Revisiting Defects4J for Fault Localization in Diverse Development Scenarios

Md Nakhla Rafi*, An Ran Chen†, Tse-Hsun (Peter) Chen*, and Shaohua Wang‡
*Software Performance, Analysis, and Reliability (SPEAR) Lab,
Concordia University, Montréal, Québec, Canada
Email: {r_mdnakh, peterc}@encs.concordia.ca
†University of Alberta, Edmonton, Canada
Email: anran6@ualberta.ca
‡Central University of Finance and Economics, Beijing, China
Email: davidshwang@ieee.org

***Abstract*—Defects4J stands out as a leading benchmark dataset for software testing research, providing a controlled environment to study real bugs from prominent open-source systems. While Defects4J provides a clean and valuable dataset, we aim to explore how fault localization techniques perform under less-controlled development scenarios. In this paper, we revisited Defects4J to study developers' changes to fault-triggering tests after the bugs were reported/fixed. We aim to introduce a new evaluation scenario within Defects4J, focusing on the implications of regression tests and test changes added after the bug was fixed. We analyze when these tests were modified relative to bug report creation and examine spectrum-based fault localization (SBFL) performance in less-controlled settings. Our findings show that 1) 55% of the fault-triggering tests were added to replicate the bug or test for regression; 2) 22% of the tests were changed after the bug reports, incorporating information related to the bug; 3) developers often update tests with new assertions or changes to match source code updates; and 4) SBFL performance differs significantly in less-controlled settings (down by at most 90% for Mean First Rank). Our study points out the diverse development scenarios in the studied bugs, highlighting new settings for future SBFL evaluations and bug benchmarks.**

***Index Terms*—Fault localization, Defects4J, Empirical study**

I. Introduction

Locating and fixing bugs is an expensive and time-consuming task, especially due to the ever-increasing complexity of modern software. Prior research [8] has found that developers spend a significant portion of their time on bug-fixing activities. To assist developers in locating and fixing bugs, prior research has proposed different fault localization (FL) techniques [38], [48], [53], [56], [66] and automated program repair (APR) techniques [33], [34], [50]. Spectrum-based fault localization (SBFL), one of the most widely used fault localization techniques, analyzes the code coverage to suggest a ranked list of possible buggy statements in the code based on failed tests [4], [24]. They operate on the intuition that statements covered by more failed tests and fewer passed tests are more likely to contain a bug. These techniques are also crucial for automated program repair, as they help identify potential buggy locations [38].

Given the increasing community attention on software testing research, prior studies introduced benchmarks for automatic program repair and fault localization. For example, there are many benchmarks available for both C and Java, such as ManyBugs [30] QuixBugs [37], Bugs.jar [49], Bugswarm [55], and Defects4J [1]. These benchmarks often provide code coverage, test failure information, and the corresponding bug fix. Researchers can easily evaluate new fault localization techniques using these benchmarks.

Particularly, Defects4J is one of the most widely-used benchmarks [43], [44], [48], [53], [66]. Defects4J has been instrumental in advancing software testing research by providing clean, easily executable data. Defects4J provides a controlled environment by isolating real bugs from version control histories. It offers both buggy and fixed versions and comprehensive test suites, which greatly enhance the reliability and reproducibility of testing studies. Defects4J 2.0 contains 835 bugs from 17 open-source Java systems, and every bug is accompanied by at least one failing test that triggers it (i.e., *fault-triggering test*).

While Defects4J aims to provide complete information to replicate bugs, some bugs may not have tests available when they were first reported. Some prior studies [22], [40] found that developers derived certain fault-triggering tests in Defects4J from system versions where the bug had already been resolved. Hence, although the structured data provided by Defects4J has greatly facilitated research, it only represents one of the many perspectives of development scenarios when all the data is available. Real-world development scenarios, including evolving test suites, ad-hoc fixes, and various developer-specific practices, can be less controlled.

In this paper, we revisited Defects4J to study and organize a collection of bugs that can serve as a benchmark for evaluating fault localization techniques in less controlled settings. We conduct the first comprehensive study on the fault-triggering tests in Defects4J. We define *fault-triggering tests* as tests that fail when the bug is introduced and pass once the bug is fixed [1], [59]. Firstly, we classify the tests in relation to developers' bug-fixing activities (e.g., a test is newly added or modified after a bug was reported). We find that 55% of the fault-triggering tests in Defects4J were newly added after the bugs were fixed, and 22% of the tests were modified

after the bugs were reported for replication and regression testing purposes. Given the large proportion of such tests, our findings confirm a common development practice—where tests are often written or adapted after a bug is reported—and highlight the need to examine how fault localization techniques perform when applied in these evolving and less-controlled settings.

Secondly, we analyze developers' modifications on the fault-triggering tests and identify different categories of reasons why developers modify them. We find that developers often add new assertions or modify the tests to cope with source code changes and fixes (i.e., 77% of the analyzed cases). Finally, we examine how existing fault localization techniques perform on these datasets by applying state-of-the-art SBFL techniques to the commits before and after the bugs are fixed. Our findings show that spectrum-based fault localization techniques perform significantly worse in these less controlled settings. This highlights the importance of proposing and evaluating fault localization techniques on controlled datasets like Defects4J and other datasets that reflect more variable development scenarios.

Building on the strengths of Defects4J, our work further contributes by organizing a dataset that distinguishes between bugs with and without post-resolution information in tests, as well as the coverage of tests when the bugs were reported. This new dataset is valuable for 1) evaluating new and existing fault localization and automated program repair techniques in various development settings, 2) providing insights and guidance for future bug benchmarks, and 3) offering a better understanding of the developers' bug-fixing process to suggest additional settings for evaluation. We hope the study can complement the dataset provided by Defects4J and enhance its utility for research that simulates diverse development scenarios.

The main contributions of this paper are:

- Our findings show that most fault-triggering tests (77%) were modified after bugs were reported or even fixed, highlighting other perspectives of software development scenarios that may happen in less-controlled settings.
- We manually studied and categorized the modification on fault-triggering tests. We uncovered five categories, such as modifying tests to co-evolve with code and adding new assertions. The uncovered categories may provide a starting point for understanding developers' bug-fixing process and assist in proposing future techniques or creating/extending bug benchmarks.
- We evaluated the performance of SBFL techniques in less-controlled settings by comparing the reported version (before the bug was reported) with the version provided by Defects4J. The fault localization results on less-controlled settings are significantly worse than that on the Defects4J version, with a degradation up to –90% for Mean First Rank and –85% for Mean Average Rank.
- We discussed the implications of our findings and highlighted future research direction. We also made our dataset publicly available online [2], which includes a comprehensive list of the bugs that contain tests added or modified after the bug report, the details of our manual study results, and the test results and coverage that we collected for the buggy commit (before the bug was reported).

Our study points out the diverse development scenarios in the studied bugs. Future research may use our annotated data to study fault localization and automated program repair in less-controlled settings.

Paper Organization. Section II discusses the background and motivation of this study. Section III presents our studied dataset and the motivation, approach, and results of the research questions. Section IV discusses the findings and highlights potential future research directions. Section V presents related work to our study. Section VI discusses threats to validity. Finally, Section VII concludes the paper.

II. Background and Motivation

The Defects4J v2.0.0 benchmark contains 835 bugs from 17 Java open source systems [1]. All 835 bugs are extracted from different phases of software development, and the 17 projects span a wide range of domains and maturities. The benchmark aims to facilitate research in software testing and debugging. Due to its ease of use and the realistic nature of the bugs, Defects4J has been widely used for research in fault localization [38], [48], [53], [56], [66], automated program repair [18], [43], [44], [63], and automated test generation [16], [51], [65].

In Defects4J, each bug comes with at least one failed test to ensure reproducibility. This failed test is known as the *fault-triggering test*. As not all bugs are guaranteed to have a fault-triggering test at the time they are first uncovered, Defects4J utilizes an automated step to mine candidate fault-triggering tests from the bug fix and buggy commit of the system [1]. Specifically, a fault-triggering test must deterministically pass on the fixed commit and fail on the buggy commit. Every bug and fault-triggering test is then manually examined to eliminate irrelevant code changes, such as the addition of new features. By default, Defects4J uses developer-written tests as fault-triggering tests to reproduce the bugs.

However, fault-triggering tests may be mined from the bug-fixing commit where developers may have already fixed the bug or were in the process of fixing it. This can include added information about the bug that was not available at the time the bug was reported in the tests. Table I provides an excerpt of a bug, CLI-51, from the Defects4J benchmark with a test that has developer knowledge. CLI-51 is a bug from Commons-Cli where the code misinterpreted parameter values as new parameters. When developers first received this bug report, there was no test failure in the system. Developers provided a patch to fix the bug and commented in the report that a fault-triggering test (i.e., BugCLI51Test) was developed as part of the patch for regression testing. The fault-triggering test was introduced after the corresponding bug fixes. In particular, developers developed the test based on the content of the bug report and the bug fix. The fault-triggering test verifies

TABLE I: An excerpt of the Bug report CLI-51 from the Defects4J benchmark.

BugID	CLI-51
Summary	Parameter value "-something" misinterpreted as a parameter
Developer's comment	"*Fix so parser doesn't burst options which are not defined. (-s) in the above case. Includes unit test [BugCLI51Test].*"

the parameter value "-t -something" that triggers the bug, as the bug report mentions. As a result, the test contains post-resolution information, which provides hints on the causes and location of the bug in the source code.

Although prior studies [10], [23], [26], [57] suggest that some tests in Defects4J may include information from the bug-fixing process, there has been no systematic study differentiating these tests from those reflecting more development-centric scenarios. Fault localization is crucial for helping developers locate faults [7], [41], [48], [70] and for guiding automated program repair techniques [29], [38], [46]. Thus, evaluating these techniques in varied development scenarios and curating a bug collection that serves as a benchmark for fault localization research in less-controlled settings is essential.

III. Study Design and Results

In this section, we first discuss an overview of the studied systems. Then, we present the motivation, approach, and results for the three research questions (RQs).

A. Overview of the Studied Systems

We performed our study on the Defects4J (V2.0.0) benchmark [1], which includes real and reproducible faults from a wide range of systems. Defects4J forms the basis of many prior studies on fault localization [31], [48], [53], [57], [66], where these studies use it as the benchmark dataset for evaluation and comparison with state-of-the-art. Table II provides an overview of our studied systems. We collected the number of lines of code (LOC) and the number of tests from the HEAD version of each system. The sizes of the studied systems in Defects4J range from 4K to 90K lines of code, and the benchmark contains 1,655 fault-triggering tests. We did not consider the system Chart from Defects4J since it does not use Git as the version control system. For our study, we rely on analyzing the development history in Git repositories to understand the changes in the fault-triggering tests. We studied 809 bugs from 16 systems in total. Since one bug may have more than one fault-triggering test, there are more fault-triggering tests than bugs.

TABLE II: An overview of our studied systems from Defects4J

System	**#Bugs**	**LOC**	**#Tests**	**Fault-triggering Tests**
Cli	39	4K	94	66
Closure	174	90K	7,911	545
Codec	18	7K	206	43
Collections	4	65K	1,286	4
Compress	47	9K	73	72
Csv	16	2K	54	24
Gson	18	14K	720	34
JacksonCore	26	22K	206	53
JacksonDatabind	112	4K	1,098	132
JacksonXml	6	9K	138	12
Jsoup	93	8K	139	144
JxPath	22	25K	308	37
Lang	64	22K	2,291	121
Math	106	85K	4,378	176
Mockito	38	11K	1,379	118
Time	26	28K	4,041	74
Total	809	409K	25,708	1,655

RQ1: Were Fault-triggering Tests Added/Modified After a Bug Was Reported?

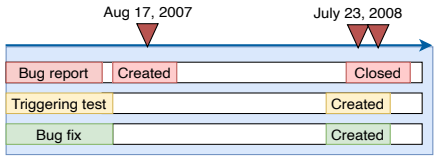
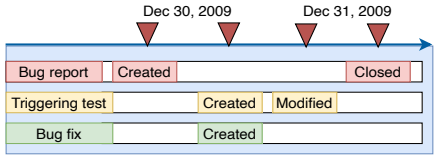
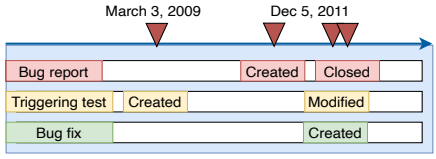
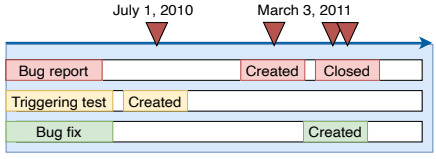
Motivation. When localizing faults, SBFL techniques primarily rely on analyzing the coverage of fault-triggering tests. Prior research [10], [26], [57] has highlighted that Defects4J may include some tests added by developers after the bug was reported or fixed. Although the data provided by Defects4J has greatly facilitated research, it primarily represents scenarios where most data is available in a controlled setting. Therefore, in this RQ, we classify the tests in relation to developers' bug-fixing activities to examine the timelines of test modifications and additions for every bug, from the creation of the bug report until its resolution. We also investigate whether these changes incorporate information from the developers' debugging process. These findings aim to provide a clearer distinction between bugs with cleaner data and those reflecting less controlled development conditions, ultimately highlighting alternative perspectives on software development scenarios that may occur in more variable settings.

Approach. We conducted a tool-assisted manual study on the timelines of events for every bug. We first collected the bug report, fault-triggering tests, and bug-fixing patches for all of the 809 studied bugs. We then identified the events (e.g., bug report creation) associated with each piece of information. We analyze these events automatically in relation to bug resolution and manually review the test modifications to ensure their relevance to the bug. Below, we discuss our data collection process in detail.

Bug report creation and resolution date: To determine the time interval of the bug resolution and identify whether the fault-triggering tests were modified during this period, we collected the creation and resolution time of the bug reports. We retrieved the bug reports from the bug tracking systems (i.e., Jira and GitHub) using REST APIs and the bug IDs provided in Defects4J. We then extracted the creation and resolution time from the "Created" and "Resolved" fields (or the issue creation and closed dates on GitHub) of each bug report. Using the creation and resolution dates, we can determine if a test was added/modified after a bug was reported.

Date of fault-triggering test creation and modification: We identified all the commits that are associated with the fault-triggering tests and analyzed when the commits happened (e.g., before or after the bug was reported/fixed). We used the git command "`git log -L:[funcname]:[file]`"

TABLE III: Timelines of the changes on the fault-triggering tests. Note that the same bug may belong to more than one pattern because a bug may have more than one fault-triggering test.

Patterns	Description	Example Timeline	#Tests	#Bugs
Pattern 1	Fault-triggering tests were newly added after the bug was reported.		872 (53%)	558
Pattern 2	Fault-triggering tests were newly added then modified after the bug was reported.		44 (2%)	31
Pattern 3	Fault-triggering tests were modified after the bug was reported and before the bug was marked as fixed.		362 (22%)	155
Pattern 4	Fault-triggering tests were not modified after the bug was reported and before the bug was marked as fixed.		377 (23%)	149
Total			1,655	-

to identify the list of commits that modified the fault-triggering test and the modification date.

Bug Fix Date: In addition to the bug report creation/resolution time, we study the time of the bug fix to understand if a test was modified before or after the fix became available. To find the bug fix date, we first retrieved the resolution date of the bug report from the bug tracking system. Using this date and the bug ID, we traced related commits by analyzing commit messages referencing the bug ID. We then used the "`git log [commit]`" command to inspect these commits in detail. For cases with multiple related commits, we identified the final bug-fix commit as the one where the fault-triggering test failed on the buggy version but passed on this commit. We then aligned the bug fix dates with modifications made to the fault-triggering tests.

Change patterns of the tests and their relevance to the bugs: We arranged the events – bug report creation and resolution, creation and modification of fault-triggering tests, and bug fix time – in chronological order to reconstruct the timeline. In particular, we focused on the creation and modifications of fault-triggering tests with respect to bug resolution. We used an automated script to sequence the events into timelines. We then manually studied the commit messages and related code changes to examine if the added/modified tests included developer knowledge about the bug. Our collected data is publicly available [2].

Results. ***We find that 77% of the fault-triggering tests in Defects4J were either added or modified during the bug fixing process (i.e., after the bug report was created).*** In total, we uncovered four timelines of change patterns on the fault-triggering tests. Table III shows the uncovered change patterns and corresponding timelines of the events. In summary, our manual analysis reveals that developers often modify or add tests after bug reports are created, incorporating additional bug-related information into all tests from Pattern 1 and Pattern 2, and the majority of tests from Pattern 3 (357/362). Below, we discuss each pattern in detail.

Pattern 1: Test created after bug report creation (53%). When developers are trying to fix a bug, they often rely on fault-triggering tests to understand and replicate the problem [6], [10], [14], [52], [59]. However, as illustrated in Table III, we found that in Defects4J, a large portion of these fault-triggering tests may not exist before the bug report was created. The most common change pattern is that the tests were added to the system only after developers began to fix the bug. As an example, in CLI-144, the bug report was created on August 17, 2007. On July 23, 2008, developers created a new fault-triggering test called BugCLI144Test as part of the bug fixing commit (the test was named using the bug report ID). This new fault-triggering test was added because existing tests were unable to capture the reported bug. Thus, the test contains information on the ground truth of the bug-fixing location. However, this test was marked as the fault-triggering

test in Defects4J. Through manual inspection, we found that developers added all the fault-triggering tests belonging to this pattern as part of the bug-fixing process. Namely, nearly 50% of the fault-triggering tests in Defects4J were regression tests that were added to replicate/prevent the bug.

Pattern 2: Test created and modified after bug report creation (2%). As shown in Table III, after a bug report is created, developers may create initial versions of the fault-triggering tests that require further enhancement (e.g., the initial version is not able to cover all possible scenarios). Nevertheless, in our manual analysis, similar to Pattern 1, we found that all these fault-triggering tests were created for regression testing purposes and contained information specific to the bug-fixing process. As an example, in MATH-320, the developer initially added a bug fix and a new test to reproduce the bug. However, the developer commented that the fix was incomplete and shared the test to facilitate discussions with other developers. Later, the developer applied a patch to fix the bug along with the updated test. For all the tests that belong to Pattern 2, either the commit messages or the names of these fault-triggering tests contain the ID of the bug report, further indicating their connection to the bug-fixing process.

Pattern 3: Test modified after bug report creation (22%). In practice, some bugs reported by users or other developers cannot be revealed by existing tests. Hence, when fixing these bugs, developers may modify and enhance the tests for regression testing purposes. For example, in COMPRESS-10, the test UTF8ZipFilesTest was enhanced as part of the bug fix. Developers modified the assertions to better capture the bug. It is possible that the test modification is not related to the bug fix. However, after our manual analysis, we found that 99% (357/362) of the tests contained changes that altered the test execution for replicating the bug, while the remaining 1% (5/362) did not introduce new knowledge to the tests (e.g., code re-styling, and enabling or disabling a failed test). In short, we find that most modifications to the fault-triggering tests were made during the bug-fixing process, incorporating information related to the bug after it was reported.

Pattern 4: Test unmodified during bug resolution (23%). We found that developers may fix the bug without making any changes to the fault-triggering tests. As an example (CLOSURE-79), when the problem was initially uncovered, the fault-triggering test testPropReferenceInExterns3 failed. Developers work on the bug fix without having to change the test as it was working as intended (i.e., failing upon unexpected behavior). The fault-triggering tests were not changed during the resolution of the bug report. This pattern consists of 378 manually verified fault-triggering tests.

Discussion. In our manual analysis, we identified four change patterns in fault-triggering tests. Based on Patterns 1 and 2, *55% (916/1,655) of the fault-triggering tests were created after the bug report was filed*. These tests were not involved in the initial detection of bugs and included information from the bug-fixing process, potentially providing information on the bug's location. While leveraging these tests in fault localization can be beneficial, it also introduces potential biases in how effectively code coverage can identify faults in less controlled environments. Additionally, 22% (362/1,655) of the tests were modified after the bug report, resulting in significant differences from their initial versions and potentially influencing fault localization outcomes. We found that 23% of the studied fault-triggering tests detected bugs (i.e., the tests failed), accounting for 19% (150/809) of the total bugs.

Fault-triggering tests in Defects4J might not be available in all situations, as highlighted in our findings. Future research should consider finding other data (e.g., logs or stack traces) to complement fault localization in case such tests are not available [11], [47]. Our findings highlight the importance of considering the context and timing of test creation and modification, emphasizing the need to evaluate fault localization techniques on datasets reflecting more variable development scenarios.

RQ1-Takeaway. The majority of the fault-triggering tests in Defects4J (77%) incorporate details from the bug-fixing process. These tests were either added or modified to replicate the bug or to prevent future regressions. Only 23% of the tests were able to detect the bugs (19% of the total bugs) as intended.

RQ2: How Do Developers Modify the Fault-triggering Tests?

Motivation. Automated testing environments often detect bugs first through failing tests [12], [17], [25], [27]. Once developers identify a bug, they analyze these test failures to debug the issue and understand its root causes. However, as we found in RQ1, many fault-triggering tests may not exist before the bug report. Even when they exist, they can only trigger the bug after developers modify them during the bug-fixing process. In this RQ, we manually study the modifications made by developers to the fault-triggering tests and discuss the common reasons behind them.

Approach. We conducted a manual study on 16 studied systems by analyzing a sample of 300 fault-triggering tests drawn from 1,361 modified tests collected from Patterns 1, 2, and 3. The sample size was determined to achieve a 95% confidence level with a 5% confidence interval, ensuring statistical validity for the population size. We use the stratified sampling strategy [45] to obtain the number of samples for each studied system that is proportional to their total number of tests. For each test, we study its code changes, code comments, commit message, and corresponding bug report and bug fix to understand the potential motivation for its modification. Following prior studies [15], [35], [36], we conduct our manual study in three phases using the open coding method.

Phase 1: The first author manually reviews the tests to create a preliminary list of categories for 100 fault-triggering tests that are randomly selected. The author also lists the justifications for creating each category regarding its code changes, code comments, commit messages, bug reports, and bug fixes.

TABLE IV: List of categories of the modifications to fault-triggering tests.

Category	Description	Count	Percentage
Adding New Test	Developers added a new test to reproduce the bug.	189	63%
Adjusting Test Outputs	Developers modified the expected output in tests to cope with source code evolution.	58	20%
Adding New Assertion	Developers added a new assertion to replicate the bug and for regression testing purposes.	41	13%
Improving Test Logic During Bug Fixes	While modifying a test to replicate the bug, developers also enhance the structure or the logic in the test.	7	2%
Others	Developers reformated the style in the test (e.g., indentation), or eliminated code that is not essential to the reproduction of faults.	2	1%

Together with the second author, the two authors revise the list of categories and address any discrepancies.

Phase 2: The two authors independently review the remaining 300 tests based on the discussed categories and assign the test to one of the identified categories from Phase 1.

Phase 3: The two authors independently assigned categories and discussed any disagreements until reaching a consensus. A Cohen's Kappa [13] score of 0.86, calculated before the consensus process, indicates substantial agreement on the initial categorization. Each disagreement was reviewed, with both authors considering the context of code changes and bug reports until a final agreement was achieved.

Results. ***Other than test addition, most changes on the fault-triggering tests are related to improving test oracles or updating tests in response to source code changes.*** In our manual analysis, we uncover five categories of reasons why developers changed the fault-triggering tests as shown in Table IV. Below, we discuss each category in detail.

Adding New Test (189/300, 63%). We found that some tests are specifically created to aid in reproducing a bug. As developers work towards resolving a bug, they may create new tests designed to replicate the problematic behavior. Once the bug is fixed, developers often incorporate these tests into their patch to ensure it does not re-occur in future releases (i.e., regression testing). All test changes from this category belong to Patterns 1 and 2 that we found in RQ1. These tests are often named after the bug report ID to ease traceability and management.

Adjusting Test Outputs (58/300, 19%). We find that developers may change the test code to accommodate the bug-fixing changes in the source code. As shown in Listing 1, developers fixed bug #207 in JacksonCore by modifying the calcHash method in the source code (as shown below). In addition to the bug fix, developers patched the test testSyntheticWithBytesNew responsible for verifying the stability of the hash code, which failed after the new bug fix. One of the developers highlighted in a comment that this bug fix improved "*hashing [with regards to] existing test*". The system is expected to have a different distribution of collision count. Therefore, developers modified the test to match its expected output to the system's actual behavior.

```
// ByteQuadsCanonicalizer.java
public int calcHash(int q1){
   int hash = q1 ^ _seed;
   hash += (hash >>> 16); // to xor hi- and low- 16-bits
   - hash ^= (hash >>> 12); // as well as lowest 2 bytes
   + hash ^= (hash << 3); // shuffle back a bit
   + hash += (hash >>> 12); // and bit more
   return hash;
}
```

```
// TestSymbolTables.java
public void testSyntheticWithBytesNew() throws
     IOException{
   ...
   // anywhere between 70-80% primary matches
   - assertEquals(8524, symbols.primaryCount());
   + assertEquals(8534, symbols.primaryCount());
}
```

Listing 1: Example of adjusting test outputs.

Adding New Assertion (41/300, 14%). During the bug-fixing process, developers may add new assertions to help reproduce a bug (belongs to Pattern 3 from RQ1). Typically, as we found in our manual study, the modification to the test involves adding single-line assertion statements. For example, as seen in the test testCreateNumber shown in Listing 2 (LANG-822), developers added a new assertion to reproduce the bug. The assertion checks whether the method createNumber can execute as expected if the input starts with the string "--". According to the developer's comment, numerous edge cases can expose problematic behaviors when calling the method, so whenever a new edge case arises, it is added to the test as a new assertion statement.

```
public void testCreateNumber() {
...
  // LANG-693
  assertEquals("createNumber(String) LANG-693 failed",
       Double.valueOf(Double.MAX_VALUE),
       NumberUtils.createNumber("" + Double.MAX_VALUE));
+ // LANG-822
+ assertFalse("createNumber(String) LANG-822 succeeded",
     checkCreateNumber("--1.1E-700F"));
}
```

Listing 2: Examples of new assertion.

Improving Test Logic During Bug Fixes (7/300, 2%). Developers may modify tests to change the logic of the tests when trying to fix a bug. Typically, these modifications happen when developers are trying to replicate the bug in a test. Examples of modifications include simplifying complex logic and reorganizing the code structure, which can alter the execution of the test. In Figure 3, we show an example of a test modification in this category based on the JsonAdapterNullSafeTest from issue #800 in GSON. During the bug-fixing process, developers discussed providing a "simpler" test to reproduce the bug. They incorporated new changes that simplified the initialization of the Device class from the test and removed logic that was irrelevant in triggering the fault. It contributed to the improvement of test quality and maintenance.

```
public void testNullSafeBugDeserialize() throws
     Exception {
  - String json =
       "\"\\\"id\\\":\\\"ec57803e2\\\",\\\"category\\\":2\"";
  - Device device = gson.fromJson(json, Device.class);
```

```
+ Device device = gson.fromJson("'id':'ec57803e2'",
      Device.class);
...
@JsonAdapter(Device.JsonAdapterFactory.class)
private static final class Device {
  String id;
  - int category;
  - Device(String id, int category)
  + Device(String id)
    ...
}
```

Listing 3: Example of improved test logic during bug fixes.

Others (2/300, 1%). We find two other reasons why developers may modify the tests, which do not belong to any of the above categories. In particular, we observe that developers may reformat the source code files without necessarily changing any of the logic in the code. For instance, developers removed the extra indentation in the test to improve its readability. We also observe that developers may clean up the test which entails eliminating any obsolete variables or comments.

RQ2-Takeaway. Developers often add new tests (63%) or assertions (14%) to replicate the bug and sometimes improve tests (21%), e.g., to reflect the fix in source code or improve test logic to regression test the fix.

RQ3: How Do Spectrum-Based Fault Localization Approaches Perform in Less Controlled Settings?

Motivation. In the previous RQs, we discovered that many tests are newly added or modified for replicating bugs and regression testing. In many development scenarios, similar information or tests may not be available. Therefore, it is important to understand how fault localization techniques perform when such data is absent. In this RQ, we aim to compare fault localization results between controlled and less-controlled settings. The findings will provide insights into how these tests impact fault localization performance in diverse development scenarios.

Approach. Among existing fault localization techniques, spectrum-based fault localization (SBFL) is one of the most widely studied techniques [3], [24], [31]. SBFL techniques are also an important building block for automated program repair techniques since they rely on SBFL to list potentially buggy locations to start repairing [38]. SBFL techniques differentiate the buggy statements from the non-buggy ones through the program spectrum (code coverage profile). Intuitively, a statement covered by more failed tests but fewer passed tests is more likely to contain the bug. As a result, developer-modified fault-triggering tests can significantly affect SBFL, as these tests may not have existed when the bug was first reported, or may not have been able to trigger the bug (i.e., do not fail) when first introduced.

In particular, we study the effectiveness of SBFL techniques on bugs where fault-triggering tests existed before the bug report but were modified after the bugs were reported. This lets us compare fault localization techniques in controlled settings with those in less controlled, real-world scenarios. The tests in Pattern 3 existed before the bug report and were later modified, often incorporating additional information from the bug-fixing process (RQ2). Therefore, we focus our study on bugs that belong to Pattern 3. We compare the performance of SBFL techniques in two versions: 1) *vReported:* the version when the bug was first reported, and 2) *vD4J:* the version provided by Defects4J, where the tests were modified during bug fixes. Our comparison is conducted on all 155 bugs with fault-triggering tests in Pattern 3. By examining these scenarios, we aim to understand how fault localization techniques perform in less controlled, unclean development environments.

Since Defects4J only provides the code coverage and test execution result for *vD4J*, we need to collect the data for *vReported*. To collect *vReported*, for each bug, we extract the timestamp when the bug report was created and identify the nearest commit *before* the bug was reported. In practice, the bugs remain unfixed in these commits, and if there are any test failures, they are more likely to be related to the bug. We check out these commits on Git, compile the systems, and execute the tests. Note that, if a bug has no failed tests in *vReported*, we exclude the bug from our analysis for a more direct comparison between the two versions. Since most systems do not report code coverage, we manually modified the systems to use JaCoCo to collect the coverage information. We configured JaCoCo and resolved compilation issues, such as missing dependencies and incompatible environment settings. Our data is made publicly available in order to facilitate replication and future research [2].

After collecting the data for *vReported*, we apply SBFL techniques on both *vReported* and *vD4J* and compare the localization result. In particular, we use four following commonly used SBFL techniques [9]: Ochiai [3], Tarantula [19], DStar [60], and Barinel [5]. To evaluate the fault localization techniques, we use the following metrics:

Top-K is defined as the number of faults with at least one faulty statement correctly identified within the first K statements in the ranking. Therefore, a better Top-K result indicates that developers are able to find faulty statements by examining fewer statements. We set K to 1, 3, and 5 in our evaluation.

Mean First Rank (MFR) calculates, for all the bugs, the mean of the first faulty statement in the ranked result. Therefore, a smaller value means that the technique, on average, is able to identify a faulty statement early in the ranked list.

Mean Average Rank (MAR) first calculates the average rank of all the faulty statements for a bug. Then, MAR calculates the average of the ranks from all the bugs. A smaller value means that the faulty statements are ranked earlier.

Results. ***For all the four fault localization techniques that we studied, the localization results degrade significantly on vReported compared to vD4J on bugs in Pattern 3.*** Table V shows the fault localization (FL) results on *vReported* and *vD4J* for the bugs whose fault-triggering tests belong to Pattern 3 (fault-triggering tests exist but were modified after the bug was reported). Out of the 155 unique bugs that we considered, 118 of them caused test failures. The remaining 25 bugs either

TABLE V: Fault localization results for the bugs that belong to Pattern 3 (fault-triggering tests exist but were modified). We compare the results of running the techniques in the commits before the bugs were reported (*vReported*) and in the commits that were provided by Defects4J (*vD4J*).

Technique	Version	Bugs	Top-1	Top-3	Top-5	MFR	MAR
Ochiai	*vReported*	118	3	3	3	2965 (-75.92%)	3254 (-62.69%)
	vD4J	118	49	64	70	714	1214
Tarantula	*vReported*	118	4	4	4	2988 (-76.41%)	3244 (-61.84%)
	vD4J	118	47	62	66	705	1238
D-Star	*vReported*	118	10	11	15	2827 (-90.34%)	3231 (-85.73%)
	vD4J	118	29	39	44	273	461
Barinel	*vReported*	118	3	3	3	2954 (-73.56%)	3282 (-63.19%)
	vD4J	118	45	60	65	781	1208

TABLE VI: Fault localization results for the bugs that belong to Pattern 4 (the fault-triggering tests were not modified). We compare the results of running the techniques in the commits before the bugs were reported (*vReported*) and in the commits that were provided by Defects4J (*vD4J*).

Technique	Version	Bugs	Top-1	Top-3	Top-5	MFR	MAR
Ochiai	*vReported*	105	2	2	6	2097 (-59.23%)	2561 (-50.64%)
	vD4J	105	10	18	26	855	1264
Tarantula	*vReported*	105	2	2	6	2100 (-59.48%)	2587 (-51.53%)
	vD4J	105	12	17	28	851	1254
D-Star	*vReported*	105	2	2	5	2073 (-60.01%)	2553 (-48.73%)
	vD4J	105	10	16	23	829	1309
Barinel	*vReported*	105	2	2	5	2075 (-57.20%)	2564 (-45.83%)
	vD4J	105	9	17	28	888	1389

did not lead to any test failures or their commits before the bug report (*vReported*) were too old and could not be retrieved. Thus, we perform our analysis on 118 bugs. Table V shows that the results in *vReported* are significantly worse than that of *vD4J* for all the metrics. All four FL techniques in *vReported* have much fewer times of a faulty element ranked in Top-1, Top-3, and Top-5 than in *vD4J*. The finding indicates that in the best scenario, 15 of the bugs have faulty statements that are ranked early in the list. In comparison, for *vD4J*, the number of faulty elements ranked in Top-1 and Top 5 is around 40 and 60, respectively. The MFR and MAR results for *vReported* are in the range of 3,000, whereas the results for *vD4J* are in the range of 200 to 1200. In other words, most faulty statements are ranked very low in *vReported* and, the ranking results cannot help developers with locating the bugs. The performance decrease in MRF and MAR values is in the range of 60% to 90%. In short, the findings indicate that, in *vReported*, the fault-triggering tests (excluding Pattern 3 tests) are ineffective in locating the bugs.

As a comparison, we also study the effectiveness of fault localization (FL) techniques on *vReported* and *vD4J* for the bugs that belong to Pattern 4 (the fault-triggering tests were not modified). Unlike the bugs that belong to Pattern 3, Pattern 4 consists of 150 bugs where developers did not modify the fault-triggering tests. This implies that the bugs of Pattern 4 may provide a more representative setting for development and fault localization. We performed a similar analysis on bugs in Pattern 4. 105 out of the 150 unique bugs we analyzed have test failures, while the remaining 45 bugs either have no

TABLE VII: The number of bugs with failed fault-triggering tests on the buggy commit (*vReported*). We consider the bugs whose fault-triggering tests belong to Pattern 3 and Pattern 4 (Table III).

Project	Total bugs	#Bugs with no failed tests		#Bugs with failed tests	
		Pattern 3	Pattern 4	Pattern 3	Pattern 4
Cli	17	0	2	9	6
Closure	93	11	3	32	47
Codec	5	1	0	3	1
Collections	0	0	0	0	0
Compress	14	1	2	8	3
Csv	2	1	0	1	0
Gson	9	1	6	1	1
JacksonCore	7	2	3	1	1
JacksonXml	1	1	0	0	0
JacksonDatabind	17	0	0	10	7
JxPath	1	1	0	0	0
Lang	27	0	0	22	5
Math	36	2	2	30	2
Mockito	31	1	0	2	28
Time	3	0	0	2	1
Total	263	22	18	121	102

test failures occurred or the commits before the bug report (*vReported*) being too old to be retrieved. Surprisingly, the fault localization results for the bugs belonging to Pattern 4 also have worsened. Table VI shows that for Top-1, Top-3, and Top-5, *vReported* ranges from 2 to 6 while *vD4J* ranges from 9 to 28. Similarly, MFR and MAR are much higher for *vReported* (in the range of 2,000), meaning developers must investigate an average of 2,000 to find the faulty statements. The MFR and MAR range from 800 to 1,300 for *vD4J*. The decrease in MRF and MAR performance is within around 45% to 60%.

RQ3-Takeaway 1. The fault localization results on *vReported* (MFR and MAR around 3,000) are significantly worse than those on *vD4J* (MFR and MAR around 700 and 1,200). Our results indicate that fault localization outcomes with Defects4J may differ significantly from those observed in less controlled, real-world development settings.

Discussion. Given that the effectiveness of fault localization techniques can degrade significantly in less controlled settings, we further investigate the possible causes beyond the presence of specific information from the bug-fixing process in the tests. We examine how many bugs with fault-triggering tests from Patterns 3 and 4 actually failed in *vReported* (i.e., the version when the bug was first reported). Note that our study focuses on 263 out of 305 bugs, as some earlier system versions could not be retrieved. Table VII shows the number of bugs with failed fault-triggering tests in *vReported*.

We noticed that in both Pattern 3 and Pattern 4, a small percentage (15%) of fault-triggering tests did not cause any failure in *vReported*, indicating that they could not detect the fault. On the other hand, the remaining fault-triggering tests (85%) failed and assisted in fault localization. This result suggests that while most of these tests failed in *vReported*, they were not as effective in detecting the bugs compared to more

controlled settings. This highlights the need for additional data (e.g., using stack traces or logs from bug reports that contain system execution information [11], [47]) to complement fault localization techniques when such tests are not available.

Interestingly, in Pattern 4, despite not having fault-triggering tests modified by developers during the bug-fixing process, we also observed a decrease in the fault localization result for *vReported*. Through manual analysis, we found that developers might have introduced additional information that affects fault localization results even without direct modifications to the fault-triggering tests. This information can be introduced in various ways, such as modifying functions involved in test execution or adjusting the test setup configuration. For example, in the case of bug Lang 57, developers added a setUp method to initialize all test execution within the test suite LocaleUtilsTest. This new setUp method made the fault-triggering test fail if some variables were not correctly initialized before execution. Although this modification was made after the bug report was submitted and no direct changes were made to the fault-triggering test, this change pattern belongs to Pattern 4. The additional information introduced in the test setup configuration altered test execution and coverage, affecting fault localization results.

Our findings show that additional information from the bug-fixing process may not always be directly incorporated into the fault-triggering tests themselves. Future research may explore bugs from Pattern 4 to investigate and characterize instances where this information is introduced outside of the fault-triggering tests. Nevertheless, our findings reveal that fault-triggering tests in Pattern 4, which were not modified by developers during the bug-fixing process, resulted in better performance for fault localization, as indicated by both MFR and MAR metrics. These bugs are better suited for evaluating fault localization techniques in less controlled, real-world development settings. Future studies may benefit from using bugs/tests from Pattern 4 to assess fault localization techniques in a more development-oriented context.

RQ3-Takeaway 2. Future studies may consider using the fault-triggering tests that were not modified by developers after the bug report to provide a practical setting for evaluating fault localization techniques.

IV. Discussion and Future Work

In this paper, we studied the bugs and their corresponding fault-triggering tests in Defects4J. We classified the bugs based on whether their tests contained additional information from the bug-fixing process and the code coverage data on the buggy commit (i.e., the commit prior to the bug report creation). We made the dataset publicly available [2], and we believe that it can inspire future research. Below, we discuss the implications of our findings and potential future research directions.

Future research may use our annotated data to re-evaluate fault localization and automated program repair techniques. We find that a majority of the fault-triggering tests in Defects4J contain information from the bug-fixing process, which can affect the fault localization results. Our findings provide additional insights into the effectiveness of fault localization techniques in development settings where these techniques are used in practice (e.g., analyzing the failing tests associated with a reported bug). Our dataset also annotates the fault-triggering tests based on whether they are influenced by the bug-fixing process. We believe the dataset can be used in three valuable directions. First, future studies can leverage the dataset to re-evaluate the effectiveness of fault localization or automated program repair techniques using fault-triggering tests that are not influenced by the bug-fixing process. Second, future research can use our dataset to conduct separate evaluations of the bugs in Defects4J, considering those with and without influence from the bug-fixing process. Third, future studies may use our dataset to investigate the characteristics of fault-triggering tests that require less maintenance during bug fixes, such as tests that involve minimal or no code changes (e.g., tests in Pattern 4). We believe these insights complement the strengths of Defects4J and can enhance its utility for simulating real-world development scenarios.

Future studies may need to consider developers' bug-fixing process when creating benchmarks. In RQ2, we discovered that developers might not have fault-triggering tests at the time of receiving a bug report, often creating them during the bug-fixing process. This underscores the need for empirical studies to understand better these activities and their implications for benchmark dataset creation. Our dataset offers a starting point for investigating the bug-fixing process, suggesting expansion to other systems and contexts. Our findings are particularly relevant in environments where testing drives bug diagnosis and fixing. For projects emphasizing automated testing and test modifications post-fixing, our insights can aid in developing and evaluating benchmark datasets.

There is a need for more comprehensive and diverse benchmarks. To help facilitate software testing research, several bug benchmarks have been proposed, with Defects4J [20] being the most popular and widely used. Defects4J provides a clean, well-organized dataset that is easy to use and has significantly advanced the field. However, many existing benchmarks assume that fault-triggering tests are available at the time of bug discovery, whereas in real-world scenarios, tests are often created or modified as part of the debugging process. Our study highlights that some fault-triggering tests in Defects4J may include information from subsequent commits, potentially influencing fault localization results. This insight offers an additional perspective that may be useful in real-world development scenarios.

Other datasets like Bugs.jar [49] and BEARS [42] may exhibit similar characteristics. For example, Bugs.jar also isolates bugs and tests based on later commits. In our initial examination, some fault-triggering tests in BEARS were also derived from later commits. These datasets share similarities with Defects4J and may be influenced by the bug-fixing pro-

cess. We acknowledge the challenges of curating benchmarks and the inherent biases in existing datasets. A key question arises: how effective are fault localization and automated program repair techniques that assume pre-existing tests, given that many tests are written after a bug is reported? Future research should explore how these methods perform in real-world scenarios where tests are unavailable at bug discovery and whether alternative sources, such as bug reports or logs, can aid fault localization.

Our findings serve as a reminder and guide for future researchers or practitioners. We urge the community to consider the assumptions and constraints under which these datasets were created and to exercise caution when generalizing findings. There is an opportunity to develop more comprehensive benchmarks. For example, a recent study [12] compiles consecutive commits across systems to gather code coverage, offering a more representative basis for assessing fault localization and repair techniques. This underscores the need for improved benchmarks in software testing research to complement datasets like Defects4J.

V. Related Work

Empirical studies on Defects4J. The popularity of automated debugging has made the Defects4J benchmark essential for evaluating research approaches, as it enables a controlled evaluation environment. However, few studies [23], [26], [40] have thoroughly examined how subsequent test modifications within this benchmark impact its evaluations. Liu et al. [40] found that certain fault-triggering tests might be derived from versions of the system where the bug had already been resolved, potentially aiding fault localization techniques. Kabadi et al. [22] highlighted the importance of distinguishing between tests added or modified after the bug was fixed and those that existed before, as tests added or modified after the fix may inadvertently offer extra assistance to automated program repair techniques due to their specificity. Koyuncu et al. [26] noted that tests modified during bug fixes could influence evaluations but lacked a systematic study on their prevalence and effects on fault localization. Just et al. [21] also discussed some limitations of fault-triggering tests in Defects4J, noting that they may not fully reflect real-world scenarios. In our study, we conducted a comprehensive analysis of the timeline and evolution of fault-triggering tests, indicating that these tests may undergo changes throughout the entire bug resolution process. We also conducted a manual study on how developers modified these tests and evaluated their impact on the performance of state-of-the-art spectrum-based fault localization techniques, providing insights into their impact on fault localization in less-controlled settings.

Bug benchmarks. Several bug benchmarks have been proposed by the automated testing research community to support empirical evaluations. Besides Defects4J, two other popular Java benchmarks are Bugs.jar [49] and BEARS [42]. Both of these benchmarks follow the same bug replication process as Defects4J, where fault-triggering tests may be committed to system versions before bug fixes. However, through our initial investigation, some of these tests were added after the bug report or during the bug fix, which means they may also suffer from a similar issue as Defects4J. Kang et al. [23] highlighted this issue and proposed LIBRO, a Large Language Model-based framework that generates fault-triggering tests directly from bug reports, supporting automated debugging techniques in the absence of pre-existing tests. Future research should analyze fault-triggering test timelines in benchmarks and their effects on research methods.

Fault Localization and Program Repair Techniques. Due to the importance and high cost of bug-fixing activities, a large number of research studies focus on fault localization (FL) [7], [32], [62], [69] and automated program repair (APR) [33], [34], [50]. The Defects4J dataset has been used extensively as a benchmark in the evaluation of FL and APR techniques.

Fault Localization (FL). Since its release in 2014, Defects4J has become a benchmark for evaluating FL techniques. Learning-to-Rank (LtR) methods have advanced FL by using suspiciousness scores from various techniques to rank potential faults [7], [32], [54], [62]. For instance, MULTRIC [62] and others integrate spectrum-based fault localization (SBFL) scores with additional metrics, such as program invariants [7] and code complexity [54]. Recent efforts also apply deep learning to interpret code coverage data, generating suspiciousness scores [31], [41], [67], [68], such as GRACE [41], which uses a graph representation merging method. Our study utilizes Defects4J's test cases and source code to examine the impact of developer knowledge on SBFL performance, providing insights for refining coverage-based FL techniques.

Automated Program Repair (APR). Defects4J is widely used for evaluating APR techniques [18], [38], [39], [58]. A common evaluation approach is verifying that candidate patches make the program pass all test cases. Our work investigates the prevalence of information from subsequent commits in tests, which may impact prior APR research on Defects4J. Fault localization, crucial in APR, reduces the search space for generating patches. Fault-triggering tests play a key role in identifying code to repair. Some studies [61], [64] use test cases to eliminate overfitted patches and improve patch quality, which may also be influenced by subsequent test modifications.

VI. Threats to Validity

Internal Validity. Threats to internal validity are related to analysis errors or biases. One main threat comes from the human analysis in our study. We manually analyzed and categorized the modifications on fault-triggering tests in RQ2. However, we adopted the manual analysis approaches used in prior studies [15], [28], [35], [36] to mitigate subjectivity. Three phases were used and two authors independently were involved in the analysis. The analysis results achieve a Cohen's Kappa of 0.86, which indicates a substantial level of agreement [13]. In RQ1, we also manually reviewed the relevance of the test modification to a bug. However, the main work in extracting events about bug resolution was automatic and we

only manually checked whether there were any errors in the generated results.

External Validity. Threats to external validity concern our findings' generalizability. Though focused on Defects4J, different datasets may lead to varying outcomes. Given Defects4J's prominence in fault localization and automated program repair (APR) research, especially for Java programs, its characteristics might influence research results and methodologies. However, our insights on developer knowledge in fault-triggering tests and test modifications could apply to other datasets, particularly those with similar development practices or curation processes. While our study is based on Defects4J, the themes we identified, like the effect of developer knowledge, might be common in other datasets. Researchers should thus consider the potential biases highlighted by our study when exploring similar datasets.

VII. Conclusion

A significant amount of research has been conducted in fault localization and automated program repair to provide developers with solutions for automated debugging. Benchmarks like Defects4J are crucial for assessing these techniques, offering real bugs, fault-triggering tests, and fixes in a controlled environment. However, our findings show that most fault-triggering tests (77%) were modified or added after the bugs were reported or fixed, highlighting other perspectives of software development scenarios that may occur in less-controlled settings. Our experiments demonstrated that spectrum-based fault localization techniques perform significantly worse in these less-controlled environments. We categorized these modifications to provide insights into developers' bug-fixing processes and emphasize the need to evaluate fault localization techniques on datasets that reflect more variable development scenarios. Our publicly available dataset supports future research in these less-controlled environments.

References

[1] The Defects4J dataset version 2.0.0. https://github.com/rjust/defects4j, 2022.

[2] Public repository of our work. https://github.com/nakhlarafi/Studying-Data-Cleanness-in-Defects4J, 2024.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[4] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007.

[5] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, 2009.

[6] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. Fonte: Finding bug inducing commits from failures. *arXiv preprint arXiv:2212.06376*, 2022.

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

[8] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 229, 2013.

[9] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, page 378–381, 2012.

[10] An Ran Chen, Tse-Hsun Chen, and Junjie Chen. How useful is code change information for fault localization in continuous integration? In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[11] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. Demystifying the challenges and benefits of analyzing user-reported logs in bug reports. *Empirical Software Engineering*, 26(1):1–30, 2021.

[12] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. T-evos: A large-scale longitudinal study on CI test execution and failure. *IEEE Transactions on Software Engineering*, 2022.

[13] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[14] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor S Trivedi. Fault triggers in open-source software: An experience report. In *2013 IEEE 24th International symposium on software reliability engineering (ISSRE)*, pages 178–187. IEEE, 2013.

[15] Zishuo Ding, Jinfu Chen, and Weiyi Shang. Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet? In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1435–1446. IEEE, 2020.

[16] Gregory Gay and René Just. Defects4J as a challenge case for the search-based software engineering community. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*, pages 255–261. Springer, 2020.

[17] Bo Jiang, Zhenyu Zhang, Tsun-Him Tse, and Tsong Yueh Chen. How well do test case prioritization techniques support statistical fault localization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 99–106. IEEE, 2009.

[18] Jiajun Jiang, Yingfei Xiong, and Xin Xia. A manual inspection of Defects4J bugs and its implications for automatic program repair. *Science china information sciences*, 62:1–16, 2019.

[19] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 273–282, 2005.

[20] René Just, Darioush Jalali, and Michael D Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[21] René Just, Chris Parnin, Ian Drosos, and Michael D Ernst. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 287–297, 2018.

[22] Vinay Kabadi, Dezhen Kong, Siyu Xie, Lingfeng Bao, Gede Artha Azriadi Prana, Tien-Duy B. Le, Xuan-Bach D. Le, and David Lo. The future can't help fix the past: Assessing program repair in the wild. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 50–61, 2023.

[23] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323, 2023.

[24] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125. IEEE, 2017.

[25] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.

[26] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 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*, pages 314–325, 2019.

[27] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 821–830, 2017.

[28] Maxime Lamothe and Weiyi Shang. When apis are intentionally bypassed: An exploratory study of api workarounds. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 912–924. IEEE, 2020.

[29] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[30] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015.

[31] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180, 2019.

[32] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[33] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 602–614, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Yi Li, Shaohua Wang, and Tien N Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 511–523, 2022.

[35] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. Where shall we log? studying and suggesting logging locations in code blocks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 361–372, 2020.

[36] Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang, and Weiyi Shang. Dlfinder: characterizing and detecting duplicate logging code smells. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 152–163. IEEE, 2019.

[37] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56, 2017.

[38] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pages 102–113. IEEE, 2019.

[39] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 31–42. ACM, 2019.

[40] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021.

[41] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021.

[42] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478. IEEE, 2019.

[43] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[44] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? In *Proceedings of the 40th International Conference on Software Engineering*, pages 25–25, 2018.

[45] Jerzy Neyman. On the two different aspects of the representative method: the method of stratified sampling and the method of purposive selection. In *Breakthroughs in statistics*, pages 123–150. Springer, 1992.

[46] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[47] Lorena Barreto Simedo Pacheco, An Ran Chen, Jinqiu Yang, et al. Leveraging stack traces for spectrum-based fault localization in the absence of failing tests. *arXiv preprint arXiv:2405.00565*, 2024.

[48] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[49] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*, pages 10–13, 2018.

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

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

[52] Jeongju Sohn, Yasutaka Kamei, Shane McIntosh, and Shin Yoo. Leveraging fault localisation to enhance defect prediction. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–294. IEEE, 2021.

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

[54] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*, pages 273–283. ACM, 2017.

[55] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349. IEEE, 2019.

[56] Shaowei Wang and David Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[57] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering*, 47(11):2348–2368, 2019.

[58] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering*, pages 1–11, 2018.

[59] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 326–337, 2019.

[60] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63:290–308, 2014.

[61] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[62] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 191–200. IEEE, 2014.

[63] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F Bissyandé. Where were the repair ingredients for Defects4J bugs? exploring the impact of repair ingredient retrieval on the performance of 24 program repair systems. *Empirical Software Engineering*, 26:1–33, 2021.

[64] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 831–841, 2017.

[65] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24:33–67, 2019.

[66] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, pages 261–272, 2017.

[67] Z. Zhang, Y. Lei, X. Mao, and P. Li. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pages 445–455, 2019.

[68] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. Cnn-fl: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455. IEEE, 2019.

[69] Wei Zheng, Desheng Hu, and Jing Wang. Fault localization analysis based on deep neural network. *Mathematical Problems in Engineering*, 2016, 2016.

[70] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2019.