Can Automated Program Repair Refine Fault Localization?

Yiling Lou∗
HCST (Peking University), China
louyiling@pku.edu.cn

Ali Ghanbari
Xia Li
Lingming Zhang
UT Dallas, USA
{Ali.Ghanbari,Xia.Li3,lingming.zhang}@utdallas.edu

Dan Hao
Lu Zhang
HCST (Peking University), China
{haodan,zhanglucs}@pku.edu.cn

Abstract
Software bugs are prevalent in modern software systems and notoriously hard to debug manually. Therefore, a large body of research efforts have been dedicated to automated software debugging, including both automated fault localization and program repair. However, the existing fault localization techniques are usually ineffective 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, we observe that in the literature their only connection 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 explore their connection in the other direction, 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, a simplistic approach using 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 localize 161 of the 395 studied bugs within Top-1, while state-of-the-art spectrum and mutation based fault localization techniques at most localize 117 within Top-1. We also demonstrate ProFL’s effectiveness under different settings. Lastly, we show that ProFL can further boost state-of-the-art fault localization via both unsupervised and supervised learning.

1 Introduction
Software bugs (also called software faults, errors, defects, flaws, or failures [65]) 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 [61]. 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 [60]. Therefore, a large body of research efforts have been dedicated to automated debugging to reduce manual-debugging efforts [6, 26, 43, 51, 60].

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 [3, 12, 22, 33, 42, 71, 72] and program repair [24, 28, 29, 35, 36, 52, 53, 63]. For example, pioneering spectrum-based fault localization (SBFL) techniques [3, 12, 22] compute the code elements covered by more failed tests or less passed tests as more suspicious, pioneering mutation-based fault localization (MBFL) techniques [42, 46, 72] inject code changes (e.g., changing > into >=) based on mutation testing [15, 20] to each code element to check its impact on test outcomes, and pioneering search-based program repair techniques (e.g., GenProg [29]) 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 [41, 67]. To date, unfortunately, although debugging has been extensively studied and even has drawn attention from industry (e.g., FaceBook [38, 58] and Fujitsu [57]), we still lack practical automated debugging techniques: (1) existing fault localization techniques have been shown to have limited effectiveness in practice [47, 68]; (2) existing program repair techniques can only fix a small ratio of real bugs [17, 21, 64] or specific types of bugs [38].

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 [3] SBFL technique is leveraged in many recent program repair techniques, including PraPR [17], CapGen [64], and SimFix [21]. Different from prior work, we aim to connect 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. Based on this insight, we designed, ProFL (Program Repair for Fault Localization), a simplistic feedback-driven fault localization approach that leverages patch-execution information from state-of-the-art PraPR [17] 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 [17, 21, 64]) 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 [17, 21, 31, 59, 64]. 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 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 96.2% with negligible effectiveness drop; (5) we also apply ProFL on a newer version of Defects4J, Defects4J (V1.4.0) [19], 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 [73, 74] and supervised [30] learning, localizing 185 and 216.8 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 dimension for improving fault localization via off-the-shelf 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).

• We have implemented a fully automated feedback-driven fault localization approach, ProFL, based on the patch-execution results from state-of-the-art program repair technique, PraPR.

• 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.

2 BACKGROUND AND RELATED WORK

Fault Localization [3, 7, 12, 22, 33, 42, 48, 54–56, 71, 72] 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 [22], Ochiai [3], and Ample [12]) or learning techniques [7, 54–56] 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) [42, 45, 46, 72], which injects changes to each code element (based on mutation testing [15, 20]) to check its impact on the test outcomes. MBFL has been applied to both general bugs (pioneered by Metallaxis [45, 46]) and regression bugs (pioneered by FIFL [72]). 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 [71], predicate switching [75], code complexity [59], and program invariant [5] information), and have also utilized supervised learning to incorporate such different feature dimensions for fault localization [30, 31, 70]. However, the effectiveness of supervised-learning-based fault localization techniques may largely depend on the training sets, which may not always be available. Therefore, researchers recently have also proposed to recompute SBFL suspiciousness by considering the contributions of different tests via the unsupervised-learning-based PageRank analysis [73, 74]. 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 [10, 13, 16, 18, 35, 36, 39–41, 44, 49, 62, 69] 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. State-of-the-art APR techniques can be divided into two broad categories: (1) techniques that dynamically monitor program executions to find deviations from certain specifications, and then heal the program under test via modifying its runtime states in case of abnormal behaviors [37, 50]; (2) techniques that directly modify program code representations based on different rules/strategies, and then use either tests or formal specifications as the oracle to validate each generated candidate patch to find plausible patches (i.e., the patches passing all tests/checks) [10, 13, 18, 36, 40, 44, 49, 69]. Among these code-representation-level techniques, those based on tests have gained popularity since testing is the prevalent methodology for detecting software bugs in practice. Based on different hypotheses, state-of-the-art code-representation-level techniques leverage a variety of strategies to generate/synthesize patches. 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) [14, 21, 29, 64]. Alternatively, semantics-based techniques use deeper semantical analyses (including symbolic execution [11, 25]) to synthesize program conditions, or even more complex code snippets, that can pass all the tests [40, 44, 69]. 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 [23]. 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 [17, 21, 64] or specific types of bugs [38].
Can Automated Program Repair Refine Fault Localization?
Conference’17, July 2017, Washington, DC, USA

3 MOTIVATION EXAMPLES

In this section, we present two real-world bug examples to show the limitations of the widely used SBFL fault localization and also the potential benefits that we can obtain from program repair.

3.1 Example 1: Math-40

We use Math-40 from Defects4J (V1.2.0) [23], 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 [4] from 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 [3], which has been widely recognized as one of the most effective SBFL formulae [31, 48, 73]. Inspired by prior work [59], 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 [17], 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. Columns “#F” and “#P” report the number of failing and passing tests on each generated patch, respectively. 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 P1 is a plausible patch, meaning that it passes all of the available tests but it might be not a genuine fix; P2 passes originally failing tests, while fails to pass 8 originally passing tests.

3.1.1 Method Signature

We use Math-40 from Defects4J (V1.2.0) [23], 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 [4] from 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 [3], which has been widely recognized as one of the most effective SBFL formulae [31, 48, 73]. Inspired by prior work [59], 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 [17], 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. Columns “#F” and “#P” report the number of failing and passing tests on each generated patch, respectively. 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 P1 is a plausible patch, meaning that it passes all of the available tests but it might be not a genuine fix; P2 passes originally failing tests, while fails to pass 8 originally passing tests.

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

| EID   | Method Signature       | SBFL | PID | #F | #P |
|-------|------------------------|------|-----|----|----|
| e1    | incrementEvaluationCount | 0.95 | P1  | 1  | 119 |
| e2    | BracketingNthOrderBrentSolver(Number) | 0.53 | P2  | 1  | 117 |
| e3    | BracketingNthOrderBrentSolver(double ...) | 0.28 | P1  | 1  | 117 |
| e4    | doSolve()              | 0.27 | P2  | 0  | 317  |
| e5    | guessed(double, ...)   | 0.36 | P2  | 0  | 316  |

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

| EID   | Method Signature       | SBFL | PID | #F | #P |
|-------|------------------------|------|-----|----|----|
| e1    | toString()             | 0.34 | P3  | 3  | 707 |
| e2    | getSortedPropTypes()   | 0.33 | P3  | 3  | 683 |
| e3    | toString(StringBuilder, ...) | 0.27 | P3  | 3  | 704 |
| e4    | functionCallHasSideEffects(Node, ...) | 0.18 | P3  | 1  | 658 |
| e5    | nodeTypeMayHaveSideEffects(Node, ...) | 0.09 | P3  | 1  | 676 |

### Table 3: Five top-ranked methods from Closure-61

| EID   | Method Signature       | SBFL | PID | #F | #P |
|-------|------------------------|------|-----|----|----|
| e1    | toString()             | 0.34 | P3  | 3  | 707 |
| e2    | getSortedPropTypes()   | 0.33 | P3  | 3  | 683 |
| e3    | toString(StringBuilder, ...) | 0.27 | P3  | 3  | 704 |
| e4    | functionCallHasSideEffects(Node, ...) | 0.18 | P3  | 1  | 658 |
| e5    | nodeTypeMayHaveSideEffects(Node, ...) | 0.09 | P3  | 1  | 676 |
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. \( P_4 \) 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 \( P_5 \) makes even more (i.e., 8) originally passing tests fail.

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 fault localization information and PraPR repair results for the Top-5 methods in Table 2.

Based on Table 2, we observe that even the non-plausible noisy patch \( P_{10} \) 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 \( P_o \), a candidate patch \( P \) can be created by modifying one or more program elements within \( P_o \). The set of all candidate patches generated for the program is denoted by \( \mathcal{P} \).

In this paper, we focus on the APR techniques that conduct only first-order program transformations, which only change one program element in each patch, such as PraPR [17]. 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 \( P_o \), its test suite \( T \), and its corresponding set of all candidate patches \( \mathcal{P} \), the patch execution matrix, \( M \), is defined as the execution results of all patches in \( \mathcal{P} \) on all tests in \( T \). Each matrix cell result, \( M[P,t] \), represents the execution result of test \( t \in T \) on patch \( P \in \mathcal{P} \), and can have the following possible values, \( \{✓, X, \emptyset\} \), which represent failed, passed, and unknown yet.

Note that for the ease of presentation, we also include the original program execution results in \( M \), i.e., \( M[P_o,t] \) denotes the execution results of test \( t \) on the original program \( P_o \).

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 \( P \) is called a Clean-Fix Patch, i.e., \( \mathcal{G}[P] = \text{CleanFix} \), if it passes some originally failing tests while does not fail any originally passing tests, i.e., \( \exists t \in T, M[P_o,t] = ✓ \land M[P,t] = ✓ \). Note that \( \mathcal{G}[P] \) returns the category group for each patch \( P \).

**Definition 4.4 (Noisy-Fix Patch).** A patch \( P \) is called a Noisy-Fix Patch, i.e., \( \mathcal{G}[P] = \text{NoisyFix} \), if it passes some originally failing tests but also fails on some originally passing tests, i.e., \( \exists t \in T, M[P_o,t] = ✓ \land M[P,t] = ✓ \). Note that \( \mathcal{G}[P] \) returns the category group for each patch \( P \).

**Definition 4.5 (None-Fix Patch).** A patch \( P \) is called a None-Fix Patch, i.e., \( \mathcal{G}[P] = \text{NoneFix} \), if it does not impact any originally failing or passing tests. More precisely, \( \exists t \in T, M[P_o,t] = ✓ \land M[P,t] = ✓ \), and \( \exists t \in T, M[P_o,t] = ✓ \land M[P,t] = ✓ \).

**Definition 4.6 (Negative-Fix Patch).** A patch \( P \) is called a Negative-Fix Patch, i.e., \( \mathcal{G}[P] = \text{NegFix} \), if it does not pass any originally failing test while fails some originally passing tests, i.e., \( \exists t \in T, M[P_o,t] = ✓ \land M[P,t] = ✓ \). 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 [3]) at the statement level, and then perform suspiciousness aggregation [59] 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 [73]). 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 re-ranks all the program elements via considering their suspiciousness and group information in tandem.

We next explain each layer in details 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 ($P_1$ to $P_6$) and the 9 tests whose statuses changed on these patches (denoted as $t_1$ to $t_9$). Based on Definition 4.2, we present $M$ in Figure 3. The first row stands for $M[P_0, T]$, the execution results of $T$ on the original buggy program $P_0$, and from the second row, each row represents $M[P, T]$, the execution results of each patch $P$ as shown in Table 1 on $T$.

4.2.1 Layer 1: Suspicious Computation. Given the original program statements, e.g., $S = \{s_1, s_2, \ldots, s_n\}$, we directly apply an off-the-shelf spectrum-based fault localization technique (e.g., the default Ochiai [3]) to compute the suspiciousness for each statement, e.g., $S[s_j]$ for statement $s_j$. Then, the proposed approach applies suspiciousness aggregation [59] 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 [9, 59]. Given the initial list of $E = \{e_1, e_2, \ldots, e_m\}$, for each $e_j \in E$, suspiciousness aggregation computes its suspiciousness as $S[e_j] = \max_{s_i \in e_j} S[s_j]$, 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, $S[e_1, e_2, e_3, e_4, e_5] = \{0.57, 0.33, 0.28, 0.27, 0.20\}$.

4.2.2 Layer 2: Patch Categorization. In this layer, ProFL automatically invokes off-the-shelf program repair engines (PraPR [17] for this work) to try various patching opportunities and record the detailed patch-execution matrix, $M$. Then, based on the resulting $M$, ProFL automatically categorizes each patch into different groups. Given program element $e$ and all the patches generated for the program, $P$, the patches occurring on $e$ can be denoted as $P[e]$. Then, based on Definitions 4.3 to 4.6, each patch within $P[e]$ for each element $e$ can be categorized into one of the four following groups, $\{\text{CleanFix}, \text{NoisyFix}, \text{NoneFix}, \text{NegFix}\}$. Recall that $G(P)$ represents the group information for $P$, e.g., $G(P) = \text{CleanFix}$ denotes that $P$ is a clean-fix patch.

For the example, the group of each patch in the motivation example is as follows: $G[P_1, P_2, P_3, P_4, P_5, P_6] = \{\text{NegFix}, \text{NegFix}, \text{NoneFix, CleanFix, NoisyFix, NoisyFix}\}$.

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: $\text{CleanFix} > \text{NoisyFix} > \text{NoneFix} > \text{NegFix}$. 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 $e$:

\[
G(e) = \begin{cases} 
\text{CleanFix} & \text{if } \exists P, P \in P[e] \land G(P) = \text{CleanFix} \\
\text{NoisyFix} & \text{else if } \exists P, P \in P[e] \land G(P) = \text{NoisyFix} \\
\text{NoneFix} & \text{else if } \exists P, P \in P[e] \land G(P) = \text{NoneFix} \\
\text{NegFix} & \text{else if } \exists P, P \in P[e] \land G(P) = \text{NegFix} 
\end{cases}
\]

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

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

4.2.4 Layer 4: Feedback-driven Re-ranking. 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 NoneFix 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 $R[e_1, e_2]$ to denote the total-order ranking between any two program elements, it can be formally defined as:

\[
R[e_1, e_2] = \begin{cases} 
e_1 \geq ne_2 & \text{if } G[e_1] > G[e_2] \lor \\
G[e_1] = G[e_2] \land S[e_1] \geq S[e_2] & \text{else if } G[e_1] = G[e_2] \\
G[e_1] = G[e_2] \land S[e_1] \geq S[e_2] & \text{else if } G[e_1] = G[e_2] \land S[e_1] \geq S[e_2] \land S[e_2] \
\end{cases}
\]

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 [17] 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, e.g., $R_1$ and $R_2$ study the two different rules 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 [3] formula, all the other formulae in SBFL can be adopted in ProFL. We study all the 34 SBFL formulae considered in prior work [31, 59]. The impact of these formulae on ProFL would be studied later.

**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 [20]. 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, 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 practice, which we call a partial matrix, $\mathbb{M}_p$. Recall Definition 4.2, different from $\mathbb{M}_f$, the cells in $\mathbb{M}_p$ can be $\emptyset$ besides ✓ and X. For example, when $t$ is not executed on $\mathcal{P}$, $\mathbb{M}_p[\mathcal{P}, t] = \emptyset$.

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:

$$
\mathbb{M}_p = 
\begin{bmatrix}
\mathcal{P}_0 \\
\mathcal{P}_1 \\
\mathcal{P}_2 \\
\mathcal{P}_3 \\
\mathcal{P}_4 \\
\mathcal{P}_5 \\
\mathcal{P}_6 \\
\end{bmatrix}
= 
\begin{bmatrix}
✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\
✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\
✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\
✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\
✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\
✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\
✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ & ✓ \\
\end{bmatrix}
\tag{3}
$$

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 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 experimental 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?

### 5.2 Benchmark

We conduct our study on all bugs from the Defects4J benchmark [23], which has been widely used in prior fault-localization work [30, 31, 48, 59, 73]. Defects4J is a collection of reproducible real bugs.

**Table 3: Finer-grained patch categorization rules**

| ID  | Extended Categorization Aggregation Rules                         |
|-----|------------------------------------------------------------------|
| $R_1$ | CleanAllFix $\rightarrow$ CleanPartFix $\rightarrow$ NoisyFix $\rightarrow$ NoneFix $\rightarrow$ NegFix |
| $R_2$ | CleanPartFix $\rightarrow$ CleanAllFix $\rightarrow$ NoisyFix $\rightarrow$ NoneFix $\rightarrow$ NegFix |

**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 | 106 | 7,927 | 90K |
| Mockito | Mockito framework | 38   | 1,366 | 23K |

| ID             | Name               | #Bug | #Test | LoC  |
|----------------|--------------------|------|-------|------|
| Defects4J (V1.2.0) | 395            | 21,475 | 234K  |      |
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 [19]. Therefore, we further perform the first fault localization study on Defects4J V1.4.0 to reduce the threats to external validity.

We present the details of the used benchmarks in Table 4. Column “ID” presents the subject IDs used in this paper. 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 from our evaluation due to build/test framework incompatibility with PraPR [17]. 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 [9, 59]. (b) Mutation-based (MBFL): we consider two representative MBFL techniques, MUSE [42] and Metallaxis [46]. (c) Hybrid of SBFL and MBFL (MCBFL): we also consider the recent MCBFL [48], which represents state-of-the-art hybrid spectrum- and mutation-based fault localization. Furthermore, we also include state-of-the-art learning-based fault localization techniques: (a) Unsupervised: we consider state-of-the-art PRFL [73] and PRFLMA [74] (which further improves PRFL via suspiciousness aggregation) that aim to boost SBFL with the unsupervised PageRank algorithm. (b) Supervised: we further consider state-of-the-art supervised-learning-based fault localization, DeepFL [30], which outperforms all other learning-based fault localization [31, 59, 70]. Note that, SBFL and Metallaxis can adopt different SBFL formulae, and we by default uniformly use Ochiai [3] since it has been demonstrated to perform the best for both SBFL and MBFL [31, 48, 73].

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 [31, 59] 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) O1: the default order in original test suite; (ii) O2: 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) O3: running originally-passing tests first and then originally-failed tests. The partial matrices collected by these orders are denoted as $h_{p}^{O1}$, $h_{p}^{O2}$, and $h_{p}^{O3}$ 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 [5, 30, 31, 59, 73], 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 [27, 47]. We use the following widely used metrics [30, 31]:

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 [47], 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 [27]. Therefore, following prior work [30, 31, 73, 76], 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 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 [30, 31], 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 [17], 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. Note that, ProFL does not rely on any specific APR technique, since the feedback input (i.e., patch-execution information) for our approach is general and can work with 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 [8] to instrument the code on-the-fly via JavaAgent [1]. (ii) To collect the mutation testing information required by MBFL, we use state-of-the-art PIT mutation testing framework [2] (Version 1.3.2) with all its available mutators, following prior MBFL work [30, 31]. 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 [48]) required by Metallaxis. For PRFL, PRFLMA, and DeepFL, we directly used the implementation released by the authors [30, 74].

All the experiments are conducted on a Dell workstation with Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz and 330GB RAM, running Ubuntu 18.04.1 LTS.
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      | 82    | 167   | 198   | 97.58| 106.2|
| Metallaxis | 94    | 191   | 244   | 14.28| 16.93|
| MCBFL     | 132   | 227   | 268   | 17.98| 23.24|
| ProFL     | 161   | 255   | 286   | 9.48 | 14.37|

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 [31, 73, 74] are consistent. We also directly used the original implementations from prior work [30, 74].

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 [27, 47] and widely used in previous work [30, 31, 59, 73].

Threats to external validity mainly lie in the benchmark suites used in our experiments. To reduce this threat, we chose the widely used Defects4J benchmark, which includes hundreds of real bugs collected during real-world software development. To further reduce the threats, compared to previous work, we not only used the original version of Defects4J, but also conducted the first fault localization evaluation on an extended version of Defects4J.

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, 29 more than MCBFL, 44 more than aggregation-based SBFL, 67 more than Metallaxis, and 79 more than MUSE. In addition, MAR and MFR values are also significantly improved (e.g., at least 33.61% improvements for MFR compared with all existing techniques), indicating a consistent improvement for all buggy elements in the ranked lists. Note that we observe that SBFL outperforms state-of-the-art MBFL techniques in terms of Top-ranked bugs, which is not consistent with prior fault localization work at the method level [31]. We find the main reason to be that the prior work did not use suspicious aggregation (proposed in parallel with the prior work) for SBFL. This further demonstrates 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 $G_i$, for each bug in the benchmark, we use metric $\text{Ratio}_{i,p}$ to represent the ratio of the number of buggy elements (i.e., methods in this work) categorized into group $G_i$ to the number of all elements categorized into group $G_i$. Formally, it can be presented as:

$$\text{Ratio}_{i,p}(G_i) = \frac{|\{e \in P \mid e \in G_i \land P \in \mathbb{P}[e]\} \land e \in B|}{|\{e \in P \mid e \in G_i \land P \in \mathbb{P}[e]\}|}$$  \hspace{1cm} (4)

Figure 5: $\text{Ratio}_{i,p}$ distribution for different patch groups

Table 6: Impacts of finer patch categorization

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

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.

Figure 6: Comparison of ProFL and SBFL over all formulae

(a) Top-1

(b) MAR

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
### Table 7: Impacts of using mutation or repair information

| Tech Name | Top-1 | Top-3 | Top-5 | MFR   | MAR  |
|-----------|-------|-------|-------|-------|------|
| MUSE$_{PIT}$ | 82    | 167   | 198   | 97.58 | 106.2 |
| MUSE$_{PraPR}$ | 95    | 172   | 207   | 38.79 | 43.1 |
| Metallaxis$_{PIT}$ | 94    | 191   | 244   | 14.28 | 16.93 |
| Metallaxis$_{PraPR}$ | 77    | 170   | 211   | 21.42 | 22.94 |
| MCBFL$_{PIT}$ | 152   | 227   | 268   | 17.98 | 23.24 |
| MCBFL$_{PraPR}$ | 130   | 228   | 267   | 18.03 | 23.28 |
| ProFL$_{PIT}$ | 141   | 238   | 266   | 13.24 | 20.33 |
| ProFL$_{PraPR}$ | 161   | 255   | 286   | 9.48  | 14.37 |

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

### Table 8: Impacts of using partial matrices

| $M_p$ | Tech Name | Top-1 | Top-3 | Top-5 | MFR   | MAR  |
|-------|-----------|-------|-------|-------|-------|------|
| $M_{(O_1)}$ | MUSE$_{PraPR}$ | 92    | 148   | 172   | 118.56 | 125.11 |
| $M_{(O_2)}$ | Metallaxis$_{PraPR}$ | 64    | 128   | 167   | 113.9 | 126.79 |
| $M_{(O_3)}$ | ProFL | 165   | 254   | 287   | 15.46 | 20.96 |
| $M_{(O_4)}$ | Metallaxis$_{PraPR}$ | 87    | 130   | 152   | 191.71 | 206.0 |
| $M_{(O_5)}$ | ProFL | 169   | 252   | 285   | 9.17  | 15.15 |
| $M_{(O_6)}$ | ProFL | 158   | 244   | 278   | 19.07 | 25.25 |

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 132 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 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 increase by 40.0% 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.

### 6.2.2 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). Other techniques generally perform better than APR (no matter using mutation or repair information) only localizes 132 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.

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

### 6.2.4 RQ2d: Impact of partial execution matrix.

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 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变异对传统MBFL的公平比较，例如，原始MUSE被命名为MUSE$_{PIT}$而新的MUSE变量则被命名为MUSE$_{PraPR}$。表7呈现了实验结果，显示了两种ProFL和基于突变信息的前向突变技术的差异。

**Finding 4:** ProFL仍然在使用突变反馈信息时表现良好，但其有效性的降低与使用程序修复之间的提高不相匹配，表明程序修复在故障定位中的优先级。
Table 9: Collection time for full and partial matrices

| Subject | Timef | Timep | Reduced Time | Reduced Ratio |
|---------|-------|-------|--------------|---------------|
| Lang-1  | 0m38s | 0m31s | 0m7s         | 18.4%         |
| Closure-1| 2m58s26s | 1m00s33s | 147m35s58s | 19.5%         |
| Mockito-1| 2m43s | 2m43s | 29m46s       | 19.7%         |
| Chart-1  | 0m41s | 0m41s | 148m33s      | 19.9%         |
| Math-1   | 6m24s | 7m53s | 60m31s       | 28.5%         |
| Total    | 327m142s | 125m21s | 114m360 | 96.2%         |

Table 10: Results on Defects4J (V1.4.0)

| Tech Name | Top-1 | Top-3 | Top-5 | MFR | MAR |
|-----------|-------|-------|-------|-----|-----|
| SBFL      | 59    | 102   | 124   | 13.8 | 20.44 |
| MUSE      | 42    | 75    | 82    | 53.97 | 60.17 |
| Metallaxis | 45    | 90    | 102   | 19.05 | 24.9 |
| MCBFL     | 65    | 110   | 130   | 13.28 | 19.88 |
| ProFL     | 78    | 117   | 131   | 12.01 | 17.96 |

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. 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 158 Top-1 bugs. One potential reason that $h_{p}^{f}$ 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., $h_{p}^{f} \otimes h_{p}^{f}$), further demonstrating that ProFL is practical.

We next present the cost reduction benefits that partial execution matrices can bring to speed up the ProFL fault localization process. 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 “Timef” presents the time for executing all tests on each candidate patch. Column “Timep” 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., executing originally failed tests first then passed tests). Columns “Reduced Time” and “Reduced Ratio” show the reduced time and the reduction ratio from Timef to Timep. We use 4 threads for executing both PraPR variants. From the table, we can observe that partial execution matrix collection can overall achieve 96.2% reduction compared to full matrix collection. Furthermore, using partial execution matrices, 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 effectiveness drop when using only partial matrices).

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 ProFLMA [74]) and supervised-learning-based (i.e., DeepFL [30]) 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 ProFLMA, denoted as ProFLPRFL and ProFLPRFLMA, 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 ProFLDeepFL. The experimental results are shown in Table 11. Note that DeepFL results are averaged over 10 runs due to the DNN randomness [30]. 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 PRFLMA only localizes 136 bugs within Top-1. Second, ProFL can significantly boost unsupervised-learning-based fault localization. E.g., ProFLPRFLMA 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 [31], FLUCCS [59], and CombineFL [76] only localize 156, 160, and 168 bugs from the same dataset within Top-1, respectively [30, 76]. Lastly, we can observe that ProFL even boosts state-of-the-art supervised-learning-based technique. E.g., it boosts DeepFL to localize 216.8 bugs within Top-1, the best fault localization results on Defects4J with supervised learning to our knowledge.
Can Automated Program Repair Refine Fault Localization?

ProFL, the first 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. Lastly, ProFL even boosts state-of-the-art fault localization via both unsupervised and supervised learning. In the near future, we will work on tentative program repair, a new direction enabled by this research to allow fault localization and program repair to boost each other for more powerful debugging, e.g., patch execution results from an initial program-fixing template set can enable precise fault localization for applying later more advanced program-fixing template sets later for cost-effective debugging.

REFERENCES

[1] 2019. JavaAgent. https://docs.oracle.com/javase/7/docs/api/javlang/instrument/package-summary.html
[2] 2019. Pitest. http://pitest.org
[3] Rui Abreu, Peter Zoetewey, 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.
[4] Apache. 2019. Commons Math. https://commons.apache.org/proper/commons-math/. Accessed Aug-22-2019
[5] Tien-Duy B L, David Lo, Claire Le Goues, and Lars Grunzke. 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.
[6] Antonio Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In 2007 Future of Software Engineering. IEEE Computer Society, 85–103.
[7] Lionel C Briand, Yvan Labiche, and Xuetao Liu. 2007. Using machine learning to support debugging with tarantula. In ISSRE ’07: 137–146.
[8] Eric Bruneton, Romain Lenglert, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems 30, 19 (2002).
[9] Junjie Chen, Jiaqin Han, PeiYi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler Bug Isolation via Effective Witness Test Program Generation. In FSE. To appear.
[10] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based Program Repair Without the Contrasts. In Proceedings of the 24nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, Piscataway, NJ, USA, 637–647. http://dl.acm.org/citation.cfm?id=3155562.3155642
[11] Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. TSE 3 (1976), 215–222.
[12] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight mutation based fault localization. In 2005 Future of Software Engineering. IEEE Computer Society, 85–103.
[13] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from a Static Object Behavior Analysis. 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
[14] 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
[15] Richard A DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978.Hints on Testing Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). IEEE, 298–309.
[16] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test execution information to assist fault localization. In ICSE ’02: 467–477.
[17] 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 (ISSTA 2014). ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610834.2628055
[18] Dongsun Kim, Jaecheong Nam, Jeawoo 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.
[19] James C King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (1976), 385–394.
[20] Edward Kit and Susannah Finzi. 1995. Software testing in the real world: improving the process. ACM Press/Addison-Wesley Publishing Co.
[21] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shoping Li. 2016. Practitioners’ expectations on automated fault localization. In Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 165–176.
[22] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based Program Repair Without the Contrasts. In Proceedings of the 24nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, Piscataway, NJ, USA, 637–647. http://dl.acm.org/citation.cfm?id=3155562.3155642
[23] Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair?. In ISSTA. To appear.
[24] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In ISSTA. To appear.
[25] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In Proceedings of the 43nd 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
[26] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI ’14, Edinburgh, United Kingdom – June 09 - 11, 2014, 227–238. https://doi.org/10.1145/2594291.2594337
[27] Sanghoon Lim, Jeff Gordon, and Andrew Scott. 2019. Scalable statistical bug isolation. PML (2005), 15–26.
[28] 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 24nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, Piscataway, NJ, USA, 637–647. http://dl.acm.org/citation.cfm?id=3155562.3155642
[29] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. IEEE Computer Society, 153–162.
