Comparing Developer-Provided to User-Provided Tests for Fault Localization and Automated Program Repair

René Just
University of Massachusetts
Amherst, MA, USA
rjust@cs.umass.edu

Chris Parnin
NC State University
Raleigh, NC, USA
cjparnin@ncsu.edu

Ian Drosos
University of California
San Diego, CA, USA
idrosos@ucsd.edu

Michael D. Ernst
University of Washington
Seattle, WA, USA
mernst@cs.washington.edu

ABSTRACT

To realistically evaluate a software testing or debugging technique, it must be run on defects and tests that are characteristic of those a developer would encounter in practice. For example, to determine the utility of a fault localization or automated program repair technique, it could be run on real defects from a bug tracking system, using real tests that are committed to the version control repository along with the fixes. Although such a methodology uses real tests, it may not use tests that are characteristic of the information a developer or tool would have in practice. The tests that a developer commits *after* fixing a defect may encode more information than was available to the developer when initially diagnosing the defect.

This paper compares, both quantitatively and qualitatively, the developer-provided tests committed along with fixes (as found in the version control repository) versus the user-provided tests extracted from bug reports (as found in the issue tracker). It provides evidence that developer-provided tests are more targeted toward the defect and encode more information than user-provided tests. For fault localization, developer-provided tests overestimate a technique's ability to rank a defective statement in the list of the top-n most suspicious statements. For automated program repair, developer-provided tests overestimate a technique's ability to (efficiently) generate correct patches—user-provided tests lead to fewer correct patches and increased repair time. This paper also provides suggestions for improving the design and evaluation of fault localization and automated program repair techniques.

CCS CONCEPTS

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

KEYWORDS

Fault localization, Automated program repair, Test effectiveness

ACM Reference Format:
René Just, Chris Parnin, Ian Drosos, and Michael D. Ernst. 2018. Comparing Developer-Provided to User-Provided Tests for Fault Localization and Automated Program Repair. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, Netherlands, July 16–21, 2018 (ISSTA'18),* 11 pages.
https://doi.org/10.1145/3213846.3213870

1 INTRODUCTION

Researchers have created dozens of fault localization (FL) techniques [36] and automated program repair (APR) techniques [25]. These techniques represent promising progress toward the long-term goal of automatically locating and fixing defects in software programs. Although these techniques take diverse approaches, their inputs are generally the same: a defective program and a set of tests—at least one of which is a *triggering test* (i.e., a test that fails on the defective program but passes when the defect is fixed).

A FL or APR technique is valuable if it works well when using real defects and triggering tests. Many older empirical evaluations of FL techniques used artificial defects and/or artificial triggering tests. The most-used dataset is the Siemens suite [12], which contains artificial defects for 7 small programs (136–456 lines of code). The corresponding test suites for these programs contain 1052–5542 tests each and were created by researchers to satisfy unrealistically strong adequacy criteria. More recently, researchers have begun to perform more realistic FL and APR experiments, using datasets of real defects and tests derived from version control history (e.g., Defects4J [16] or ManyBugs [9]). Defects4J, the dataset used in this paper, contains a *developer-provided* triggering test for each defect, derived from a version control system commit that is linked to a bug report in an issue tracker.

The starting point for fixing a reported bug is the bug report, which may or may not include a *user-provided* triggering test. A developer acquires a deeper understanding of a bug while reproducing, localizing, and fixing it. Finally, the developer commits a bug fix and triggering tests that encode the developer's knowledge. Relative to the original bug report, these developer-provided triggering tests may be more extensive, more focused, or more likely to test the root cause. In practice, a FL or APR tool will be run on user-provided triggering tests that appear in bug reports. Developer-provided triggering tests may reduce the search space for the bug, and hence evaluating a FL or APR tool on developer-provided triggering tests may yield inaccurate results.

Previous evaluations (cf., [25, 36]) using datasets constructed from version control history implicitly assumed that developer-provided triggering tests are characteristic of user-provided triggering tests from bug reports, which are available before the defect is localized and fixed. However, it is possible that an evaluation of FL and APR techniques on user-provided triggering tests would yield different outcomes than previous evaluations. If so, previous rankings and absolute performance results of FL and APR techniques would need to be revised, and practitioners and researchers should choose different techniques to use and improve. It is also possible that an evaluation of FL and APR techniques on user-provided triggering tests would yield the same outcomes, thus resolving any uncertainty. Either result is of significant scientific interest. As Rizzi et al. note in their paper on Klee [32], unexamined assumptions can result in questionable research claims and wasted research effort.

This paper seeks to quantitatively and qualitatively compare triggering tests in bug reports against those that developers commit together with bug fixes. Specifically, it considers two types of triggering tests for the Defects4J dataset: (1) *user-provided triggering tests* reflecting knowledge contained in the initial bug reports, and (2) *developer-provided triggering tests* possibly written after the fix and obtained from the version control system. This paper further investigates the effect of the type of triggering test on the effectiveness of FL and APR techniques. The experiments are based on six previously studied FL techniques and two previously studied APR techniques.

This paper answers the following four high-level questions:

(1) Do user-provided and developer-provided triggering tests differ in terms of size, code coverage, and assertion strength?
(2) Does the type of the triggering test affect the performance of automated fault localization techniques?
(3) Does the type of the triggering test affect the performance of automated program repair techniques?
(4) Does test separation (i.e., creating a new, separate triggering test vs. augmenting an existing test) affect the performance of automated fault localization and program repair techniques?

This paper's main conclusions are as follows:

(1) Developers adopted only 20% of user-provided triggering tests as submitted in bug reports; usually, developers commit a more specific triggering test along with the bug fix. As a result, developer-provided triggering tests differ significantly from user-provided triggering tests in terms of size, code coverage, and assertion strength.
(2) Developer-provided triggering tests overestimate absolute FL performance, in particular the ability to rank a faulty statement in the list of the top-n most suspicious statements. Differences in FL performance between techniques are insignificant for developer-provided and user-provided triggering tests.
(3) Developer-provided triggering tests overestimate the effectiveness of APR techniques: user-provided triggering tests lead to fewer correct patches and increased repair time.
(4) Developers merged the defect-triggering functionality into an existing test for 22% of the defects. Test separation improves FL performance for these defects.

This paper's contributions and organization are as follows:

- A publicly available set of 100 user-provided triggering tests for the Defects4J dataset (section 3).
- A quantitative and qualitative comparison of the characteristics of user-provided and developer-provided triggering tests (section 4).
- An empirical study on the effect of the type of triggering test on automated fault localization (section 5.1).
- An empirical study on the effect of the type of triggering test on automated program repair (section 5.2).
- An empirical study on the effect of test separation on automated fault localization and program repair (section 5.3).
- A discussion of implications and an outline of possible research directions (section 6).

```
@Test
public void userTest() throws Exception {
    assertEquals("\uD83D\uDE30", StringEscapeUtils.escapeCsv("\uD83D\uDE30"));
}
```

(a) User-provided triggering test, extracted from the bug report.

```
249 @Test
250 public void testLang857() throws Exception {
251     assertEquals("\uD83D\uDE30", StringEscapeUtils.escapeCsv("\uD83D\uDE30"));
252     // Examples from https://en.wikipedia.org/wiki/UTF-16
253     assertEquals("\uD800\uDC00", StringEscapeUtils.escapeCsv("\uD800\uDC00"));
254     assertEquals("\uD834\uDD1E", StringEscapeUtils.escapeCsv("\uD834\uDD1E"));
255     assertEquals("\uDBFF\uDFFD", StringEscapeUtils.escapeCsv("\uDBFF\uDFFD"));
256 }
```

(b) Developer-provided triggering test.

```
466 public abstract class CharSequenceTranslator {
 :
 :
            for (int pt = 0; pt < consumed; pt++) {
476 -           pos += Character.charCount(Character.codePointAt(input, pt));
477 +           pos += Character.charCount(Character.codePointAt(input, pos));
478         }
479     }
480 }
```

(c) The committed bug fix.

Figure 1: Triggering tests and bug fix for the Lang-6 defect in Defects4J. The developer adopted the single failing input provided by the user on line 251, but added three additional calls to `escapeCsv`.

2 MOTIVATING EXAMPLE

To illustrate the differences between user-provided and developer-provided triggering tests, consider the Lang-6[1] defect from the Defects4J dataset. A user reported the following issue:

I found that there is bad surrogate pair handling in the CharSequenceTranslator. This is a simple test case for this problem. \uD83D\uDE30 is a surrogate pair.

The user also provided a triggering test (fig. 1a), explained that this test produces a `StringIndexOutOfBoundsException`, and attached a patch that fixes the bug. A developer addressed this issue and committed a bug fix (fig. 1c) along with a different triggering test (fig. 1b), which contains four calls to `escapeCsv`. This test includes the one test input provided by the user, plus three additional inputs.

In this case, the user-provided test differs from the final test that the developer committed after fixing the bug: the user-provided test is a single example, whereas the developer-provided test is more comprehensive. These differences may have a significant effect on the absolute and relative performance of FL or APR techniques.

3 SUBJECTS

Our study employs the Defects4J dataset, version 1.1.0. Defects4J contains 395 defects, each with a developer commit that fixes it and at least one triggering test that fails before the fix but passes after the fix. We omitted one of Defects4J's projects, Chart, because it contains too few bugs with issue tracker entries to be used in section 3.2; this left 5 projects and 369 bugs, summarized in table 1.

3.1 Linking Defects to Bug Reports

Each bug-fixing commit in Defects4J references an issue-tracker ID, which corresponds to a closed issue that is labeled as a bug. For each defect in Defects4J, we automatically extracted the referenced issue-tracker ID from the commit log and built a mapping from Defects4J's bug ID to the corresponding issue-tracker ID and URL.

[1]Issue tracker entry: https://issues.apache.org/jira/browse/LANG-857

Table 1: Subjects from the Defects4J dataset.
The lines of code (LOC), number of JUnit tests, and number of assertions in JUnit tests, reported for the most recent project version in Defects4J. All LOC counts in the paper are non-comment, non-blank lines, measured with sloccount (http://www.dwheeler.com/sloccount).

Project	Code LOC	Test LOC	`@Test`	`assert`
Closure	91K	85K	7,929	8,936
Lang	22K	38K	2,242	13,117
Math	84K	86K	3,581	9,512
Mockito	11K	20K	1,457	1,882
Time	28K	53K	4,132	17,658
Total	236K	282K	19,341	51,105

3.2 Extracting User-Provided Triggering Tests

For 100 defects in the Defects4J dataset, we manually extracted user-provided triggering tests from the issue trackers. In some cases the bug report already contained an executable test case. In other cases, we added scaffolding to code found within the bug report, such as a method body, `import` statements, variable declarations, and `@Test` annotations, or we transformed `printf`/`println` statements and stated assertions about the output into `assert` statements. In yet other cases, we elaborated an English description into a test case. The result is a single, triggering JUnit test case, which fails on the buggy version and passes on the fixed version of the defect.

For each project in Defects4J, we arbitrarily selected defects and examined the corresponding bug reports until we had extracted 20 tests. We examined 118 bug reports and discarded 18, or 15%. In 12 cases, the user provided no test or the user-provided test did not fail, but nonetheless the developers accepted the bug report and committed a fix. In 6 cases, the user provided a test that failed and the developers committed a fix, but the user-provided test continued to fail—in other words, the developer did not fix the user's test. An example of the latter case is Mockito-29[2]. The fix-commit log describes a fix (*Fix for issue 229 in the describeTo phase of the Same matcher*), but the commit does not pass the user-provided test.

When a developer fixes a bug or runs a FL or APR tool, a first step is to incorporate the user-provided test into the project's test suite. This enables the developer to reproduce and investigate the bug, and to ensure that no regression bugs are introduced. We performed this step manually, integrating the user-provided test into the pre-fix version of the developer's test suite at the same location the developer modified the test suite in the bug-fixing commit. This means that if the developer added a new triggering test in the commit, we added the user-provided test as a separate test; if the developer merged the defect-triggering functionality into an existing test, we merged the user-provided test into the same test at the same location. This enables a fair comparison between (1) the information the developer had available before fixing the bug, which is the previous test suite plus the user's test, and (2) the final developer-provided test suite, which might incorporate knowledge the developer obtained during the bug-fixing process.

An automated step verified that each extracted user-provided test triggers the defect on the buggy version and passes on the fixed version—in isolation and integrated into the pre-fix test suite.

[2]Issue tracker entry: https://code.google.com/archive/p/mockito/issues/229

Table 2: Summary statistics about triggering tests.
The number of assertions is underapproximated for developer-provided tests—each test harness method, which may contain multiple assert statements, counts as a single assertion.

Project	Developer-provided				User-provided			
	min	max	mean	median	min	max	mean	median
Number of tests								
Closure	1	8	2.6	1	1	1	1.0	1
Lang	1	2	1.4	1	1	1	1.0	1
Math	1	28	3.0	1	1	1	1.0	1
Mockito	1	7	2.2	1.5	1	1	1.0	1
Time	1	8	2.5	1	1	1	1.0	1
Number of assertions								
Closure	1	7	2.2	2	1	2	1.5	1
Lang	0	26	4.1	2	1	8	1.6	1
Math	0	4	1.5	1	0	4	1.3	1
Mockito	0	3	1.3	1	0	5	1.3	1
Time	0	17	6.2	5	0	5	1.4	1
Test LOC								
Closure	3	61	16.9	11	7	25	14.9	14
Lang	3	40	14.7	10	3	27	11.1	10
Math	3	27	12.4	12	3	45	10.7	7
Mockito	3	40	11.1	8	4	30	13.7	12
Time	6	48	24.0	24.5	3	75	17.7	11.5

4 QUANTITATIVE & QUALITATIVE ANALYSIS

Our overall goal is to compare defect-triggering tests provided by a user (submitted along with a reported issue) and defect-triggering tests provided by a developer (committed to the version control system along with a fix). This section reports on our quantitative and qualitative analysis of these two types of defect-triggering tests.

4.1 Quantitative Analysis

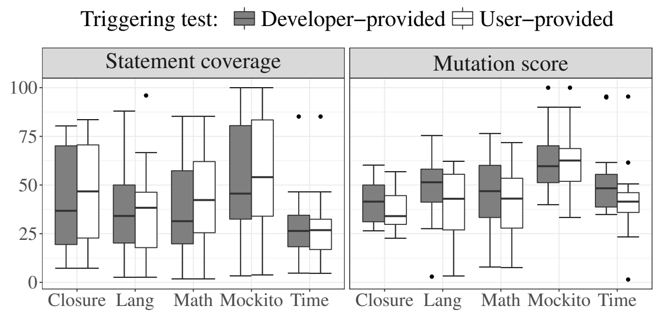
Methodology For 100 defects, we computed the following test characteristics for the user-provided and developer-provided tests:

Test size: We measured lines of code, number of assertions (`assert`, `assertEquals`, etc.), and number of test cases.

Code coverage: We measured statement coverage on the class(es) changed by the bug fix (*modified* classes). We measured coverage on the buggy program version, as FL and APR techniques do. Coverage measurements on the modified classes indicate how targeted a triggering test is—how comprehensively it covers the defective class.

Mutation score: We measured the mutation score on the patched classes of the fixed program version, using Major [15] v1.3.2 with its default settings. Computing the mutation score for all classes would be computationally expensive, and the average mutation score across many classes would wash out differences on the patched classes. Specifically, we divided the number of mutants detected by a triggering test by the number of mutants covered by that test, which indicates the strength of the triggering test's assertions [10, 18].

We performed a paired t-test (paired over the set of defects) and computed the Cohen's d effect size for each measure and project. We chose parametric statistical measures for statistical power, given a relatively small sample size of 20 defects per project, and because we did not observe serious violations of normality.



(a) Statement coverage ratios and mutation scores for the developer-provided and user-provided tests.

Ratio	Closure	Lang	Math	Mockito	Time
Coverage	—	—	0.43*	0.39*	—
Mutation	0.41*	0.51**	0.65***	—	0.44*

(b) Summary of Figure 2a showing which differences are statistically and practically significant. The numbers are Cohen's d effect size (small: < .5, medium: < .8, or large: >= .8), and the asterisks indicate statistical significance (*: $p < .1$; **: $p < .05$; ***: $p < .01$). Entries with a dash indicate a statistically insignificant difference.

Figure 2: Differences in code coverage ratios and mutation scores between developer-provided and user-provided tests.

Results On average, developer-provided tests contain more lines of code and assertions for all projects, except Mockito. Table 2 summarizes test quantity, size, and strength (assertions). All defects have only one user-provided triggering test, and the majority of defects have one developer-provided triggering test.

Figure 2 shows both code coverage and mutation scores. Most user-provided tests are less targeted to the defect: they cover more code in the defective class, even though they are generally smaller (table 2). Most user-provided tests have weaker assertions: they have a lower mutation score. The differences are consistent, but not statistically significant for each individual project.

4.2 Qualitative Analysis

We performed a qualitative analysis to characterize the developer-provided and user-provided tests, and their relationship.

Methodology We manually analyzed all triggering tests and corresponding issue-tracker entries. We determined the following characteristics, which are summarized in Table 3:

- *Project:* What project does the issue report pertain to?
- *Reproducibility:* Can we reproduce the submitted issue and create a triggering test from the bug report using the methodology of section 3.2? We may not be able to reproduce an issue because the submitter provided no test, the user-provided test passes on the buggy version, or the user-provided test fails on the fixed version.
- *Clarity:* Is the issue clearly and unambiguously described? Possible values are "Low" (a vague verbal description of the problem, without inputs or outputs), "Medium" (a description of output, such as a stack trace, but no inputs), or "High" (a complete test case with inputs and outputs). This indicates the conceptual difficulty of creating a triggering test.
- *Executability:* Does the triggering test execute or does it need further work to turn into a working test case? This indicates the mechanical, non-conceptual effort of creating a triggering test. Possible values are "No" (no source code provided), "Partial" (source code fragment that might be missing imports, method declarations, etc.), or "Full" (copy-pasteable source code).
- *Adoption (of instructions):* This compares the instructions (non-`assert` statements) in the user-provided test to the instructions in the developer-provided test. The developer-provided test might be completely unrelated to the user-provided one, it might have the same instructions, or it might be a variant of the user-provided test. For the latter, it might contain additional instructions, omit some, or both.
- *Adoption (of assertions)* This compares the assertions in the user-provided test to the assertions in the developer-provided test. The possible values are the same as for *Adoption (of instructions)*.
- *New Test:* Did the developer create a new test or merge the defect-triggering functionality into an existing test?
- *Submitter:* Is the issue submitter a user or a developer of the project? We considered anyone with at least five commits to the code base to be a developer.
- *Patch:* Did the report include a proposed fix for the issue?

Table 3: Characteristics of the user-provided tests.

Characteristic	Possible values
Project	Closure, Lang, Math, Mockito, Time
Reproducibility	Yes, No Test, Passes Buggy, Fails Fixed
Clarity	Low, Medium, High
Executability	No, Partial, Full
Adoption	No, As Is, Minimized, Augmented
New Test	New, Existing, Both
Submitter	User, Developer
Patch	No, Yes

Results Table 4 summarizes the characteristics. The majority of the reported issues had high clarity, and 87% of the issues included a partially or fully executable test case. For 78% of the issues, a developer added a new triggering test, but only 20% of the user-provided tests were adopted as is. Figures 1 and 3 give concrete examples for each category of adoption:

- *No* (fig. 3a): The developer implemented a more targeted test, which uses different inputs, and added assertions.
- *As is* (fig. 3b): The user-provided test throws an unexpected exception. The developer adopted this concise test as is.
- *Minimization* (fig. 3c): The `OpenMapRealMatrix` instantiation should cause an integer overflow exception. The developer minimized this test by removing five out of six lines and declaring the expected exception. There is no need for the additional code, which should be unreachable.
- *Augmentation* (fig. 1): The motivating example in section 2 shows an augmented user-provided test.

Table 4: Summary of qualitative characteristics per project and submitter type (user vs. developer).

Project	Tests	Clarity			Executability			Adoption								New test			Patch?		Submitter	
								Instructions				Assertions										
		Lo	Med	Hi	No	Part	Full	No	As is	Min	Aug	No	As is	Min	Aug	New	Exist	Both	No	Yes	User	Dev
Closure	20	0	0	20	0	11	9	4	2	5	9	3	4	0	13	13	3	4	20	0	20	0
Lang	20	0	5	15	2	14	4	2	3	1	14	2	4	1	13	14	4	2	14	6	13	7
Math	20	0	2	18	3	11	6	1	8	5	6	3	5	2	10	15	4	1	12	8	19	1
Mockito	20	5	6	9	4	14	2	5	7	3	5	7	8	2	3	18	1	1	16	4	17	3
Time	20	4	0	16	4	11	5	7	0	3	10	6	2	1	11	18	2	0	18	2	19	1
Total	100	9	13	78	13	61	26	19	20	17	44	21	23	6	50	78	14	8	80	20	88	12

Submitter	Tests	Clarity			Executability			Adoption								New test			Patch?	
								Instructions				Assertions								
		Lo	Med	Hi	No	Part	Full	No	As is	Min	Aug	No	As is	Min	Aug	New	Exist	Both	No	Yes
User	88	6%	10%	84%	9%	63%	28%	19%	19%	18%	43%	22%	24%	6%	49%	78%	13%	9%	80%	20%
Dev	12	33%	33%	33%	42%	50%	8%	17%	25%	8%	50%	17%	17%	8%	58%	75%	25%	0%	83%	17%

```
- CharSequence cs1 = "1 < 2";
- CharSequence cs2 = CharBuffer.wrap("1 < 2".toCharArray());
- System.out.println(StringEscapeUtils.ESCAPE_HTML4.translate(cs1));
- System.out.println(StringEscapeUtils.ESCAPE_HTML4.translate(cs2));
+ final LookupTranslator lt = new LookupTranslator(
+     new CharSequence[][] { { new StringBuffer("one"),
+     new StringBuffer("two")}});
+ final StringWriter out = new StringWriter();
+ final int result = lt.translate(
+     new StringBuffer("one"), 0, out);
+ assertEquals("Incorrect codepoint consumption", 3, result);
+ assertEquals("Incorrect value", "two", out.toString());
```

(a) No adoption (Lang-4)

```
StrBuilder sb = new StrBuilder(
    "\n%BLAH%\nDo more stuff\neven more stuff\n%BLAH%\n");
sb.deleteAll("\n%BLAH%");
assertEquals("\nDo more stuff\neven more stuff\n", sb.toString());
```

(b) Adopted as is (Lang-61)

```
-   OpenMapRealMatrix m =
    new OpenMapRealMatrix(3, Integer.MAX_VALUE);
-   m.setEntry(0, 0, 2);
-   m.setEntry(2, 2, 3);
-   //Should print "2.0", but instead it prints "3.0"
-   System.out.println(m.getEntry(0, 0));
```

(c) Adopted after minimization (Math-45)

Figure 3: Examples for test adoption.
Differences between user-provided and developer-provided tests are shown in unified diff format. Method signatures are omitted, and the given defect IDs (e.g., Lang-4) refer to Defects4J defects.

5 EFFECT ON AUTOMATED DEBUGGING

We conducted an experiment using six fault localization and two automated program repair techniques to investigate whether the observed differences between user-provided and developer-provided triggering tests affect their accuracy. Recall from section 3.2 that we obtained the non-triggering tests and the developer-provided triggering tests from the version control repository, and we obtained the user-provided triggering tests from the issue tracker.

5.1 Effect on Fault Localization

Methodology Our analysis follows the evaluation methodology proposed by Pearson et al. [30] and reuses its experimental infrastructure. Specifically, our analysis considers the following FL techniques, debugging scenario, and effectiveness measure:

FL techniques: We selected six widely studied FL techniques: Barinel, DStar, Jaccard, Ochiai, Op2, and Tarantula.

Debugging scenario: We consider the "best-case" debugging scenario: localizing any one defective statement is sufficient [30].

FL effectiveness: A FL technique outputs a list of statements ranked by suspiciousness; its *absolute score* is the rank of the first defective statement in that list. We measured the absolute score for each FL technique and defect. Based on the absolute score, we measured *top-n*, the best current measure of FL effectiveness, which determines how often a technique reports the first defective statement in the top-n suggested statements. According to two recent studies, top-5 and top-10 are relevant for practitioners [20], and top-200 is relevant for APR [23].

We ran each FL technique twice—once with the developer-provided triggering tests and once with the user-provided triggering tests, each run together with all non-triggering tests. We ran it on the defective version, which is the version before the commit with the bug fix. Each triggering test fails.

Results FL techniques consistently perform worse for user-provided tests. Figures 4 and 5 show differences between developer-provided and user-provided triggering tests for each FL technique.

Figure 4 shows a density plot of the absolute scores. For each FL technique, the scores are statistically significantly worse when using user-provided tests, and the effect size is small (paired t-test; $p < 0.01$; Cohen's d between 0.2 and 0.5). We did not observe project-specific differences (those plots are omitted for space).

Figure 5 shows what fraction of all defects each FL technique can usefully localize. More specifically, it shows what fraction have a score of ≤ 5, ≤ 10, and ≤ 200. The top-n performance of all FL techniques is 5–14% less for user-provided tests. We observed

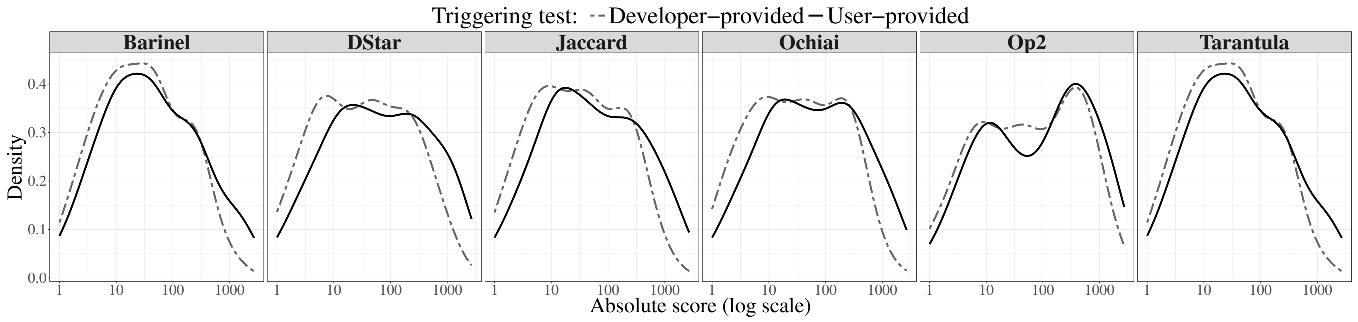


Figure 4: Comparing fault localization performance when using developer-provided vs. user-provided triggering tests.

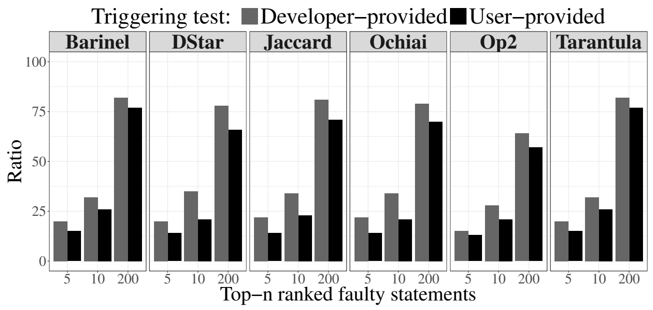


Figure 5: Comparing fault localization performance when using developer-provided vs. user-provided tests. Top-n is the percentage of defects whose defective statements appear within the top *n* of the technique's suspiciousness ranking.

some differences in relative performance when comparing FL techniques. For example, when considering top-5, Jaccard and Ochiai perform best on developer-provided tests, but Barinel and Tarantula perform best on user-provided tests. However, none of these differences is statistically significant. Overall, our results corroborate previous findings that most spectrum-based FL techniques are equally effective—the choice of the scoring formula matters little.

5.2 Effect on Automated Program Repair

Methodology Our analysis considers the following APR techniques and effectiveness measures for 18 defects.

APR techniques: We selected two APR techniques that were previously evaluated on Defects4J, using developer-provided triggering tests: jGenProg/astor [24], a search-based repair technique, and ACS [38], a synthesis-based technique. We used the implementations provided by the techniques' authors[3].

APR effectiveness: To evaluate the differences between developer-provided and user-provided tests, we selected three measures: (1) the ability to generate a *t-adequate* (test-suite-adequate) patch that passes all tests, (2) patch correctness, and (3) repair time. To determine correctness, we applied the same criteria used by previous evaluations—manual inspection and comparison with the developer-committed fix. For repair time, we averaged the measures over 10 runs, again for consistency [24]. We also inspected the tests and generated patches to understand how the differences in the tests affected the patch generation.

[3]https://github.com/SpoonLabs/astor/, https://github.com/Adobee/ACS

Defects: Of the 100 defects for which we successfully extracted user-provided triggering tests, ACS or jGenProg generated a patch for 7[4]. To increase the number of repairable defects for this analysis, we extracted an additional 11 user-provided triggering tests for the remaining Defects4J defects that were previously repaired [24, 38]. Overall, we evaluated the APR techniques on 18 defects.

We ran each APR technique twice—once with the developer-provided triggering tests and once with the user-provided triggering tests, each run together with all non-triggering tests. We adopted the timeouts reported in the previous evaluations for comparability. Specifically, we used a 3-hour timeout for jGenProg and for ACS. Consistent with previous evaluations, we limited the search for a repair to the defective package. To provide a consistent environment for the APR experiments, we created Docker images containing Defects4J and the APR tools, and executed them on a cluster with controlled resources per job (4GB RAM, 2 CPUs).

Results User-provided tests result in fewer generated patches, fewer correct patches, and a substantial increase in repair time. Table 5 gives the results for patch production, patch correctness, and repair time for jGenProg and ACS.

With developer-provided tests, jGenProg generated a patch for 7 defects—one of which is correct. With user-provided tests, jGenProg generated a patch for 6 defects—none of which is correct. For Math-2, jGenProg generated the same incorrect patch regardless of the type of triggering test. For Math-60, jGenProg generated a different incorrect patch. Finally, jGenProg's repair time substantially increased with user-provided tests. For example, it increased from 203 to 979 seconds for Math-2, while generating the same incorrect patch. In two cases, Math-8 and Math-95, the type of triggering test had no noticeable effect on the repair time. For Math-8, the instructions in both triggering tests were identical, but the user-provided test had one less assertion. For Math-95, the developer-provided test contained more instructions and assertions.

With developer-provided tests, ACS generated a patch for 12 defects—11 of which are correct. With user-provided tests, ACS generated a patch for 6 defects—5 of which are correct. For Lang-24, Math-35, Math-82, Math-85, and Math-93, ACS generated a correct patch regardless of the type of triggering test—in all cases, the user-provided test was adopted as is. For Math-3, ACS generated a correct patch with the developer-provided test but an incorrect

[4]For the 100 defects, prior work [24, 38] reported 9 successful repairs, but we were unable to replicate a repair for Time-4 and Math-7, using jGenProg.

Table 5: Evaluation of patch production (*Patch?*), patch correctness (*Correct*), and repair time in seconds (*Time*) for developer-provided and user-provided tests.
Correctness is based on the methodology of previous evaluations: manual inspection and comparison with developer fix. A patch may be *correct* or test suite adequate (*t-adeq*). Red color indicates most important differences.

Defect	Patch?		Correct		Time		Tests Identical?
	Dev	User	Dev	User	Dev	User	
Repaired with jGenProg/astor							
Math-2	yes	yes	t-adeq	t-adeq	203	979	no
Math-5	yes	timeout	correct	NA	189	NA	no
Math-8	yes	yes	t-adeq	t-adeq	242	258	no
Math-49	yes	yes	t-adeq	t-adeq	321	2288	no
Math-60	yes	yes	t-adeq	t-adeq	25	82	no
Math-80	yes	yes	t-adeq	t-adeq	25	same	yes
Math-95	yes	yes	t-adeq	t-adeq	18	20	no
Repaired with ACS							
Lang-7	yes	error	correct	NA	178	NA	no
Lang-24	yes	yes	correct	correct	148	same	yes
Math-3	yes	yes	correct	t-adeq	503	482	no
Math-5	yes	timeout	correct	NA	910	NA	no
Math-25	yes	timeout	correct	NA	1751	NA	no
Math-35	yes	yes	correct	correct	1844	same	yes
Math-82	yes	yes	correct	correct	1356	same	yes
Math-85	yes	yes	correct	correct	83	same	yes
Math-93	yes	yes	correct	correct	289	same	yes
Math-97	yes	error	t-adeq	NA	313	NA	no
Math-99	yes	error	correct	NA	846	NA	no
Time-15	yes	timeout	correct	NA	224	NA	no

one with the user-provided test. Overall, the precision of ACS with user-provided tests drops from 92% to 42%.

After inspecting the tests and generated patches, we observed that even small differences in failing inputs can have large differences in patch correctness. For example, in Math-3, the developer-provided test contains the following assertion:

```
assertEquals(a[0] * b[0], MathArrays.linearCombination(a, b), 0d))
```

As a result, the following correct patch is generated:

```
if (len == 1) {return a[0] * b[0];}
```

In contrast, the user-provided test contains the following assertion:

```
assertEquals(1, MathArrays.linearCombination(a, b), 1e-10))
```

As a result, the following incorrect patch is generated:

```
if (len == 1) {return 1;}
```

5.3 Effect of Test Separation

In 14% of cases, developers added instructions and/or assertions to an existing test (section 4.2). Would FL and APR tools have worked better if developers had created new tests instead?

Methodology For each of the 14 defects (Table 4) for which the developer extended an existing test, we extracted the defect-triggering functionality of our merged user-provided test into a new, separate triggering test—keeping the non-triggering functionality in the existing test. In other words, we added the user-provided triggering test as a separate test to the pre-fix version of the existing developer test suite. This simulates what the developer could have done instead. (This might have undermined the organization of the test suite, but we do not consider such costs here, and a developer could put the new test in a more logical spot after fixing the bug.) We only considered the user-provided triggering tests for this analysis because they are the information available before the bug fix.

We repeated the experiments described in sections 5.1 and 5.2 for the 14 defects to compare the merged and separate user-provided triggering tests. The overall goal is to study the benefits of creating a new, small, more focused test vs. augmenting an existing test—that is, the effect of test separation on the performance of the FL and APR techniques.

Results All FL techniques consistently score better on separate triggering tests. There is a particularly large increase for top-5 (from 6% to 38%), but not top-10, which suggests that many rank-6–10 defects are now in the top-5. Figures 6 and 7 show relative differences between merged and separate user-provided triggering tests for each FL technique.

Only two defects (Lang-7 and Lang-24) considered in section 5.2 had a merged user-provided test. For Lang-7, ACS failed to generate a patch, using merged or separate user-provided tests. For Lang-24, ACS generated the same correct patch, using merged or separate user-provided tests. The boosted FL performance of separate triggering tests may benefit jGenProg, and search-based APR in general; we leave a deeper investigation for future work.

6 DISCUSSION

This section discusses implications of our results and observations from our study on fault localization and automated program repair research. It further outlines research directions and discusses limitations and threats to validity.

6.1 Implications for Automated Debugging

Fault Localization A significant amount of effort has been devoted to finding better FL techniques, and some of these novel techniques were reportedly effective in evaluations on artificial faults. However, when evaluated on real faults, the best performing FL technique changes and differences between FL techniques are mostly negligible [19, 30]. Our results confirm prior findings and suggest that evaluating FL techniques on real faults and user-provided triggering tests further diminishes these already small differences. For developer-provided tests, there is a 7% performance difference between the best and worst FL technique, considering the top-5 measure. For user-provided tests, this difference drops to 1%.

A concrete recommendation for improving FL techniques is to separate triggering tests into as many distinct tests as possible. We conjecture that merged tests yield poor FL performance for two reasons. First, a FL technique might not be able to effectively distinguish between defective and non-defective code when run with a large triggering test that achieves high code coverage. Second, once a triggering test fails, the remaining statements and assertions of that test are not executed, and hence cannot provide information about whether these would succeed or fail. FL techniques should be improved by processing and separating triggering tests before fault localization is performed. Indeed, this recommendation is consistent with Xuan and Monperrus's suggestion of using *test case purification* to improve FL techniques [40].

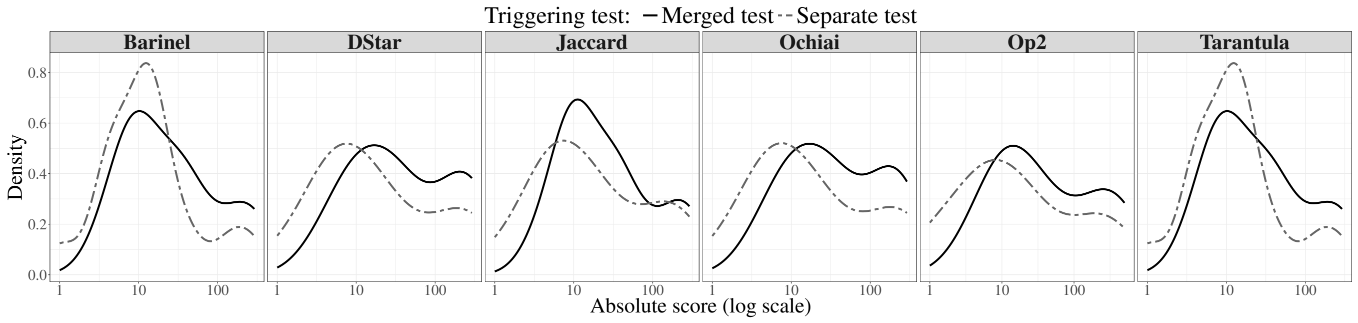


Figure 6: Comparing fault localization performance when using separate vs. merged triggering tests.

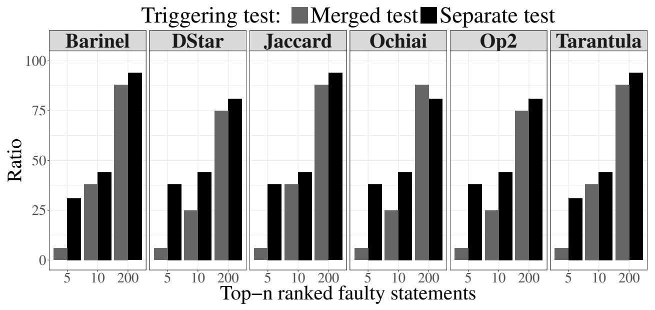


Figure 7: Comparing fault localization performance when using merged vs. separate triggering tests. Top-n is the percentage of defects whose defective statements appear within the top *n* of the technique's suspiciousness ranking. Data is for fixes in which developers augmented an existing test.

Automated Program Repair APR techniques produce fewer (correct) patches with user-provided triggering tests, which differ from developer-provided triggering tests in two ways. First, user-provided tests tend to be less targeted—that is, they achieve higher code coverage on the defective class. Second, user-provided tests tend to have weaker assertions. Prior work on APR quality and applicability (e.g., [27, 34]) primarily focused on code coverage as an effectiveness measure for the entire test suite used to guide the repair. Our results suggest that future studies should 1) separately measure code coverage for triggering and non-triggering tests and 2) measure assertion strength in addition to code coverage. The latter is particularly important because tests with poor assertions, even with very high code coverage, are likely to miss many defects, and hence result in incorrect patches [17, 23].

We observed timeouts and generally an increased repair time with user-provided triggering tests. Since fault localization is an integral part of APR techniques, FL performance may constrain effective repair. For example, researchers often use the top-200 measure for evaluating FL performance in the context of program repair. Our results for the top-200 measure show a 15% drop in FL performance when using user-provided tests. This indicates that when applying APR techniques in practice, their absolute performance may be much lower than existing benchmarks would suggest. Indeed, we observed a 4× increase in repair time with user-provided triggering tests.

Passing pre-fix tests can be wrong Our analysis revealed that the pre-fix test suite for 7% of the defects (Closure-{1, 85, 86, 89}, Math-{5, 102}, Mockito-6) contained tests that passed on the pre-fix but failed on the post-fix version of the code. These tests encoded the wrong specification and needed to be fixed with the code, which the developer did. The user-provided triggering test for these defects are all correctly failing on the pre-fix and passing on the post-fix version. This means that realistic *automated program repair is infeasible for these defects due to contradictory tests*. While some of these defects arguably represent feature requests, others are examples for tests written to test the current implementation rather than the specification. APR techniques should account for the fact that non-triggering tests could be invalid and possibly identify such tests, which contradict the triggering test(s).

6.2 Implications for Defect Benchmarks

Our findings for Defects4J are likely to affect other defect benchmarks constructed from version control history. In some cases, the effects may be even stronger. For example, for APR benchmarks such as ManyBugs [9], a common assumption is as follows: "We use all available viable tests, even those added after the version under consideration, under the assumption that the most recent set of tests correspond to the *most correct* known specification of the program." Our analysis, which replaces a single developer-provided triggering test from a single version with the original user-provided triggering test, suggests that adding many more viable and triggering tests from the latest version points to the possibility of even greater inaccuracy in estimating APR performance.

6.3 Differences in Communities by Project

By inspecting the bug reports associated with the defects in Defects4J, we had the opportunity to observe different practices and behaviors associated with different communities.

Closure The users always submitted (partially) executable triggering tests with high clarity due to the requirements set for bug reporting. Closure is a compiler that optimizes JavaScript code, and its tests usually map input source code to expected output source code or compiler errors/warnings. As a consequence, the majority of bug reports did not have a long conversation, but rather "Closure's output is unexpected for the following input source code", followed by "Fixed in revision X'". All bug reports contained executable code, but in some cases a user could not provide an expected output, as there are many ways to generate correct code. The majority of user-provided tests were adopted after augmentation—very few as-is.

Lang The users often submitted a minimal triggering test together with an analysis of the defect or a suggested patch. Compared to other projects, most users seemed to be very knowledgeable about the internals of the Lang library and put more research into why each defect was a defect—many bug reports exhibited long conversations after the initial bug report. As a result, many user-provided triggering tests were adopted after augmentation. For example, the bug report for Lang-13 included a patch and fully executable triggering test, which was still augmented to cover a `void.class` case. Developers were very active in submitting bug reports themselves (35%). However, somewhat surprisingly, developers were less likely to provide triggering tests in the bug report and were more likely to file bug reports with poor clarity.

Math The users often submitted a very targeted triggering test together with a suggested patch, yielding the largest number of provided patches across all studied projects. In other words, the users exhibited a strong domain expertise and presumably already had knowledge about the root cause of the defect. As a result, developers adopted many user-provided triggering tests as-is or with minor modifications, yielding the largest number of as-is adoptions across all studied projects.

Mockito The users often submitted convoluted triggering tests, because Mockito is a mocking framework and many of its tests require extensive scaffolding. For example, a test needs to define a mocked class or interface, attributes and methods of interest, and mocked behavior in terms of attribute values and method-call sequences. Given this complexity, the user-provided tests were often adopted as-is. In other cases, the developer separated the user-provided triggering test into multiple triggering tests or changed it to reuse existing scaffolding originally created for other tests.

Time The users often submitted a complete triggering test that was neither minimized nor targeted toward the defective code. In other words, the users mostly reported a defect without knowledge about its root cause. As a result, developers often committed (entirely) different triggering tests, yielding the smallest number of as-is adoptions across all studied projects. In these cases, it seems plausible that the developer first localized (and maybe even fixed) the defect before providing a more specific triggering test. In cases where the developer partially adopted the user-provided test, (s)he augmented it with many more assertions.

6.4 Research Directions

Test variants in empirical studies When evaluating FL or APR techniques against a benchmark, we may be lured by fragile victories in experimental settings [31] that fail to generalize to realistic settings. Our results show that even small changes to a triggering test's construction (failing input, assertion strength, and test separation) can largely affect automated debugging effectiveness.

Moving forward, researchers can increase the robustness and reliability of their evaluations by using variants of triggering tests with different levels of assertion strength and different instances of failing inputs. These test variants can be created automatically or drawn from bug reports. To support this effort, we have augmented the Defects4J dataset with the ability to run FL and APR technique evaluations with 100 user-provided triggering tests. Using these test variants in evaluations will strengthen future implementations and provide a more realistic view on FL and APR performance.

Automated extraction of triggering tests Despite the valuable and structured information contained in a bug report [6], the process of extracting a test from the report and reproducing the bug remains a manual one. Moreover, only 26% of the user-provided triggering tests, found in the inspected bug reports, where fully executable. However, an additional 61% were *partially executable*, meaning that with slight addition of scaffolding (e.g., method body, test annotations, or import statements), they became fully executable.

A tangible and achievable research goal would be to focus on *automating the process of extracting fully executable tests from partially executable ones, found in bug reports.* Why? Bug reports are often the first step in the process of fixing a defect, and many benefits can emerge from automating the extraction of executable triggering tests. For example, a newly extracted triggering test could initiate the process of fault localization and program repair—automatically generating a pull request if a candidate patch was found. The bug submitter also has the opportunity to gain feedback and improve the quality of the bug report, e.g., if the bug can not be reproduced. Furthermore, our analysis showed that for six defects, a user provided a triggering test and a developer committed a fix, but the user-provided test continued to fail. In other words, the developer did not fix the issue reported by the user (e.g., Mockito-29). Automating the process of test extraction can prevent overlooking a user-provided triggering test.

What about the remaining 13% of defects for which the bug report contained only natural text or examples? One promising direction is to combine neural machine translation with program synthesis to create a technique that, given a bug report, can synthesize an executable triggering test. Neural machine translation has been successfully used to translate code changes (diffs) to natural text commit summaries [13]. Similarly, neural machine translation also has been applied to synthesize simple programs given a natural language query [22].

6.5 Limitations and Threats to Validity

Construct validity The most important threat to construct validity relates to the test extraction process. It is possible that we interpreted something that was not the user's intention, which could have been due to poor clarity in the report, or a lack of deep expertise by the paper authors who interpreted the report. To mitigate this threat, we recorded the clarity and executability of every user-provided test in order to track our perceived understanding of the bug report (table 4). Most tests (78%) had high clarity. Furthermore, at least two authors of this paper reviewed each qualitative measurement; whenever they disagreed, the authors examined the bug report more closely.

For the 13/100 bug reports with "no executability" (table 4), we read the user bug report and created a triggering test. We might have misinterpreted the bug report. To mitigate this threat, three evaluators (not authors of this paper) examined all 100 bug reports and extracted user-provided triggering tests. In 99 cases, their majority vote was that the extracted user-provided test corresponds to the bug report. The only exception was for Lang-1[5], whose bug report was closed with a developer-provided fix and later re-opened to handle a different bug.

[5]Issue tracker entry: https://issues.apache.org/jira/browse/LANG-747

The triggering test and commit in Defects4J relate to the latter. This cannot be determined by reading the bug report text alone; we determined it by examining the commits as well.

In our empirical evaluations, we selected several measures for FL and APR effectiveness. These measures may not be good proxies for actual effectiveness and applicability. However, we have selected the most accepted measures to date. For example, we report on top-n rather than EXAM scores for FL performance.

Internal validity Threats to internal validity relate to the question whether our experiments properly isolate the studied effect of the type of triggering test. To control for possible confounding factors, we integrated each user-provided triggering test into the pre-fix version of the project's test suite at the exact same location that the developer modified in the bug-fixing commit. This means, that all non-triggering tests and the test suite structure remained identical. Additionally, we investigated the effect of test separation.

External validity A general threat to external validity is the representativeness of the selected subjects. Our study used open-source projects from Defects4J, which may not be representative of other software projects, development processes, or issue reporting behavior. Most of the studied projects are libraries or developer tools. The nature of these projects may have encouraged users to disproportionately submit triggering tests along with the bug report. We selected 6 widely studied spectrum-based FL techniques, but other kinds of FL techniques, such as IR-based FL techniques may perform differently. We selected 2 APR techniques: one search-based technique and one synthesis-based technique that focuses on precise repair of program conditions. As a result, our findings may not apply to other types of APR techniques.

Further, our evaluation makes a conservative assumption: at the time a developer runs a FL or APR tool, the developer only relies on the user-provided triggering test. It is possible that this assumption is wrong: a developer may first write additional triggering tests before attempting to automatically localize and fix the bug. Our results may not carry over to such a use case.

7 RELATED WORK

Rizzi et al. note in their paper on Klee [32] that unexamined assumptions can result in questionable research claims and wasted research effort. When examined, assumptions about use cases and developers have been successful in establishing new research directions for the automated debugging community [2, 29]. Similarly, several researchers have highlighted the importance of considering contextual factors in empirical studies in software engineering, such as organization factors [3], sensitivity factors [35], and human aspects of testing and debugging [7, 29]. This paper examines implicit assumptions about fault localization and automated program repair, focusing on the difference between user-provided and developer-provided tests.

Fault localization research has a long history of techniques and evaluations. The most common techniques are spectrum-based [1, 4, 11, 14], but slice-based [39], model-based [37], and mutation-based [26, 28] techniques have also been proposed and evaluated. To our knowledge, this study is the first to compare differences in triggering tests and explain the impact on fault localization and automated program repair.

Several benchmarks containing seeded or real faults with triggering tests exists [8, 9, 12, 16]. Some benchmarks are constructed by hand-seeding or from mutation. Other benchmarks use real bugs from open source repositories. Defects4J [16] provides 395 real bugs for 6 real-world programs ranging between 22K and 91K LOC. Moreover, all 6 programs feature developer-provided tests and each bug is reproducible with an exposing test case. Other benchmarks, such as ManyBugs [9], have a more aggressive stance for obtaining test cases. ManyBugs provides 185 real bugs for 9 real-world programs ranging between 97K and 1,099K LOC. The programs also have a comprehensive test suite, with a total of 10,468 test cases. In the benchmark, all "viable tests", even those added many years later, are used when evaluating a defect. Finally, DBGBench [7], provides a dataset containing 27 real errors and a full debugging history of professional developers solving defects. Unfortunately, none of these fault databases provides user-provided tests.

Many studies have investigated how developers file bug reports and what information they contain. Bug reports contain more than natural text; they also contain stack traces, tests, source code, and patches that can be automatically extracted [6]. Including these elements increases the chances that a bug will be resolved [5]. Rather than using triggering tests, researchers have explored using information retrieval (IR) techniques to search code by using terms from bug reports in order to perform fault localization [33] or even combining IR techniques with spectrum-based techniques [21].

8 CONCLUSION

For decades, researchers have envisioned how automated debugging tools could help developers localize and repair defects in code. This paper used triggering tests extracted from bug reports to evaluate the effectiveness and practicality of these approaches and observed noticeable differences with previous evaluations. In particular, the triggering tests developers provide after fixing a bug are often not representative of the ones users provide before the bug is found and fixed. As a result, fault localization and automated program repair tools perform worse on user-provided tests. Incorporating user-provided triggering tests into empirical evaluations is one step toward realistic evaluations, which are important to enable automated debugging reach its full potential.

This paper draws on several observations, in particular from manual analysis of more than 100 bug reports and triggering tests, and discusses several implications that may help define future research directions. For example, researchers building automated debugging tools should consider additional factors, such as resolving conflicting tests prior to an attempted repair, accounting for triggering tests that expose a defect but only weakly assert on the correct behavior, and automatically separating triggering tests.

Programmers have been waiting a long time for usable automated debugging tools. We believe that, to further advance the state-of-the-art in this area, we must steer research toward more promising directions that take into account triggering tests that are likely to be encountered in real settings.

ACKNOWLEDGMENTS

We thank Sandy Kaplan, Ben Kushigian, Amit Rawat, and the anonymous reviewers for their comments and valuable suggestions.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*. Washington, DC, USA, 89–98. http://dl.acm.org/citation.cfm?id=1308173.1308264

[2] Aaron Ang, Alexandre Perez, Arie van Deursen, and Rui Abreu. 2017. *Revisiting the Practical Use of Automated Software Fault Localization Techniques*. IEEE, United States, 175–182. https://doi.org/10.1109/ISSREW.2017.68

[3] J. Aranda and G. Venolia. 2009. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE 2009, Proceedings of the 31st International Conference on Software Engineering*. Vancouver, BC, Canada, 298–308. https://doi.org/10.1109/ICSE.2009.5070530

[4] Aritra Bandyopadhyay. 2011. Improving Spectrum-based Fault Localization Using Proximity-based Weighting of Test Cases. In *ASE 2011: Proceedings of the 26th Annual International Conference on Automated Software Engineering (ASE '11)*. Lawrence, KS, USA, 660–664. https://doi.org/10.1109/ASE.2011.6100150

[5] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *FSE 2008: Proceedings of the ACM SIGSOFT 16th Symposium on the Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. New York, NY, USA, 308–318. https://doi.org/10.1145/1453101.1453146

[6] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Extracting Structural Information from Bug Reports. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR '08)*. New York, NY, USA, 27–30. https://doi.org/10.1145/1370750.1370757

[7] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is It Fixed? An Experiment with Practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. New York, NY, USA, 117–128. https://doi.org/10.1145/3106237.3106255

[8] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Softw. Engg.* 10, 4 (Oct. 2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[9] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256. https://doi.org/doi.ieeecomputersociety.org/10.1109/TSE.2015.2454513

[10] Ralph Guderlei, René Just, and Christoph Schneckenburger. 2008. Benchmarking testing strategies with tools from mutation analysis. In *International Conference on Software Testing Verification and Validation Workshop (ICSTW)*. 360–364.

[11] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. 2000. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability* 10, 3 (2000), 171–194. https://doi.org/10.1002/1099-1689

[12] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. Los Alamitos, CA, USA, 191–200. http://dl.acm.org/citation.cfm?id=257734.257766

[13] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically Generating Commit Messages from Diffs Using Neural Machine Translation. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. Piscataway, NJ, USA, 135–146.

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

[15] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, CA, USA, 433–436.

[16] 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)*. New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[17] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[18] René Just and Franz Schweiggert. 2011. Automating unit and integration testing with partial oracles. *Software Quality Journal (SQJ)* 19, 4 (2011), 753–769.

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

[20] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[21] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. New York, NY, USA, 579–590. https://doi.org/10.1145/2786805.2786880

[22] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. 2017. *Program synthesis from natural language using recurrent neural networks*. Technical Report UW-CSE-17-03-01. University of Washington Department of Computer Science and Engineering, Seattle, WA, USA.

[23] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. New York, NY, USA, 702–713. https://doi.org/10.1145/2884781.2884872

[24] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*. https://doi.org/10.1145/2931037.2948705

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

[26] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. Washington, DC, USA, 153–162. https://doi.org/10.1109/ICST.2014.28

[27] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2017. Do Automated Program Repair Techniques Repair Hard and Important Bugs? *Empirical Software Engineering Journal (ESEM)* (2017), 1–47.

[28] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based Fault Localization. *Softw. Test. Verif. Reliab.* 25, 5-7 (Aug. 2015), 605–628. https://doi.org/10.1002/stvr.1509

[29] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. New York, NY, USA, 199–209. https://doi.org/10.1145/2001420.2001445

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

[31] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[32] Eric F. Rizzi, Sebastian Elbaum, and Matthew B. Dwyer. 2016. On the Techniques We Create, the Tools We Build, and Their Misalignments: A Study of KLEE. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. New York, NY, USA, 132–143. https://doi.org/10.1145/2884781.2884835

[33] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. 2014. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In *2014 IEEE International Conference on Software Maintenance and Evolution*. Jaipur, India, 161–170. https://doi.org/10.1109/ICSME.2014.38

[34] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *ESEC/FSE 2015: The 10th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Bergamo, Italy, 532–543.

[35] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. New York, NY, USA, 1–11. https://doi.org/10.1145/2771783.2771797

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

[37] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. 2002. Model-Based Debugging or How to Diagnose Programs Automatically. In *Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence (IEA/AIE '02)*. London, UK, UK, 746–757.

[38] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. Piscataway, NJ, USA, 416–426. https://doi.org/10.1109/ICSE.2017.45

[39] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (March 2005), 1–36. https://doi.org/10.1145/1050849.1050865

[40] Jifeng Xuan and Martin Monperrus. 2014. Test Case Purification for Improving Fault Localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. New York, NY, USA, 52–63. https://doi.org/10.1145/2635868.2635906