer-grained patch categorizations and observe that they do not have clear impact on ProFL; (2) we investigate the impact of different off-the-shelf SBFL formulae used in ProFL, and observe that ProFL consistently outperforms traditional SBFL regardless of the used formulae; (3) we replace the repair feedback information with traditional mutation feedback information in ProFL (since they both record the impacts of certain changes to test outcomes), and observe that ProFL still localizes 141 bugs within Top-1, significantly outperforming state-of-the-art SBFL and MBFL; (4) we feed ProFL with only *partial* patch-execution information (since the test execution will be aborted for a patch as soon as it gets falsified by some test for the sake of efficiency in practical program repair scenario), and observe that, surprisingly, ProFL using such partial information can reduce the execution overhead by 26.17X with no clear effectiveness drop; (5) we also apply ProFL on a newer version of Defects4J, Defects4J (V1.4.0) [22], and observe that ProFL performs consistently. In addition, we further observe that ProFL can even significantly boost state-of-the-art fault localization via both unsupervised [82, 83] and supervised [39] learning, localizing 185 and 216.80 bugs within Top-1, the best fault localization results via unsupervised/supervised learning to our knowledge.

This paper makes the following contributions:

- This paper opens a new unified debugging dimension for improving fault localization via off-the-shelf state-of-the-art program repair techniques, and also extends the application scope of program repair techniques to all possible bugs (not only the bugs that can be directly automatically fixed).
- The proposed approach, ProFL, has been implemented as a fully automated Maven plugin[1] for automated feedback-driven fault localization based on the patch executions of PraPR, a state-of-the-art program repair technique.
- We have performed an extensive study of the proposed approach on the widely used Defects4J benchmarks, and demonstrated the effectiveness, efficiency, robustness, and general applicability of the proposed approach.
- ProFL plugin has been deployed in an international IT company with over 1 billion global users. We also conducted a real-world industry case study within the company.

2 BACKGROUND AND RELATED WORK

Fault Localization [5, 9, 14, 23–25, 28, 30, 42, 51, 57, 62–64, 80, 81] aims to precisely diagnose potential buggy locations to facilitate manual bug fixing. The most widely studied *spectrum-based fault localization* (SBFL) techniques usually apply statistical analysis (e.g., Tarantula [30], Ochiai [5], and Ample [14]) or learning techniques [9, 62–64] to the execution traces of both passed and failed tests to identify the most suspicious code elements (e.g., statements/methods). The insight is that code elements primarily executed by failed tests are more suspicious than the elements primarily executed by passed tests. However, a code element executed by a failed test does not necessarily indicate that the element has impact on the test execution and has caused the test failure. To bridge the gap between coverage and impact information, researchers proposed *mutation-based fault localization* (MBFL) [51, 54, 55, 81], which injects changes to each code element (based on mutation testing [17, 26]) to check its impact on the test outcomes. MBFL has

[1] https://github.com/yilinglou/proFL

been applied to both general bugs (pioneered by Metallaxis [54, 55]) and regression bugs (pioneered by FIFL [81]). ProFL shares similar insight with MBFL in that program changes can help determine the impact of code elements on test failures. However, ProFL utilizes program repair information that aims to fix software bugs to *pass* more tests rather than mutation testing that was originally proposed to create new artificial bugs to *fail* more tests; ProFL also embodies a new feedback-driven fault localization approach. Besides SBFL and MBFL, researchers have proposed to utilize various other information for fault localization (such as program slicing [80], predicate switching [84], code complexity [67], program invariant [7] information, and bug reports [37]), and have also utilized supervised learning to incorporate such different feature dimensions for fault localization [39, 40, 79]. However, the effectiveness of supervised-learning-based fault localization techniques may largely depend on the training sets, which may not always be available. In this work, we aim to explore a new direction for simplistic fault localization without supervised learning, i.e., leveraging patch-execution information (from program repair) for powerful fault localization.

Automated Program Repair (APR) techniques [12, 15, 19, 21, 27, 44–46, 48–50, 53, 58, 71, 78] aim to directly fix software bugs with minimal human intervention via synthesizing *genuine* patches (i.e., the patches semantically equivalent to developer patches). Therefore, despite a young research area, APR has been extensively studied in the literature. Various techniques have been proposed to directly modify program code representations based on different rules/strategies, and then use tests as the oracle to validate each generated candidate patch to find *plausible* patches (i.e., the patches passing all tests/checks) [12, 15, 21, 45, 49, 53, 58, 78]. Note that not all plausible patches are genuine patches; thus existing APR techniques all rely on manual inspection to find the final genuine patches among all plausible ones. *Search-based* APR techniques assume that most bugs could be solved by searching through all the potential candidate patches based on certain patching rules (i.e., program-fixing templates) [16, 29, 38, 73]. Alternatively, *semantics-based* techniques use deeper semantical analyses (including symbolic execution [13, 33]) to synthesize program conditions, or even more complex code snippets, that can pass all the tests [49, 53, 78]. Recently, search-based APR has been extensively studied due to its scalability on real-world systems, e.g., the most recent PraPR technique has been reported to produce genuine patches for 43 real bugs from Defects4J [31]. Despite the success of recent advanced APR techniques, even the most recent program repair technique can only fix a small ratio (i.e., <20% for Defects4J) of real bugs [20, 29, 73] or specific types of bugs [47].

In this work, we aim to leverage program repair results to help with fault localization. More specifically, we design, ProFL, a simplistic feedback-driven fault localization approach guided by patch-execution results (from program repair). Note that Ghanbari et al. [20] and Timperley et al. [69] have considered that plausible patches may potentially help localize bugs. However, Ghanbari et al. did not present a systematic fault localization approach working for all possible bugs, while Timperley et al. showed that this direction is ineffective. In contrast, we present the first systematic fault localization approach driven by state-of-the-art program repair, perform the first extensive study under various settings and on a large number of real-world bugs, and *show for the first time that state-of-the-art program repair can substantially boost state-of-the-art fault localization.* Feedback-driven fault localization techniques have also been investigated before [41, 43]. However, existing feedback-driven fault localization techniques usually require manual inspection to guide debugging. In contrast, we present a fully automated feedback-driven fault localization, i.e., ProFL utilizes patch-execution results as feedback to enable powerful automated fault localization.

```
Class: org.apache.commons.math.analysis.solvers.BracketingNthOrderBrentSolver
Method:protected double doSolve()
Developer patch:
233: if (agingA >= MAXIMAL_AGING) {
234: // ...
235: -  targetY = -REDUCTION_FACTOR * yB;
236: +  final int p = agingA - MAXIMAL_AGING;
237: +  final double weightA = (1 << p) - 1;
238: +  final double weightB = p + 1;
239: +  targetY = (weightA * yA - weightB * REDUCTION_FACTOR * yB)
       / (weightA + weightB);
240:    } else if (agingB >= MAXIMAL_AGING) {
241: -   targetY = -REDUCTION_FACTOR * yA;
243: // ...
243: +   final int p = agingB - MAXIMAL_AGING;
244: +   final double weightA = p + 1;
245: +   final double weightB = (1 << p) - 1;
246: +   targetY = (weightB * yB - weightA * REDUCTION_FACTOR * yA)
       / (weightA + weightB);
```

```
Patch P4, generated by PraPR:
260: -   if (signChangeIndex - start >= end - signChangeIndex) {
260: +   if (MAXMAL_AGING - start >= end - signChangeIndex) {
261:        ++start;
262:     } else {
263:        --end;
264:     }
```

```
Patch P5, generated by PraPR:
317: -   x[signChangeIndex] = nextX;
317: +   x[agingA] = nextX;
318:     System.arraycopy(y, signChangeIndex, y, signChangeIndex +
       1, nbPoints - signChangeIndex);
319:     y[signChangeIndex] = nextY;
```

Figure 1: Developer and generated patches for Math-40

```
Class: com.google.javascript.jscomp.NodeUtil
Method:static boolean functionCallHasSideEffects
Developer patch:
958: +   if (nameNode.getFirstChild().getType() == Token.NAME) {
959: +   String namespaceName = nameNode.getFirstChild().getString
       ();
960: +   if (namespaceName.equals("Math")) {
961: +       return false;
962: +       }
963: +   }
```

```
Patch P10, generated by PraPR:
933: -   if (callNode.isNoSideEffectsCall()) {
933: +   if (callNode.hasChildren()) {
934:        return false;
935:    }
```

Figure 2: Developer and generated patches for Closure-61

3 MOTIVATING EXAMPLES

In this section, we present two real-world bug examples to show the potential benefits of program repair for fault localization.

3.1 Example 1: Math-40

We use Math-40 from Defects4J (V1.2.0) [31], a widely used collection of real-world Java bugs, as our first example. Math-40 denotes the 40th buggy version of Apache Commons Math project [6] from

Table 1: Five top-ranked methods from Math-40

EID	Method Signature	SBFL	PID	#F (1)	#P (3177)
e_1	`incrementEvaluationCount()`	0.57	$\mathcal{P}_1$	1	3170
e_2	`BracketingNthOrderBrentSolver(Number)`	0.33	$\mathcal{P}_2$	1	3172
e_3	`BracketingNthOrderBrentSolver(double, ...)`	0.28	$\mathcal{P}_3$	1	3177
e_4^*	`doSolve()`	0.27	$\mathcal{P}_4$ $\mathcal{P}_5$	0 0	3177 3169
e_5	`guessX(double[], ...)`	0.20	$\mathcal{P}_6$	0	3176

Defects4J (V1.2.0). The bug is located in a single method of the project (method `doSolve` of class `BracketingNthOrderBrentSolver`).

We attempted to improve the effectiveness of traditional SBFL based on Ochiai formula [5], which has been widely recognized as one of the most effective SBFL formulae [40, 57, 82]. Inspired by prior work [67], we used the *aggregation* strategy to aggregate the maximum suspiciousness values from statements to methods. Even with this improvement in place, Ochiai still cannot rank the buggy method in the top, and instead ranks the buggy method in the 4th place (with a suspiciousness value of 0.27). The reason is that traditional SBFL captures only coverage information and does not consider the actual impacts of code elements on test behaviors.

In an attempt to fix the bug, we further applied state-of-the-art APR technique, PraPR [20], on the bug. However, since fixing the bug requires multiple lines of asymmetric edits, the genuine patch is beyond the reach of PraPR and virtually other existing APR techniques as well. Analyzing the generated patches and their execution results, however, gives some insights on the positive effects that an APR technique might have on fault localization.

Among a large number of methods in Math-40, Table 1 lists the Top-5 most suspicious methods based on Ochiai. Each row corresponds to a method, with the highlighted one corresponding to the actual buggy method (i.e., `doSolve`). Column "EID" assigns an identifier for each method. Column "SBFL" reports SBFL suspiciousness values for each method, and "PID" assigns an identifier for each patch generated by PraPR that targets the method. Column "#F"/"#P" reports the number of originally failing/passing tests that still fail/pass on each generated patch. The numbers within the parentheses in the table head are the number of failing/passing tests on the original buggy program. We also present the details of the developer patch for the bug and two patches generated by PraPR on the buggy method in Figure 1. From the table, we observe that $\mathcal{P}_4$ is a plausible patch, meaning that it passes all of the available tests but it might be not a genuine fix; $\mathcal{P}_5$ passes originally failing tests, while fails to pass 8 originally passing tests.

Several observations can be made at this point: First, whether the originally failing tests pass or not on a patch, can help distinguish the buggy methods from some correct methods. For example, for e_1, e_2 and e_3, the originally failing test remains failing on all of their patches, while for the buggy method e_4, the originally failing test becomes passing on both its patches. Second, whether the originally passing tests fail or not, can also help separate the buggy methods from some correct methods, e.g., $\mathcal{P}_4$ for the buggy method e_4 does not fail any originally passing tests while the patch for the correct method e_5 still fails some originally passing tests. Lastly, the detailed number of tests affected by the patches may not matter much. For example, for the correct method e_5, its patch only fails one originally passing test, but for the buggy method e_4, patch $\mathcal{P}_5$ makes even more (i.e., 8) originally passing tests fail.

Table 2: Five top-ranked methods from Closure-61

EID	Method Signature	SBFL	PID	#F (3)	#P (7082)
e_1	`toString()`	0.34	$\mathcal{P}_7$	3	7079
e_2	`getSortedPropTypes()`	0.33	$\mathcal{P}_8$	3	6981
e_3	`toString(StringBuilder, ...)`	0.27	$\mathcal{P}_9$	3	7042
e_4^*	`functionCallHasSideEffects(Node, ...)`	0.18	$\mathcal{P}_{10}$	1	6681
e_5	`nodeTypeMayHaveSideEffects(Node, ...)`	0.09	$\mathcal{P}_{11}$	1	6766

3.2 Example 2: Closure-61

We further looked into Closure-61, another real-world buggy project from Defects4J (V1.2.0), but for which PraPR is even *unable* to generate any plausible patch. Similar with the first example, we present the Ochiai and PraPR results for the Top-5 methods in Table 2.

Based on Table 2, we observe that even the non-plausible noisy patch $\mathcal{P}_{10}$ is related to the buggy methods. The patches targeting method `getSortedPropTypes` and the two overloading methods of `toString` (which have higher suspiciousness values than that of the buggy method `functionCallHasSideEffects`) cannot generate any patch that can pass any of the originally failing tests. In addition, the fact that the number of passed tests which now fail in the patches of the buggy method are much larger than that for the correct method `nodeTypeMayHaveSideEffects` further confirms our observation above that, the detailed impacted test number does not matter much with the judgement of the correctness of a method.

Based on the above two examples, we have following implications to utilize the patch execution results to improve the original SBFL: (1) the patches (no matter plausible or not) positively impacting some failed test(s) may indicate the actual buggy locations and should be favored; (2) the patches negatively impacting some passed test(s) may help exclude some correct code locations and should be unfavored; (3) the detailed number of the impacted tests does not matter much for fault localization. Therefore, we categorize all the patches into four different basic groups based on whether they impact originally passed/failed tests to help with fault localization, details shown in Section 4.

4 APPROACH

4.1 Preliminaries

In order to help the readers better understand the terms used throughout this paper, in what follows, we attempt to define a number of key notions more precisely.

Definition 4.1 (Candidate Patch). Given the original program $\mathcal{P}_o$, a candidate patch $\mathcal{P}$ can be created by modifying one or more program elements within $\mathcal{P}_o$. The set of all candidate patches generated for the program is denoted by $\mathbb{P}$.

In this paper, we focus on APR that conducts only *first-order* program transformations, which only change one program element in each patch, such as PraPR [20]. Note that our approach is general and can also be applied to other APR techniques in theory, even including the ones applying *high-order* program transformations.

Definition 4.2 (Patch Execution Matrix). Given a program $\mathcal{P}_o$, its test suite $\mathcal{T}$, and its corresponding set of all candidate patches $\mathbb{P}$, the patch execution matrix, $\mathbb{M}$, is defined as the execution results of all patches in $\mathbb{P}$ on all tests in $\mathcal{T}$. Each matrix cell result, $\mathbb{M}[\mathcal{P}, t]$, represents the execution results of test $t \in \mathcal{T}$ on patch $\mathcal{P} \in \mathbb{P}$, and

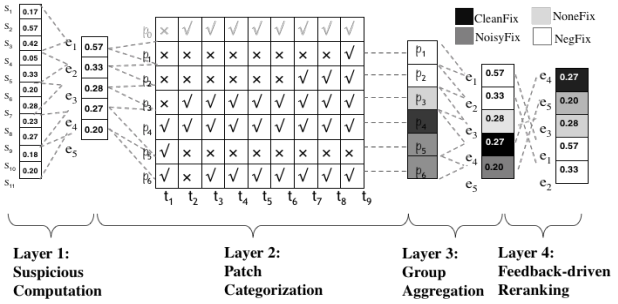


Figure 3: Overview of ProFL

can have the following possible values, {✓, ✗, ❍}, which represent *failed*, *passed*, and *unknown yet*.

Note that for the ease of presentation, we also include the original program execution results in $\mathbb{M}$, i.e., $\mathbb{M}[\mathcal{P}_o, t]$ denotes the execution results of test t on the original program $\mathcal{P}_o$.

Based on the above definitions, we can now categorize candidate patches based on the insights obtained from motivating examples:

Definition 4.3 (Clean-Fix Patch). A patch $\mathcal{P}$ is called a Clean-Fix Patch, i.e., $\mathbb{G}[\mathcal{P}]$ = `CleanFix`, if it passes some originally failing tests while does not fail any originally passing tests, i.e., $\exists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = ✗ \wedge \mathbb{M}[\mathcal{P}, t] = ✓$, and $\nexists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = ✓ \wedge \mathbb{M}[\mathcal{P}, t] = ✗$.

Note that $\mathbb{G}[\mathcal{P}]$ returns the category group for each patch $\mathcal{P}$.

Definition 4.4 (Noisy-Fix Patch). A patch $\mathcal{P}$ is called a Noisy-Fix Patch, i.e., $\mathbb{G}[\mathcal{P}]$ = `NoisyFix`, if it passes some originally failing tests but also fails on some originally passing tests, i.e., $\exists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = ✗ \wedge \mathbb{M}[\mathcal{P}, t] = ✓$, and $\exists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = ✓ \wedge \mathbb{M}[\mathcal{P}, t] = ✗$.

Definition 4.5 (None-Fix Patch). A patch $\mathcal{P}$ is called a None-Fix Patch, i.e., $\mathbb{G}[\mathcal{P}]$ = `NoneFix`, if it does not impact any originally failing or passing tests. More precisely, $\nexists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = ✗ \wedge \mathbb{M}[\mathcal{P}, t] = ✓$, and $\nexists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = ✓ \wedge \mathbb{M}[\mathcal{P}, t] = ✗$.

Definition 4.6 (Negative-Fix Patch). A patch $\mathcal{P}$ is called a Negative-Fix Patch, i.e., $\mathbb{G}[\mathcal{P}]$ = `NegFix`, if it does not pass any originally failing test while fails some originally passing tests, i.e., $\nexists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = ✗ \wedge \mathbb{M}[\mathcal{P}, t] = ✓$, and $\exists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = ✓ \wedge \mathbb{M}[\mathcal{P}, t] = ✗$.

Based on our insights obtained from the motivating example, the ranking of different patch groups is: `CleanFix` > `NoisyFix` > `NoneFix` > `NegFix`. Note that in Section 4.3, we will discuss more patch categorization variants besides such default patch categorization to further study their impacts on ProFL.

4.2 Basic ProFL

The overview of ProFL is shown in Figure 3. According to the figure, ProFL consists of four different layers. The input for ProFL is the actual buggy program under test and the original failing test suite, and the final output is a refined ranking of the program elements based on the initial suspiciousness calculation. In the first layer, ProFL conducts naive SBFL formulae (e.g., Ochiai [5]) at the statement level, and then performs *suspiciousness aggregation* [67] to calculate the initial suspiciousness value for each program element. Note that besides such default initial suspiciousness computation, ProFL is generic and can leverage the suspiciousness values computed by any other advanced fault localization technique in this layer (such as the PageRank-based fault localization [82]). In the second layer, ProFL collects the patch execution matrix along the program repair process for the program under test, and categorizes each patch into different groups based on Section 4.1. In the third layer, for each element, ProFL maps the group information of its corresponding patches to itself via *group aggregation*. In the last layer, ProFL finally reranks all the program elements via considering their suspiciousness and group information in tandem.

We next explain each layer in detail with our first motivation example. Since the number of tests and patches are really huge, due to space limitation, we only include the tests and patches that are essential for the ranking results of the elements. After reduction, we consider the six patches ($\mathcal{P}_1$ to $\mathcal{P}_6$) and the 9 tests whose statuses changed on these patches (denoted as t_1 to t_9). Based on Definition 4.2, we present $\mathbb{M}$ in Figure 3, The first row stands for $\mathbb{M}[\mathcal{P}_o, \mathcal{T}]$, the execution results of $\mathcal{T}$ on the original buggy program $\mathcal{P}_o$, and from the second row, each row represents $\mathbb{M}[\mathcal{P}, \mathcal{T}]$, the execution results of each patch $\mathcal{P}$ as shown in Table 1 on $\mathcal{T}$.

4.2.1 Layer 1: Suspicious Computation. Given the original program statements, e.g., $\mathcal{S} = [s_1, s_2, ..., s_n]$, we directly apply an off-the-shelf spectrum-based fault localization technique (e.g., the default Ochiai [5]) to compute the suspiciousness for each statement, e.g., $\mathbb{S}[s_j]$ for statement s_j. Then, the proposed approach applies *suspiciousness aggregation* [67] to compute the element suspiciousness values at the desired level (e.g., method level in this work) since prior work has shown that suspicious aggregation can significantly improve fault localization results [11, 67]. Given the initial list of $\mathcal{E} = [e_1, e_2, ..., e_m]$, for each $e_i \in \mathcal{E}$, suspiciousness aggregation computes its suspiciousness as $\mathbb{S}[e_i] = Max_{s_j \in e_i} \mathbb{S}[s_j]$, i.e., the highest suspiciousness value for all statements within a program element is computed as the suspiciousness value for the element.

For our first motivation example, after suspicious aggregation, for the five elements, $\mathbb{S}[e_1, e_2, e_3, e_4, e_5] = [0.57, 0.33, 0.28, 0.27, 0.20]$.

4.2.2 Layer 2: Patch Categorization. In this layer, ProFL automatically invokes off-the-shelf program repair engines (PraPR [20] for this work) to try various patching opportunities and record the detailed patch-execution matrix, $\mathbb{M}$. Then, based on the resulting $\mathbb{M}$, ProFL automatically categorizes each patch into different groups. Given program element e and all the patches generated for the program, $\mathbb{P}$, the patches occurring on e can be denoted as $\mathbb{P}[e]$. Then, based on Definitions 4.3 to 4.6, each patch within $\mathbb{P}[e]$ for each element e can be categorized into one of the four following groups, {`CleanFix`, `NoisyFix`, `NoneFix`, `NegFix`}. Recall that $\mathbb{G}[\mathcal{P}]$ represents the group information for $\mathcal{P}$, e.g., $\mathbb{G}[\mathcal{P}]$ = `CleanFix` denotes that $\mathcal{P}$ is a clean-fix patch.

For the example, the group of each patch in the motivation example is as follows: $\mathbb{G}[\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4, \mathcal{P}_5, \mathcal{P}_6]$ = [`NegFix`, `NegFix`, `NoneFix`, `CleanFix`, `NoisyFix`, `NoisyFix`]

4.2.3 Layer 3: Group Aggregation. For each program element e, we utilize its corresponding patch group information to determine its own group information. Recall that the ranking of different patch groups is: `CleanFix`>`NoisyFix`>`NoneFix`>`NegFix`. Then, the group information for a program element can be determined by the best group information of all patches occurring on the program

element. Therefore, we present the following rules for determining the group information for each e:

$$\mathbb{G}[e] = \begin{cases} \texttt{CleanFix} & \text{if } \exists \mathcal{P},\ \mathcal{P} \in \mathbb{P}[e] \wedge \mathbb{G}[\mathcal{P}] = \texttt{CleanFix} \\ \texttt{NoisyFix} & \text{else if } \exists \mathcal{P},\ \mathcal{P} \in \mathbb{P}[e] \wedge \mathbb{G}[\mathcal{P}] = \texttt{NoisyFix} \\ \texttt{NoneFix} & \text{else if } \exists \mathcal{P},\ \mathcal{P} \in \mathbb{P}[e] \wedge \mathbb{G}[\mathcal{P}] = \texttt{NoneFix} \\ \texttt{NegFix} & \text{else if } \exists \mathcal{P},\ \mathcal{P} \in \mathbb{P}[e] \wedge \mathbb{G}[\mathcal{P}] = \texttt{NegFix} \end{cases} \tag{1}$$

Shown in Equation 1, element e is within Group `CleanFix` whenever there is any patch $\mathcal{P}$ within e such that $\mathcal{P}$ is a clean-fix patch; otherwise, it is within Group `NoisyFix` whenever there is any patch $\mathcal{P}$ within e such that $\mathcal{P}$ is a noisy-fix patch.

After group aggregation, the group of each program element (i.e., method) in the motivation example is $\mathbb{G}[e_1, e_2, e_3, e_4, e_5]$ = [`NegFix`, `NegFix`, `NoneFix`, `CleanFix`, `NoisyFix`].

4.2.4 Layer 4: Feedback-driven Reranking. In this last layer, we compute the final ranked list of elements based on the aggregated suspiciousness values and groups. All the program elements will be first clustered into different groups with Group `CleanFix` ranked first and Group `NegFix` ranked last. Then, within each group, the initial SBFL (or other fault localization techniques) suspiciousness values will be used to rank the program elements. Assume we use $\mathbb{R}[e_1, e_2]$ to denote the total-order ranking between any two program elements, it can be formally defined as:

$$\mathbb{R}[e_1, e_2] = \begin{cases} e_1 \geq e_2 & \text{if } \mathbb{G}[e_1] > \mathbb{G}[e_2] \text{ or} \\ & \mathbb{G}[e_1] = \mathbb{G}[e_2] \wedge \mathbb{S}[e_1] \geq \mathbb{S}[e_2] \\ e_2 \geq e_1 & \text{if } \mathbb{G}[e_2] > \mathbb{G}[e_1] \text{ or} \\ & \mathbb{G}[e_1] = \mathbb{G}[e_2] \wedge \mathbb{S}[e_2] \geq \mathbb{S}[e_2 1] \end{cases} \tag{2}$$

That is, e_1 is ranked higher or equivalent to e_2 only when (i) e_1 is within a higher-ranked group, or (ii) e_1 is within the same group as e_2 but has a higher or equivalent suspicious value compared to e_2. Therefore, the final ranking of our example is: $e_4 \geq e_5 \geq e_3 \geq e_1 \geq e_2$, ranking the buggy method e_4 at the first place.

4.3 Variants of ProFL

Taking the approach above as the basic version of ProFL, there can be many variants of ProFL, which are discussed as follows.

Finer-grained Patch Categorization. Previous work [20] found that plausible patches are often coupled tightly with buggy elements, which actually is a subset of `CleanFix` defined in our work. Inspired by this finding, we further extend ProFL with finer-grained patch categorization rules, which respectively divide `CleanFix` and `NoisyFix` into two finer categories according to the criterion whether all failed tests are impacted. We use Figure 4 to show the relation between the four finer-grained patch categories and the four basic categories. Considering the finer categories, we further extend the group aggregation strategies in the third layer of ProFL accordingly as shown in Table 3 to study the impact of further splitting `CleanFix` and `NoisyFix` categories. For example, R_1 (ranking CleanPartFix below CleanAllFix) and R_2 (ranking CleanAllFix below CleanPartFix) study the two different ways for splitting `CleanFix`.

SBFL Formulae. The elements are reranked in the last layer based on their aggregated suspiciousness values and groups. In theory, ProFL is not specific for any particular way to calculate the aggregated suspiciousness value. Therefore, besides our default Ochiai [5] formula, all the other formulae in SBFL can be adopted in ProFL. We study all the 34 SBFL formulae considered in prior work [40, 67]. The impact of these formulae on ProFL would be studied later.

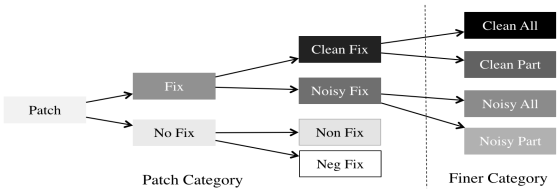


Figure 4: Patch categorization tree

Table 3: Finer-grained patch categorization rules

ID	Extended Categorization Aggregation Rules
R_1	`CleanAllFix>CleanPartFix>NoisyFix>NoneFix>NegFix`
R_2	`CleanPartFix>CleanAllFix>NoisyFix>NoneFix>NegFix`
R_3	`CleanFix>NoisyAllFix>NoisyPartFix>NoneFix>NegFix`
R_4	`CleanFix>NoisyPartFix> NoisyAllFix>NoneFix>NegFix`

Feedback Sources. Generally speaking, not only the patch execution results can be the feedback of our approach, any other execution results correlated with program modifications can serve as the feedback sources, e.g., mutation testing [26]. For example, a mutant and a patch are both modifications on the program, thus ProFL can directly be applied with the mutation information as feedback. However, mutation testing often includes simple syntax modifications that were originally proposed to simulate software bugs to *fail* more tests, while program repair often includes more (advanced) modifications that aim to *pass* more tests to fix software bugs. Therefore, although it is feasible to use mutation information as the feedback source of our approach, the effectiveness remains unknown, which would be studied.

Partial Execution Matrix. During program repair, usually the execution for a patch would terminate as soon as one test fails, which is the common practice to save the time cost. In this scenario, only partial execution results are accessible. In the previous sections, $\mathbb{M}$ is considered as complete (as traditional MBFL also requires full mutant execution matrices), which we denote as *full* matrix, $\mathbb{M}_f$, while in this section, we discuss the case where $\mathbb{M}$ is considered as incomplete in APR practice, which we call a *partial* matrix, $\mathbb{M}_p$. Recall Definition 4.2, different from $\mathbb{M}_f$, cells in $\mathbb{M}_p$ can be ❍ besides ✓ and ✗. E.g., when t is not executed on $\mathcal{P}$, $\mathbb{M}_p[\mathcal{P}, t]$ = ❍.

In the motivation example, during the patch execution, if $\mathcal{T}$ is executed in the order from t_1 to t_9, and one failed test would stop execution for each patch immediately, $\mathbb{M}_p$ is as follows.

$$\mathbb{M}_p = \begin{bmatrix} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 \\ \mathcal{P}_o & ✗ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\ \mathcal{P}_1 & ✗ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ \\ \mathcal{P}_2 & ✗ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ \\ \mathcal{P}_3 & ✗ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ \\ \mathcal{P}_4 & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\ \mathcal{P}_5 & ✓ & ✗ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ \\ \mathcal{P}_6 & ✓ & ✗ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ & ❍ \end{bmatrix} \tag{3}$$

In the scenario where only partial matrix is accessible, we can find there are many unknown results. Interestingly, in this example, we find the final ranking does not change at all even with a partial

Table 4: Benchmark statistics

ID	Name	#Bug	#Test	LoC
Lang	Apache commons-lang	65	2,245	22K
Math	Apache commons-math	106	3,602	85K
Time	Joda-Time	27	4,130	28K
Chart	JFreeChart	26	2,205	96K
Closure	Google Closure compiler	133	7,927	90K
Mockito	Mockito framework	38	1,366	23K
Defects4J (V1.2.0)		**395**	**21,475**	**344K**
Cli	Apcache commons-cli	24	409	4K
Codec	Apache commons-codec	22	883	10K
Csv	Apache commons-csv	12	319	2K
JXPath	Apache commons-jxpath	14	411	21K
Gson	Google GSON	16	N/A	12K
Guava	Google Guava	9	1,701,947	420K
Core	Jackson JSON processor	13	867	31K
Databind	Jackson data bindings	39	1,742	71K
Xml	Jackson XML extensions	5	177	6K
Jsoup	Jsoup HTML parser	63	681	14K
Defects4J (V1.4.0)		**587**	**26,964**	**503K**

matrix as input. For the patches $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{P}_5$ and $\mathcal{P}_6$, their patch categorization does not change at all. For example, since the failed tests are executed first, when $\mathcal{P}_5$ stops its execution, its execution result is that one failed test passes now and one passed test fails now, and thus $\mathcal{P}_5$ is still categorized into `NoisyFix`. For $\mathcal{P}_1$ and $\mathcal{P}_2$, although their patch categorization changes from `NegFix` to `NoneFix`, it does not impact the final ranking results. The example indicates the insensitivity of ProFL to partial matrix, and the categorization design is the main reason for it. We would further confirm this observation in the detailed experimental studies.

5 EXPERIMENT SET UP

5.1 Research Questions

In our study, we investigate the following research questions:

- **RQ1:** How does the basic ProFL perform compared with state-of-the-art SBFL and MBFL techniques?
- **RQ2:** How do different configurations impact ProFL?
  - **RQ2a:** What is the impact of finer patch categorization?
  - **RQ2b:** What is the impact of the used SBFL formula?
  - **RQ2c:** What is the impact of the feedback source used?
  - **RQ2d:** What is the impact of partial execution matrix?
  - **RQ2e:** What is the impact of the used benchmark suite?
- **RQ3:** Can ProFL further boost state-of-the-art unsupervised- and supervised-learning-based fault localization?

Besides the research questions, we also conduct a case study in an international IT company with over 700M Monthly Active Users.

5.2 Benchmark

We conduct our study on all bugs from the Defects4J benchmark [31], which has been widely used in prior fault-localization work [39, 40, 57, 67, 82]. Defects4J is a collection of reproducible real bugs with a supporting infrastructure. To our knowledge, all the fault localization studies evaluated on Defects4J use the original version Defects4J (V1.2.0). Recently, an extended version, Defects4J (V1.4.0), which includes more real-world bugs, has been released [22]. Therefore, we further perform the first fault localization study on Defects4J (V1.4.0) to reduce the threats to external validity.

In Table 4, Column "ID" presents the subject IDs. Columns "Name" and "#Bugs" present the full name and the number of bugs for each project. Columns "Loc" and "#Test" list the line-of-code information and the number of tests for the HEAD version of each project. Note that the two projects highlighted in gray are excluded due to build/test framework incompatibility with PraPR [20]. In total, our study is performed on all 395 bugs from Defects4J (V1.2.0) and 192 additional bugs from Defects4J (V1.4.0).

5.3 Independent Variables

Evaluated Techniques: We compare ProFL with the following state-of-the-art SBFL and MBFL techniques: **(a) Spectrum-based (SBFL)**: we consider traditional SBFL with suspiciousness aggregation strategy to aggregate suspiciousness values from statements to methods, which has been shown to be more effective than naive SBFL in previous work [11, 67]. **(b) Mutation-based (MBFL)**: we consider two representative MBFL techniques, MUSE [51] and Metallaxis [55]. **(c) Hybrid of SBFL and MBFL (MCBFL)**: we also consider the recent MCBFL [57], which represents state-of-the-art hybrid spectrum- and mutation-based fault localization. Furthermore, we include state-of-the-art learning-based fault localization techniques: **(a) Unsupervised**: we consider state-of-the-art PRFL [82] and PRFL$_{MA}$ [83] (which further improves PRFL via suspiciousness aggregation) that aim to boost SBFL with the unsupervised PageRank algorithm. **(b) Supervised**: we consider state-of-the-art supervised-learning-based fault localization, DeepFL [39], which outperforms all other learning-based fault localization [40, 67, 79]. Note that, SBFL and Metallaxis can adopt different SBFL formulae, and we by default uniformly use Ochiai [5] since it has been demonstrated to perform the best for both SBFL and MBFL [40, 57, 82].
Experimental Configurations: We explore the following configurations to study ProFL: **(a) Finer ProFL Categorization**: in RQ2a, we study the four extended categorization aggregation rules based on the finer patch categories as listed in Table 3. **(b) Studied SBFL Formulae**: in RQ2b, we implement all the 34 SBFL formulae considered in prior work [40, 67] to study the impact of initial formulae. **(c) Feedback Sources**: besides the patch execution results of program repair, mutation testing results can also be used as the feedback sources of ProFL. Thus, we study the impact of these two feedback sources in RQ2c. **(d) Partial Execution Matrix**: we collect partial execution matrices in three common test-execution orderings: **(i)** O_1: the default order in original test suite; **(ii)** O_2: running originally-failed tests first and then originally-passing tests, which is also the common practice in program repair to save the time cost; **(iii)** O_3: running originally-passing tests first and then originally-failed tests. The partial matrices collected by these orders are denoted as $\mathbb{M}_p^{(O_1)}$, $\mathbb{M}_p^{(O_2)}$ and $\mathbb{M}_p^{(O_3)}$ respectively. We investigate the impacts of different partial execution matrices used in RQ2d. **(e) Used Benchmarks**: we evaluate ProFL in two benchmarks, Defects4J (V1.2.0) and Defects4J (V1.4.0) in RQ2e.

5.4 Dependent Variables and Metrics

In this work, we perform fault localization at the method level following recent fault localization work [7, 39, 40, 67, 82], because the class level has been shown to be too coarse-grained while the statement level is too fine-grained to keep useful context information [35, 56]. We use the following widely used metrics [39, 40]:

Recall at Top-N: Top-N computes the number of bugs with at least one buggy element localized in the Top-N positions of the ranked list. As suggested by prior work [56], usually, programmers only inspect a few buggy elements in the top of the given ranked list, e.g., 73.58% developers only inspect Top-5 elements [35]. Therefore, following prior work [39, 40, 82, 85], we use Top-N (N=1, 3, 5).

Mean First Rank (MFR): For each subject, MFR computes the mean of the first relevant buggy element's rank for all its bugs, because the localization of the first buggy element for each bug can be quite crucial for localizing all buggy elements.

Mean Average Rank (MAR): We first compute the average ranking of all the buggy elements for each bug. Then, MAR of each subject is the mean of such average ranking of all its bugs. MAR emphasizes the precise ranking of all buggy elements, especially for the bugs with multiple buggy elements.

Fault localization techniques sometimes assign same suspiciousness score to code elements. Following prior work [39, 40], we use the *worst* ranking for the tied elements. For example, if a buggy element is tied with a correct element in the k^{th} position of the ranked list, the rank for both elements would be $k + 1^{th}$.

5.5 Implementation and Tool Supports

For APR, we use PraPR [20], a recent APR technique that fixes bugs at the bytecode level. We choose PraPR because it is one of the most recent APR techniques and has been demonstrated to be able to fix more bugs with a much lower overhead compared to other state-of-the-art techniques. PraPR is set to generate patches for all potentially buggy locations so that the misranked elements can be adjusted by ProFL. Note that, ProFL does not rely on any specific APR technique, since its feedback input (i.e., patch executions) is general and can come from any other APR technique in principle.

We now discuss the collection of all the other information for implementing ProFL and other compared techniques: (i) To collect the coverage information required by SBFL techniques, we use the ASM bytecode manipulation framework [10] to instrument the code on-the-fly via JavaAgent [2]. (ii) To collect the mutation testing information required by MBFL, we use state-of-the-art PIT mutation testing framework [3] (Version 1.3.2) with all its available mutators, following prior MBFL work [39, 40]. Note that we also modify PIT to force it to execute all tests for each mutant and collect detailed mutant impact information (i.e., whether each mutant can impact the detailed test failure message of each test [57]) required by Metallaxis. For PRFL, PRFL$_{MA}$, and DeepFL, we directly used the implementation released by the authors [39, 83]. All experiments are conducted on a Dell workstation with Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, 330GB RAM, and Ubuntu 18.04.1 LTS.

5.6 Threats to Validity

Threats to internal validity mainly lie in the correctness of implementation of our approach and the compared techniques. To reduce this threat, we manually reviewed our code and verified that the results of the overlapping fault localization techniques between this work and prior work [40, 82, 83] are consistent. We also directly used the original implementations from prior work [39, 83].

Threats to construct validity mainly lie in the rationality of assessment metrics that we chose. To reduce this threat, we chose the metrics that have been recommended by prior studies/surveys [35, 56] and widely used in previous work [39, 40, 67, 82].

Threats to external validity mainly lie in the benchmark suites used in our experiments. To reduce this threat, we chose the widely used Defects4J (V1.2.0) benchmark, which includes hundreds of real bugs collected during real-world software development. To further reduce the threats, compared to previous work, we also conducted the first fault localization evaluation on an extended version of Defects4J, Defects4J (V1.4.0).

Table 5: Overall fault localization results

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
SBFL	117	219	259	19.15	24.63
MUSE	89	152	182	47.51	52.81
Metallaxis	84	175	223	17.10	19.54
MCBFL	131	227	267	17.99	23.26
ProFL	161	255	286	9.48	14.37

6 RESULTS

6.1 RQ1: Effectiveness of ProFL

To answer this RQ, we first present the overall fault localization results of ProFL and state-of-the-art SBFL and MBFL techniques on Defects4J (V1.2.0) in Table 5. Column "Tech Name" represents the corresponding techniques and the other columns present the results in terms of Top-1, Top-3, Top-5, MFR and MAR. From the table, we observe that ProFL significantly outperforms all the existing techniques in terms of all the five metrics. For example, the Top-1 value of ProFL is 161, 30 more than MCBFL, 44 more than aggregation-based SBFL, 77 more than Metallaxis, and 72 more than MUSE. In addition, MAR and MFR values are also significantly improved (e.g., 47.30% improvement in MFR compared with state-of-the-art MCBFL), indicating a consistent improvement for all buggy elements in the ranked lists. Note that our overall bug ranking results are consistent with prior fault localization work at the method level [40], e.g., state-of-the-art MBFL can outperform SBFL, demonstrating the effectiveness of MBFL. Meanwhile, we observe that SBFL outperforms state-of-the-art MBFL techniques in terms of Top-ranked bugs, which is not consistent with prior work [40]. We find the main reason to be that the prior work did not use suspicious aggregation (which was proposed in parallel with the prior work) for SBFL, demonstrating the effectiveness of suspiciousness aggregation for SBFL.

To further investigate why the simple ProFL approach works, we further analyze each of the four basic ProFL patch categories in a post-hoc way. For each patch category group $\mathcal{G}_i$, for each bug in the benchmark, we use metric $Ratio_b$ to represent the ratio of the number of buggy elements (i.e., methods in this work) categorized into group $\mathcal{G}_i$ to the number of all elements categorized into group

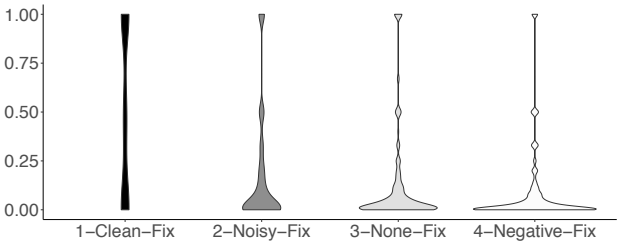


Figure 5: $Ratio_b$ distribution for different patch groups

$\mathcal{G}_i$. Formally, it can be presented as:

$$Ratio_b(\mathcal{G}_i) = \frac{|\{e|\mathbb{G}[\mathcal{P}] = \mathcal{G}_i \wedge \mathcal{P} \in \mathbb{P}[e]\} \wedge e \in \mathbb{B}|}{|\{e|\mathbb{G}[\mathcal{P}] = \mathcal{G}_i \wedge \mathcal{P} \in \mathbb{P}[e]\}|} \quad (4)$$

where $\mathbb{B}$ represents a set of buggy elements. $Ratio_b$ ranges from 0 to 1, and a higher value indicates a higher probability for a patch group to contain the actual buggy element(s). We present the distribution of the $Ratio_b$ values on all bugs for each of the four different patch groups in the violin plot in Figure 5, where the x axis presents the four different groups, the y axis presents the actual $Ratio_b$ values, and the width of each plot shows the distribution's density. From the figure we observe that the four different groups have totally different $Ratio_b$ distributions. E.g., group `CleanFix` has rather even distribution, indicating that roughly half of the code elements within this group could be buggy; on the contrary, group `NegFix` mostly have small $Ratio_b$ values, indicating that elements within this group are mostly not buggy. Such group analysis further confirms our hypothesis that different patch categories can be leveraged as the feedback information for powerful fault localization.

Finding 1: Simplistic feedback information from program repair can significantly boost existing SBFL-based fault localization techniques, opening a new dimension for fault localization via program repair.

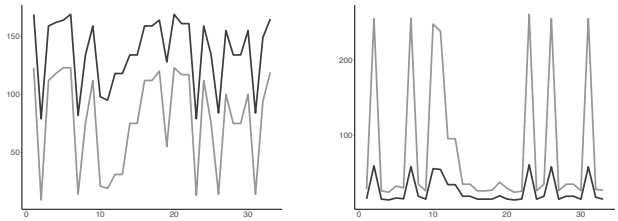
Table 6: Impacts of finer patch categorization

Tech	Top-1	Top-3	Top-5	MFR	MAR
ProFL	161	255	286	9.48	14.37
ProFL_{R_1}	162	255	286	9.53 (p=0.974)	14.41 (p=0.933)
ProFL_{R_2}	161	252	283	9.56 (p=0.904)	14.45 (p=0.876)
ProFL_{R_3}	161	255	285	9.67 (p=0.987)	14.62 (p=0.899)
ProFL_{R_4}	162	251	285	9.55 (p=0.949)	14.45 (p=0.967)

6.2 RQ2: Different experimental configurations

6.2.1 RQ2a: Impact of finer categorization. To investigate the four extended rules on the finer categorization presented in Section 4.3, we implemented different ProFL variants based on each rule in Table 3. The experimental results for all the variants are shown in Table 6. In the table, Column "Tech" presents each of the compared variants and the remaining columns present the corresponding metric values computed for each variant. Note that the four variants of ProFL implemented with different rules shown in Table 3 are denoted as ProFL_{R_1}, ProFL_{R_2}, ProFL_{R_3} and ProFL_{R_4}, respectively. From the table, we observe that ProFL variants with different extended rules perform similarly with the default setting in all the used metrics. To confirm our observation, we further perform the Wilcoxon signed-rank test [75] (at the significance level of 0.05) to compare each variant against the default setting in terms of both the first and average buggy-method ranking for each bug. The test results are presented in the parentheses in the MFR and MAR columns, and show that there is no significant difference (p»0.05) among the compared variants, indicating that considering the finer-grained grouping does not help much in practice. To explain this observation, we analyze the methods in `CleanAllFix`, `CleanPartFix` and find that they have same method sets for over 90% cases, indicating that fixing a subset of failing tests without breaking any passing tests is already challenging.

Finding 2: Finer-grained patch grouping has no significant impact on ProFL, further demonstrating the effectiveness of the default grouping.



(a) Top-1 (b) MAR

Figure 6: Comparison of ProFL and SBFL over all formulae

6.2.2 RQ2b: Impact of SBFL formulae. Our ProFL approach is general and can be applied to any SBFL formula, therefore, in this RQ, we further study the impact of different SBFL formulae on ProFL effectiveness. The experimental results are shown in Figure 6. In this figure, the x axis presents all the 34 SBFL formulae considered in this work, the y axis presents the actual metric values in terms of Top-1 and MAR, while the light and dark lines represent the original SBFL techniques and our ProFL version respectively. We can observe that, for all the studied SBFL formulae, ProFL can consistently improve their effectiveness. For example, the Top-1 improvements range from 41 (for ER1a) to 87 (for GP13), while the MAR improvements range from 36.54% (for Wong) to 77.41% (for GP02). Other metrics follow similar trend, e.g., the improvements in MFR are even larger than MAR, ranging from 49.24% (for SBI) to 80.47% (for GP02). Furthermore, besides the consistent improvement, we also observe that the overall performance of ProFL is quite stable for different SBFL formulae. For example, the MAR value for SBFL has huge variations when using different formulae, while ProFL has stable performance regardless of the formula used, indicating that ProFL can boost ineffective SBFL formulae even more.

Finding 3: ProFL can consistently improve all the 34 studied SBFL formulae, e.g., by 49.24% to 80.47% in MFR.

6.2.3 RQ2c: Impact of feedback source. Since ProFL is general and can even take traditional mutation testing information as feedback source, we implement a new ProFL variant that directly take mutation information (computed by PIT) as feedback. To distinguish

Table 7: Impacts of using mutation or repair information

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
MUSE$_{PIT}$	89	152	182	47.51	52.81
MUSE$_{PraPR}$	95	172	207	38.79	43.10
Metallaxis$_{PIT}$	84	175	223	17.10	19.54
Metallaxis$_{PraPR}$	77	170	211	21.42	22.94
MCBFL$_{PIT}$	131	227	267	17.99	23.26
MCBFL$_{PraPR}$	130	228	267	18.03	23.28
ProFL$_{PIT}$	141	238	266	15.24	20.33
ProFL$_{PraPR}$	161	255	286	9.48	14.37

the two ProFL variants, we denote the new variant as ProFL$_{PIT}$ and the default one as ProFL$_{PraPR}$. Meanwhile, all the existing MBFL techniques can also take the APR results from PraPR as input (PraPR can be treated as an augmented mutation testing tool with more and advanced mutators), thus we also implemented such variants for traditional MBFL for fair comparison, e.g., the original MUSE is denoted as MUSE$_{PIT}$ while the new MUSE variant is denoted as MUSE$_{PraPR}$. Table 7 presents the experimental results for both ProFL and prior mutation-based techniques using different information sources. We have the following observations:

First, ProFL is still the most effective technique compared with other techniques even with the feedback information from mutation testing. For example, ProFL with mutation information localizes 141 bugs within Top-1, while the most effective existing technique (no matter using mutation or repair information) only localizes 131 bugs within Top-1. This observation implies that the ProFL approach of using feedback information (from program-variant execution) to refine SBFL ranking is general in design, and is not coupled tightly with specific source(s) of feedback.

Second, ProFL performs worse when feedback source changes from program repair to mutation testing. For example, the Top-1 decreases from 161 to 141. The reason is that patches within groups `CleanFix/NoisyFix` can help promote the ranking of buggy methods. However, mutation testing cannot create many such patches. For example, we find that the number of bugs with `CleanFix/NoisyFix` patches increases by 39.84% when changing from mutation testing to APR. This further indicates that *APR is more suitable than mutation testing for fault localization since it aims to pass more tests while mutation testing was originally proposed to fail more tests.*

Third, for the two existing MBFL techniques, MUSE performs better in program repair compared to mutation testing while Metallaxis is the opposite. We find the reason to be that MUSE simply counts the number of tests changed from passed to failed and vice versa, while Metallaxis leverages the detailed test failure messages to determine mutant impacts. In this way, APR techniques that make more failed tests pass can clearly enhance the results of MUSE, but do not have clear benefits for Metallaxis.

Finding 4: ProFL still performs well even with the mutation feedback information, but has effectiveness decrements compared to using program repair, indicating the superiority of program repair over mutation testing for fault localization.

6.2.4 RQ2d: Impact of partial execution matrix. So far, we have studied ProFL using full patch execution matrices. However, in practical program repair, a patch will not be executed against the remaining tests as soon as some test falsifies it for the sake of efficiency. Therefore, we further study new ProFL variants with only partial patch execution matrices. The experimental results for three variants of ProFL using different partial matrices are shown in Table 8. From the table, we have the following observations:

Table 8: Impacts of using partial matrices

$\mathbb{M}_p$	Tech Name	Top-1	Top-3	Top-5	MFR	MAR
$\mathbb{M}_p^{(O_1)}$	MUSE$_{PraPR}$	92	148	172	118.56	125.11
	Metallaxis$_{PraPR}$	64	128	167	113.9	126.79
	ProFL	165	260	288	18.18	23.47
$\mathbb{M}_p^{(O_2)}$	MUSE$_{PraPR}$	87	130	152	191.71	206.0
	Metallaxis$_{PraPR}$	32	73	94	163.29	170.61
	ProFL	157	242	281	9.61	14.20
$\mathbb{M}_p^{(O_3)}$	MUSE$_{PraPR}$	89	128	144	169.33	174.09
	Metallaxis$_{PraPR}$	63	127	159	187.19	195.07
	ProFL	155	245	277	19.10	25.19

First, surprisingly, ProFL with different partial matrices still perform similarly with our default ProFL using full matrices, while the traditional MBFL techniques perform significantly worse using partial matrices. For example, the MFR and MAR values for existing MBFL all become over 160 when using partial matrices collected following common APR practice (i.e., $\mathbb{M}_p^{(O_2)}$), while the MFR and MAR values for ProFL have negligible change when using the same partial matrices. We think the reason to be that existing MBFL techniques utilize the detailed number of impacted tests for fault localization and may be too sensitive when switching to partial matrices. Second, ProFL shows consistent effectiveness with partial matrices obtained from different test execution orderings, e.g., even the worst ordering still produces 155 Top-1 bugs. One potential reason that $\mathbb{M}_p^{(O_3)}$ performs the worst is that if there is any passed tests changed into failing, the original failed tests will no longer be executed, missing the potential opportunities to have `CleanFix/NoisyFix` patches that can greatly boost fault localization. Luckily, in practice, repair tools always execute the failed tests first (i.e., $\mathbb{M}_p^{(O_2)}$), further demonstrating that ProFL is practical.

Note that the cost of ProFL consists of two parts: (1) APR time (full/partial matrix collection time), and (2) final fault-localization time based on the APR results. As for the latter cost, ProFL costs less than 2min to compute the suspiciousness scores for each Defects4J (V1.2.0) bug on average (including the APR result parsing time), which is negligible compared to the APR time. Therefore, we next present the cost reduction benefits that partial execution matrices can bring to speed up the ProFL APR time. The experimental results for the HEAD version (i.e., the latest and usually the largest version) of each studied subject are shown in Table 9. In the table, Column "Time$_f$" presents the time for executing all tests on each candidate patch while Column "Time$_p$" presents the time for terminating test execution on a patch as soon as the patch gets falsified (following the default test execution order of PraPR, i.e., $\mathbb{M}_p^{(O_2)}$, which executes originally failed tests first then passed tests). Similarly, Column "Execution$_f$"/"Execution$_p$" shows the number of test executions accumulated for all patches for full/partial matrices. From the table, we can observe that partial execution matrix collection can overall achieve 26.17X/718.85X speedup in terms

Table 9: Full and partial matrix collection (with 4 threads)

Subject	$Time_f$	$Time_p$	Speedup	$Execution_f$	$Execution_p$	Speedup
Lang-1	0m38s	0m31s	1.23X	2,282	157	14.54X
Closure-1	2,568m26s	110m33s	23.23X	186,451,071	253,378	735.86X
Mockito-1	452m33s	2m43s	166.58X	4,429,249	8,318	532.49X
Chart-1	32m27s	2m41s	12.09X	796,654	3,769	211.37X
Time-1	149m14s	0m41s	218.39X	677,094	1,147	590.32X
Math-1	68m24s	7m53s	8.63X	244,702	1,162	210.59X
Total	3,271m42s	125m2s	26.17X	192,601,052	267,931	718.85X

time/test-executions, e.g., even the largest Closure subject only needs less than 2 hours, indicating that ProFL can be scalable to real-world systems (since we have shown that ProFL does not have clear effectiveness drop when using only partial matrices).

Finding 5: ProFL keeps its high effectiveness even on partial patch execution matrices, especially with test execution ordering following the program repair practice, demonstrating that its runtime overhead can be reduced by 26.17X without clear effectiveness drop.

Table 10: Results on Defects4J (V1.4.0)

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
SBFL	59	102	124	13.81	20.44
MUSE	34	63	73	67.89	74.49
Metallaxis	47	88	115	21.45	28.30
MCBFL	67	112	132	13.20	19.79
ProFL	78	117	131	12.01	17.96

6.2.5 RQ2e: Impact of used benchmarks. In this RQ, we further compare ProFL and state-of-the-art SBFL/MBFL techniques on additional bugs from Defects4J (V1.4.0), to reduce the threats to external validity. The experimental results are shown in Table 10. From the table, we observe that ProFL still significantly outperforms all other compared techniques. E.g., Top-1 is improved from 59 to 78 compared to the original state-of-the-art SBFL. Such a consistent finding on additional bugs further confirms our findings in RQ1.

Finding 6: ProFL still significantly outperforms state-of-the-art SBFL and MBFL on additional bugs.

6.3 RQ3: Boosting learning-based localization

We further apply the basic ProFL to boost state-of-the-art unsupervised-learning-based (i.e., PRFL and $PRFL_{MA}$ [83]) and supervised-learning-based (i.e., DeepFL [39]) fault localization. For unsupervised-learning-based techniques, ProFL is generic and can use any existing fault localization techniques to compute initial suspiciousness (Section 4.2); therefore, we directly apply ProFL on the initial suspiciousness computed by PRFL and $PRFL_{MA}$, denoted as $ProFL_{PRFL}$ and $ProFL_{PRFLMA}$, respectively. For supervised-learning-based techniques, ProFL with all the 34 used SBFL formulae can serve as an additional feature dimension; therefore, we augment DeepFL by injecting ProFL features between the original mutation and spectrum feature dimensions (since they are all dynamic features), and denote that as $ProFL_{DeepFL}$. The experimental results are shown in Table 11. Note that DeepFL results are averaged over 10 runs due to the DNN randomness [39]. First, even the basic ProFL significantly outperforms state-of-the-art unsupervised-learning-based fault localization. E.g., ProFL localizes 161 bugs within Top-1, while the most effective unsupervised $PRFL_{MA}$ only localizes 136 bugs within Top-1. Second, ProFL can significantly boost unsupervised-learning-based fault localization. E.g., $ProFL_{PRFLMA}$ localizes 185 bugs within Top-1, *the best fault localization results on Defects4J without supervised learning to our knowledge*. Actually, such unsupervised-learning-based fault localization results even significantly outperform many state-of-the-art supervised-learning-based techniques, e.g., TraPT [40], FLUCCS [67], and CombineFL [85] only localize 156, 160, and 168 bugs from the same dataset within Top-1, respectively [39, 85]. Lastly, we can observe that ProFL even boosts state-of-the-art supervised-learning-based technique. E.g., it boosts DeepFL to localize 216.80 bugs within Top-1, *the best fault localization results on Defects4J with supervised learning to our knowledge*. The Wilcoxon signed-rank test [75] with Bonferroni corrections [18] for bug ranking also shows that ProFL significantly boosts all the studied learning-based techniques at significance level of 0.05.

Finding 7: ProFL significantly outperforms state-of-the-art unsupervised-learning-based fault localization, and can further boost unsupervised and supervised learning based fault localization, further demonstrating the effectiveness and general applicability of ProFL.

Table 11: Boosting state-of-the-art learning-based fault localization

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
PRFL	114	199	243	23.62	27.67
$ProFL_{PRFL}$	179	251	288	10.44	14.83
$PRFL_{MA}$	136	242	269	18.06	22.60
$ProFL_{PRFL_{MA}}$	185	264	295	9.04	13.73
DeepFL	211.00	284.50	310.50	4.97	6.27
$ProFL_{DeepFL}$	216.80	293.60	318.00	4.53	5.88

```
Developer patch:
Method:public static OrderDTO convertDTOFromAdOrder(AdPlan plan)
- result.setBizScene(SceneType.APP.toString());
+ result.setBizScene(plan.getSceneType().getSceneCode());
```

Figure 7: Developer patch for Bug-A in industry

6.4 Industry case study

ProFL has already been deployed in Alipay [1], a practical online payment system with over 1 billion global users. Before the deployment, the Alipay developers evaluated ProFL on a large number of real bugs and observed that ProFL consistently/largely boosts the Ochiai fault localization that Alipay used (e.g., localizing 2.1X more bugs within Top-1). We now present a case study with one real-world bug that ProFL has recently helped debug within Alipay. The bug is from `Project-A` (with 100M+ daily users, anonymized according to the company policy), a Spring-style [4] multi-module microservice system (built with Maven) with 197,402 LoC and 418 tests. State-of-the-art Ochiai and $PRFL_{MA}$ localize the buggy method as the 125th and 52nd, respectively. In contrast, within 48min, after

exploring 3459 patch executions, the default ProFL (applied on Ochiai with partial PraPR matrices) directly localizes the bug as the 1st, i.e., over 50X improvement in bug ranking. The developers looked into the code and found that although the bug is challenging to automatically fix with the current used repair system PraPR (as shown in Figure 7, this bug requires a multi-edit patch that is not currently supported by PraPR), multiple patches on the actual buggy method was able to mute the bug and make the originally failing tests pass. In this way, ProFL is able to easily point out the actual buggy location.

7 CONCLUSION

We have investigated a simple question: *can automated program repair help with fault localization?* To this end, we have designed, ProFL, the first *unified debugging* approach that leverages program repair information as the feedback for powerful fault localization. The experimental results on the widely used Defects4J benchmarks demonstrate that ProFL can significantly outperform state-of-the-art spectrum and mutation based fault localization. Furthermore, we have demonstrated ProFL's effectiveness under various settings as well as with an industry case study. Lastly, ProFL even boosts state-of-the-art fault localization via both unsupervised and supervised learning.

ACKNOWLEDGEMENTS

This work was partially supported by the National Key Research and Development Program of China under Grant No. 2017YFB1001803 and the National Natural Science Foundation of China under Grant Nos. 61872008 and 61861130363. This work was also partially supported by National Science Foundation under Grant Nos. CCF-1763906 and CCF-1942430, and Alibaba.

REFERENCES

[1] 2019. Alipay. https://intl.alipay.com/. Accessed Aug-22-2019.
[2] 2019. JavaAgent. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html
[3] 2019. Pitest. http://pitest.org
[4] 2019. Spring Framework. https://spring.io/. Accessed Jan-10-2020.
[5] 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 (TAICPART-MUTATION 2007)*. IEEE, 89–98.
[6] Apache. 2019. Commons Math. https://commons.apache.org/proper/commons-math/. Accessed Aug-22-2019.
[7] 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.
[8] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*. IEEE Computer Society, 85–103.
[9] Lionel C Briand, Yvan Labiche, and Xuetao Liu. 2007. Using machine learning to support debugging with tarantula. In *ISSRE*. 137–146.
[10] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).
[11] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler Bug Isolation via Effective Witness Test Program Generation. In *FSE*. 223–234.
[12] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based Program Repair Without the Contracts. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 637–647. http://dl.acm.org/citation.cfm?id=3155562.3155642
[13] Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *TSE* 3 (1976), 215–222.
[14] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight defect localization for java. In *ECOOP*. 528–550.
[15] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 550–554. https://doi.org/10.1109/ASE.2009.15
[16] V. Debroy and W. E. Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *2010 Third International Conference on Software Testing, Verification and Validation*. 65–74. https://doi.org/10.1109/ICST.2010.66
[17] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136
[18] Olive Jean Dunn. 1961. Multiple comparisons among means. *Journal of the American statistical association* 56, 293 (1961), 52–64.
[19] L. Gazzola, D. Micucci, and L. Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2755013
[20] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. 19–30. https://doi.org/10.1145/3293882.3330559
[21] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based Program Repair Using SAT. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Saarbrcken, Germany) *(TACAS'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 173–188. http://dl.acm.org/citation.cfm?id=1987389.1987408
[22] Greg4cr. 2019. Defects4J – version 1.4.0. https://github.com/Greg4cr/defects4j/tree/additional-faults-1.4.
[23] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. 2010. Test input reduction for result inspection to facilitate fault localization. *Autom. Softw. Eng.* 17, 1 (2010), 5–31. https://doi.org/10.1007/s10515-009-0056-x
[24] Dan Hao, Lu Zhang, Ying Pan, Hong Mei, and Jiasu Sun. 2008. On similarity-awareness in testing-based fault localization. *Autom. Softw. Eng.* 15, 2 (2008), 207–249. https://doi.org/10.1007/s10515-008-0025-9
[25] Dan Hao, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2009. Interactive Fault Localization Using Test Information. *J. Comput. Sci. Technol.* 24, 5 (2009), 962–974. https://doi.org/10.1007/s11390-009-9270-z
[26] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62
[27] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.
[28] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 502–514. https://doi.org/10.1109/ASE.2019.00054
[29] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 298–309.
[30] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. 467–477.
[31] 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* (San Jose, CA, USA) *(ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055
[32] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 802–811.
[33] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
[34] Edward Kit and Susannah Finzi. 1995. *Software testing in the real world: improving the process*. ACM Press/Addison-Wesley Publishing Co.
[35] 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*. ACM, 165–176.
[36] Xianglong Kong, Lingming Zhang, W Eric Wong, and Bixin Li. 2015. Experience report: How do techniques, programs, and tests impact automated program repair?. In *ISSRE*. 194–204.
[37] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 314–325.

[38] 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. https://doi.org/10.1109/TSE.2011.104

[39] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 169–180. https://doi.org/10.1145/3293882.3330574

[40] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 92.

[41] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 82–92. https://doi.org/10.1145/3180155.3180242

[42] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *PLDI* (2005), 15–26.

[43] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jin Song Dong. 2017. Feedback-based debugging. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 393–403. https://doi.org/10.1109/ICSE.2017.43

[44] 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, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 166–178. https://doi.org/10.1145/2786805.2786811

[45] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 298–312. https://doi.org/10.1145/2837614.2837617

[46] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 43–54.

[47] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. 269–278.

[48] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (01 Aug 2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[49] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 691–701. https://doi.org/10.1145/2884781.2884807

[50] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[51] 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.

[52] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[53] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[54] 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.

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

[56] 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.

[57] 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*. 609–620.

[58] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[59] William E Perry. 2007. *Effective Methods for Software Testing: Includes Complete Guidelines, Checklists, and Templates*. John Wiley & Sons.

[60] 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* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[61] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*. 24–36.

[62] Shounak Roychowdhury and Sarfraz Khurshid. 2011. A novel framework for locating software faults using latent divergences. In *ECML*. 49–64.

[63] Shounak Roychowdhury and Sarfraz Khurshid. 2011. Software fault localization using feature selection. In *International Workshop on Machine Learning Technologies in Software Engineering*. 11–18.

[64] Shounak Roychowdhury and Sarfraz Khurshid. 2012. A family of generalized entropies and its application to software fault localization. In *International Conference Intelligent Systems*. 368–373.

[65] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 648–659.

[66] Andrew Scott, Johannes Bader, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *arXiv preprint arXiv:1902.06111* (2019).

[67] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 273–283.

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

[69] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. 2017. An investigation into the use of mutation analysis for automated program repair. In *International Symposium on Search Based Software Engineering*. Springer, 99–114.

[70] Tricentis. 2019. "Tricentis Report". https://www.tricentis.com. "accessed 10-jan-2020".

[71] Westley Weimer. 2006. Patches As Better Bug Reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering* (Portland, Oregon, USA) *(GPCE '06)*. ACM, New York, NY, USA, 181–190. https://doi.org/10.1145/1173706.1173734

[72] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*. 356–366.

[73] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. 1–11.

[74] Wikipedia contributors. 2019. Software bug — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Software_bug [accessed 10-jan-2020].

[75] Wikipedia contributors. 2019. Wilcoxon signed-rank test — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test. [accessed 10-jan-2020].

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

[77] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. 2016. Revisit of automatic debugging via human focus-tracking analysis. In *ICSE*. 808–819.

[78] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. 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. https://doi.org/10.1109/TSE.2016.2560811

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

[80] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *FSE*. 52–63.

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

[82] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–272.

[83] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. 2019. An Empirical Study of Boosting Spectrum-based Fault Localization via PageRank. *IEEE Transactions on Software Engineering* (2019).

[84] 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.

[85] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2892102