Are mutants a valid substitute for real faults in fault localization research?

ABSTRACT

Fault localization is a technique that takes as input a possibly-faulty program and produces as output a ranked list of suspicious code locations at which the program may be defective. When researchers propose a new fault localization technique, they evaluate it on programs with known faults, and score the technique based on where in its list the actual defect appears. This enables comparison of multiple fault localization techniques to determine which one is better.

Previous research has evaluated fault localization techniques using artificial faults, called mutants, generated either by mutation tools or manually. In other words, previous research has determined which fault localization techniques are best at finding artificial faults. However, it is not known which fault localization techniques are best at finding real faults. It is not obvious that the answer is the same, given previous work showing that artificial faults have both similarities to and differences from real faults.

We evaluated 7 previously-studied fault localization techniques, applying them to 251 real faults and 2831 artificial faults in 5 real-world programs. In our experiments, artificial faults were not useful for predicting which fault localization techniques perform best on real faults.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Fault Localization; Real Faults; Mutants.

1. INTRODUCTION

A fault localization technique (for short, FL technique) directs a programmer's attention to specific parts of a program. Given one or more failing test cases and zero or more passing test cases, a FL technique outputs a sorted list of suspicious program locations, such as lines, statements, or declarations. The FL technique uses heuristics to determine which program locations are most suspicious — most likely to be erroneous and associated with the fault. A programmer can save time during debugging by focusing attention on the most suspicious locations [16].

Dozens of fault localization techniques have been proposed in the literature [42]. It is desirable to evaluate and compare these techniques, both so that practitioners can choose the ones that help them solve their debugging problems, and so that researchers can better build new fault localization techniques.

A fault localization technique is valuable if it works on real faults. Previous comparisons [42] of fault localization techniques have evaluated their efficacy on 35 real faults of a single small numerical program (`space`) [38], or on fake faults called mutants. A mutant is a small, local change to the program text; a mutation typically changes one expression or statement, by replacing one operator or expression by another. For example, the expression `a+5` could be mutated to `a-5` or to `a+0`. These mutants were sometimes manually-seeded faults created by computer science students (in the Unix set composed by 10 programs, and in the set of Unix utilities: `flex`, `grep`, `gzip`, and `make`) [41, 43] or by researchers (in the Siemens set of 7 very small programs) [17]; and were sometimes automatically created by a tool [24, 25, 46].

Mutants differ from real faults in many respects, including their size, their distribution in code, and their difficulty of being detected by tests [23]. It is possible that an evaluation of FL techniques on real faults would yield different outcomes than previous evaluations on mutants. In this case, previous recommendations would need to be revised, and practitioners and researchers would choose different techniques to use and improve. It is also possible that an evaluation of FL techniques on real faults would yield the same recommendations, thus resolving a cloud of doubt that currently hangs over the field. Either result would be of significant scientific interest. The results also have implications beyond fault localization itself. For instance, it would help to indicate which fault localization approaches, if any, should be used to guide automated program repair techniques [35].

This paper evaluates the performance of 7 previously-studied fault localization techniques on both real and artificial faults, and compares the results to discover the strength of the correlation between the two types of faults. In particular, the experiments reported in this paper are based on 5 projects, 251 real faults, and 2831 artificial faults, generated by mutating the fixed versions of the real faults.

While we find some positive correlation between each technique's performance on real faults and on similarly located artificial faults, we do *not* find any evidence that techniques that do well on artificial faults also do well on real faults.

2. EVALUATING FAULT LOCALIZATION

There have been a number of studies dedicated to evaluating and comparing FL techniques [3–6, 19, 20, 24, 25, 29, 30, 32, 36, 41, 46]. Table 1 summarizes the results of these studies in terms of the programs used, the number of real and artificial

Table 1: **Selected fault-localization studies.**

`Siemens set` includes `printtokens`, `printtokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `totinfo`. `Unix set` includes `Cal`, `Checkeq`, `Col`, `Comm`, `Crypt`, `Look`, `Sort`, `Spline`, `Tr`, and `Uniq`. Manually-seeded faults are represented by h, and mutation-based faults are represented by m.

Reference	**Language**	**Ranking** (from best to worst)	**Projects**	**Artificial faults**	**Real faults**
[19]	C	Tarantula	`Siemens`	122 h	-
[4]	C	Ochiai, Tarantula	`Siemens`	120 h	-
[3]	C	Ochiai, Tarantula	`Siemens`, `space`	128 h	34
[5]	C	Barinel, Ochiai, Tarantula	`Siemens`, `space`, `gzip`, `sed`	141 h	38
[6]	C	Tarantula	`Concordance`	200 m	13
[30]	C	Op2, Ochiai, Tarantula	`Siemens`, `space`	132 h	32
[32]	C	Metallaxis, Ochiai	`Siemens`, `space`, `flex`, `grep`, `gzip`	859 h, m	12
[25]	C	Ochiai, Tarantula	`Siemens`, `space`, `NanoXML`, `XML-Security`	164 h	35
[41]	C	DStar, Ochiai, Tarantula	`Siemens`, `space`, `ant`, `flex`, `grep`, `gzip`, `make`, `sed`, `Unix`	436 h	34
[29]	C	MUSE, Op2, Ochiai	`space`, `flex`, `grep`, `gzip`, `sed`	11 h	3
[46]	Java	Ochiai, Tarantula	`JExel`, `JParsec`, `Jaxen`, `Apache Commons Codec`, `Apache Commons Lang`, `Joda-Time`	1800 m	-
[20]	C/Java	DStar, Tarantula	`printtokens`, `printtokens2`, `schedule`, `schedule2`, `totinfo`, `Jtcas`, `Sorting`, `NanoXML`, `XML-Security`	104 h	-
[24]	C	DStar, Ochiai, Tarantula	`Siemens`, `space`, `NanoXML`, `XML-Security`	165 h	35
this	Java	Ochiai, Barinel, Tarantula, Op2, DStar, Metallaxis, MUSE	`Chart`, `Closure`, `Lang`, `Math`, `Time`	-	251
this	Java	Metallaxis, MUSE, Op2, Ochiai, Barinel, Tarantula, DStar	`Chart`, `Closure`, `Lang`, `Math`, `Time`	2831 m	-

faults considered, and the resulting ranking of techniques. Notably, the majority of studies revolve around the same set of programs and use largely artificial faults. The results in terms of which FL technique emerges as the best out of a study are quite varied across the studies. In this paper, we aim to investigate to what extent the use of artificial faults influences findings of such experiments. In order to make this possible, this section takes a closer look at how a fault localization technique's output can be evaluated.

2.1 Evaluation metrics

A fault localization technique T takes as input a program P and a test suite with at least one failing test, and it produces as output a sorted list of suspicious program locations, such as lines, statements, or declarations. For concreteness, this paper uses statements as the locations, but the ideas also apply to fault localization techniques that output suspicious locations at other levels of granularity.

Given a fault localization technique T and a program P of size N with a known defective statement d, a numerical measure of the quality of the fault localization technique can be computed as follows [34, 37]: 1) run the FL technique to compute the sorted list of suspicious lines; 2) if d is the nth element in the list, use one of the several metrics proposed in the literature to evaluate the effectiveness of a FL technique, for example, LIL [29], T-score [26], Expense [19], or *EXAM* score [40].[1]

LIL [29] measures the effectiveness of a fault localization technique on automated program repair. T-score [26] computes the percentage of components that a developer *would not have* to inspect until finding the first faulty one. By contrast, Expense [19] and *EXAM* score [40] compute the percentage of components that a developer *would have* to inspect until finding the first faulty one — that is, the cost of going through the ranked list to find the faulty component. Both Expense and *EXAM* score compute n/N in which N is the total number of executed statements for Expense, and the total number of statements in the program for *EXAM* score. The score returned ranges between 0 and 1, and smaller numbers are better.

[1]These scores are different than the "suspiciousness score" that the fault localization technique may use for constructing the sorted list of suspicious program statements.

2.2 Extensions to fault localization evaluation

The standard technique for evaluating fault localization, described in section 2.1, handles defects that consist of a single change to an executable statement in the program, as is the case for mutants. To evaluate fault localization on real faults, it is necessary to extend the methodology to account for ties in the suspiciousness score, multi-line statements, multi-line faults, faults of omission, and defective non-executable code such as declarations.

2.2.1 Ties in the suspiciousness score

Most fault localization techniques first compute a suspiciousness score for each program statement, and then sort statements according to suspiciousness score in order to produce a ranked list of program statements, which is the FL technique's output. When two statements have the identical suspiciousness score, then an arbitrary sorting choice could affect the *EXAM* score.

We compute the expected *EXAM* score, assuming that the sorting function breaks ties arbitrarily. That is, when multiple statements have the same suspiciousness score, then all of them are treated as being the nth element in the output, where n is their average rank [37, 42].

```
 public boolean updateState(State s) {
+  if (!flag)
+    return false;
   setState(s);                    // goal report
   return true;
 }
```

```
 public boolean updateState(State s) {
+  if (flag)
     setState(s);                  // goal report
   return true;
 }
```

```
 public boolean updateState(State s) {
+  if (flag) {
     setState(s);  // goal report for "if (flag)"
     otherStatement();
     return true;  // goal report for "return false"
+  }
+  return false;
 }
```

Figure 1: Faults of omission and goal statements for reporting them. For the third example, the final `}` would be a better goal report for `return false;`, but current FL techniques do not report it and so we count the preceding `return true;` statement as also a correct report.

2.2.2 Multi-line program statements

A single statement may span multiple lines in a program. Some FL techniques give reports in terms of lines and some in terms of statements. To permit consistent comparison, our methodology converts them all to report in terms of program statements. Any FL tool report that is within a statement is automatically converted to being a report about the first line of the smallest enclosing statement.

2.2.3 Multi-statement faults

The *EXAM* score assumes that a single program statement is defective. This assumption holds for mutants and often other artificial faults. However, most real-world bug fixes span multiple statements [22]. We consider a fault localization technique to identify a fault as soon any defective statement appears in the FL technique's output. Future work could compare to other metrics, such as the rank of the last defective statement that a technique identifies; that is a good measure under the assumption that a technique needs to identify all defective statements in order to be useful to a software developer.

2.2.4 Faults of omission

In some cases, a bug fix consists of adding new code rather than changing existing code. The defective program contains no defective statement: every expression, statement, and declaration in the program is correct, but some are missing. In a database of real-world faults [22], 31% of the faults are faults of omission whose fix involves only adding new code. Previous studies on the effectiveness of fault localization have not explicitly reported whether and how this issue was addressed. Wong et al. [14] suggested that analyzing the code around statements with an high suspiciousness score, and noticing unexpected executions of some statements, could be a first step to identify a fault of omission. In the following, we describe our methodology to deal with this issue.

A FL technique communicates with the programmer in terms of the program's representation: lines of source code. A FL technique is most useful if it identifies where in the source code the programmer needs to make a change. Therefore, when a bug fix involves inserting code at a statement without changing code at that statement, the technique should report the textually immediately following statement. Ideally, this is the next line, which is exactly where the programmer should insert the code. However, many FL techniques have a serious limitation: they never rank or report lines consisting of scoping braces, such as the final } of a method definition. Even though that would be the best program location to report when the insertion is at the end of a method, we count the current last statement as a correct report. Figure 1 shows examples.

A more serious complication is that the developer inserted the new code at some statement, but other statements might be equally valid choices for a bug fix. Consider the following example, drawn from the patch for the defect Closure-15 in Defects4J's:

```
1    if (n.isCall() && ...)
2      return true;
3    if (n.isNew() && ...)
4      return true;
5+   if (n.isDelProp())
6+     return true;
7    for (Node c = n.getFirstChild(); ...) {
8      ...
```

The programmer could have inserted the missing conditional before line 1, between lines 2 and 3, or where it actually was inserted. A FL technique that reports any of those statements is just as useful as one that reports the statement the programmer happened to choose.

For every fault of omission, we manually determined the set of *candidate* statements at which this code block could be inserted to fix the defect (lines 1, 3, and 6 in the example above). We consider a fault localization technique to identify a fault of omission as soon any candidate statement appears in the FL technique's output.

2.2.5 Faults in non-ranked statements

Some fault localization techniques have limitations in that they fail to report some statements in the program. Here are examples:

Non-executable code (declarations) All the fault localization techniques that we evaluate have a weakness in that they only output a list of suspicious statements; they never report a declaration in their list of suspicious locations.

However, a non-executable declaration can be faulty, such as a supertype declaration (in Java, an `extends` or `implements` clause in a class declaration) or the data type in a field or variable declaration. In a database of real-world faults [22], 4% of real faults involve some non-executable code locations and 3% involve only non-executable code locations.

Non-mutable statements The mutation-based fault localization techniques that we evaluate have a weakness in that they only output a list of mutatable statements. However, some faulty, executable statements are not mutatable due to compiler restrictions. For example, a `break` or `return` statement may not be able to be deleted (even if it were reinserted at a different line) because doing so would cause compilation errors due to the violation of control-flow properties enforced by the compiler. In a database of real-world faults [22], 10% of real faults involve some non-mutable yet executable statements.

Previous studies on the effectiveness of fault localization have not considered faults in non-ranked statements. We ensure that the ranked list of statements produced by a FL technique always contains every statement in the program: If a faulty statement is missing from the list of suspicious statements, we assign it a rank equal to the average rank of all program statements in the fault-relevant classes (see section 3.2) that do not appear in the list of suspicious statements.

For example, consider a program whose fault-relevant classes contain 1000 statements. If a FL technique ranks 900 statements, but misses the faulty statement, that statement is assigned a rank of 950 (the average rank of that statement, if the remaining 100 statements were appended to the ranking in arbitrary order).

3. SUBJECTS OF INVESTIGATION

The goal of this paper is to evaluate the efficacy of fault localization techniques on both real and artificial faults using developer-written test suites. This section details the subjects that our study investigated: fault localization techniques, programs, defects, and test suites.

3.1 Fault localization techniques

Although many fault localization techniques [42] have been proposed in previous works, this paper considers 2 families of techniques: spectrum-based fault localization (SBFL techniques for short) [4, 19, 30, 41], which is the most studied and evaluated FL technique; and a recently proposed technique called mutation-based fault localization (MBFL technique for short) which is reported to outperform SBFL techniques [29, 32].

3.1.1 Spectrum-based FL techniques

Most fault localization techniques, including all that we examine in this section, yield a ranked list of program statements sorted by the suspiciousness score $S(s)$ of the statement s. Spectrum-based fault localization techniques [4, 19, 30, 41] are statistical in nature. The more often a statement s is executed by failing tests, and the less often it is executed by passing tests, the more suspicious the statement is considered — that is, the more likely the statement is to be defective and the root cause of the failures.

The specific SBFL formulas used in this paper are: Tarantula [19], Ochiai [3, 4], DStar [41], Barinel [5], and Op2 [30]. There are many more possible formulas [42], however these 5 are amongst the best performing ones [42]. In the following, let *totalpassed* be the number of passed test cases and $\mathit{passed}(s)$ be the number of those that executed statement s (similarly for *totalfailed* and $\mathit{failed}(s)$).

1. Tarantula [19]

$$S(s) = \frac{\mathit{failed}(s)/\mathit{totalfailed}}{\mathit{failed}(s)/\mathit{totalfailed} + \mathit{passed}(s)/\mathit{totalpassed}}$$

2. Ochiai [3, 4]

$$S(s) = \frac{\mathit{failed}(s)}{\sqrt{\mathit{totalfailed} \cdot (\mathit{failed}(s) + \mathit{passed}(s))}}$$

3. Op2 [30]

$$S(s) = \mathit{failed}(s) - \frac{\mathit{passed}(s)}{\mathit{totalpassed} + 1}$$

4. Barinel [5]. For single-fault scenarios, Barinel's underlying Bayesian update reduces to

$$S(s) = 1 - \frac{\mathit{passed}(s)}{\mathit{passed}(s) + \mathit{failed}(s)}$$

5. DStar [41]

$$S(s) = \frac{\mathit{failed}(s)^*}{\mathit{passed}(s) + (\mathit{totalfailed} - \mathit{failed}(s))}$$

where variable $* > 0$. We chose $* = 2$, as that value is most thoroughly explored by Wong et al. [41].

3.1.2 Mutation-based FL techniques

Mutation-based fault localization techniques [29, 32] extend SBFL techniques by considering not just whether a statement is executed, but whether that statement's execution is important to the test's success or failure — that is, whether a change to that statement changes the test result. The more often a statement s affects failing tests, and the less often it affects passing tests, the more suspicious the statement is considered.

The key idea of MBFL is to localize injected mutants, based on the assumption that test cases that *kill* mutants carry diagnostic power. A test case kills a mutant if executing the test on the mutant leads to a different test outcome than executing it on the original program. In our study we have considered the two known MBFL techniques: MUSE [29] and Metallaxis [32].

In the MUSE [29] approach, the suspiciousness score $S(s)$ for each statement s is calculated as follows. For each statement s, a set of mutants $\mathit{mut}(s)$ is generated. Given $\mathit{mut}(s)$, the suspiciousness score of s is given by

$$S(s) = \frac{1}{|\mathit{mut}(s)|} \cdot \sum_{m \in \mathit{mut}(s)} \left(\mathit{failed}(m) - \frac{\mathit{f2p}}{\mathit{p2f}} \cdot \mathit{passed}(m) \right)$$

where *failed(m)* is the number of failing tests that passed with m inserted, and *f2p* is the number of cases in the whole program where a mutant caused any failing test to pass. *passed(m)* and *p2f* are defined similarly. The MUSE approach is based on the assumption that mutating faulty statements is more likely to cause passing tests to fail than mutating correct statements is to change the test outcome of tests.

Metallaxis [32] calculates $S(s)$ also by generating $\mathit{mut}(s)$, and then computing

$$S(s) = \max_{m \in mut(s)} \frac{failed(m)}{\sqrt{\mathit{totalfailed} \cdot (\mathit{failed}(s) + \mathit{passed}(s))}}$$

where *failed(m)* is the number of failing tests whose outcomes are changed *at all* by the insertion of m (e.g. by failing at a different point, or with a different error message) and *totalfailed* is the number of tests that fail on the original test suite. Note that this definition of *killing a mutant* is less restrictive than MUSE's: whereas MUSE only considers a failing test to kill a mutant if the mutant causes the test to pass, Metallaxis detects any change in the test's behavior.

3.1.3 Implementation

We re-implemented all the fault localization techniques using shared infrastructure. This ensures that our results reflect differences in the techniques, rather than differences in their implementations. For the coverage-based techniques, coverage data was collected using an improved version of GZoltar [9][2]. For the mutation-based techniques, mutation analysis data was obtained from a tool built upon the Major mutation framework [21][3].

As an optimization, we did not run test t on mutant m if we knew in advance that *failed*$(m) = 0$. That occurs when (a) t, run without any mutants, does not executes the code containing m, or (b) no failing test in the suite kills m.

When *failed*$(m) = 0$, the Metallaxis formula can be evaluated without knowing *passed*(m). The optimization slightly affects the output of MUSE, because MUSE assigns suspiciousness 0 to all mutants where *failed*$(m) = 0$. Ordinarily, MUSE assigns suspiciousness ≤ 0 to all mutants where *failed*$(m) = 0$, and suspiciousness > 0 to almost all others. Therefore, our optimization affects MUSE's ranking of the least-suspicious statements, but only those. We considered that to be acceptable, given that running our experiments would have taken several times longer without the optimization.

3.2 Programs

We used the programs in the Defects4J [22] dataset (v1.0.0), which consists of 357 real faults from five open source projects: *JFreeChart* (26 faults), *Google Closure compiler* (133 faults), *Apache Commons Lang* (65 faults), *Apache Commons Math* (106 faults), and *Joda-Time* (27 faults). For each fault, Defects4J provides faulty and fixed program versions with a minimized change that represents the isolated bug fix. This change indicates which lines in a program are defective.

Patch minimization.

Each fault in Defects4J consists of a faulty and a fixed version of a program; the difference between these two versions is the patch that a programmer wrote to fix the defect. We used both automated analysis, such as delta debugging, and manual analysis to minimize the patch. The result was the smallest change or patch that corrects the fault in the way the programmer intended. Examples of changes that this process removed from the patches are refactorings such as renamings, and new features that the programmer added in the same commit as a bug fix. In some cases our manual analysis made a patch larger than the minimum achievable. For example, if a patch had added a function and a call to the function, then our automated analysis would find a smaller change that just added the call to the function, treating adding the function (without calling it) as a meaning-preserving refactoring. Three authors of this paper manually minimized and examined the patches, and agreed on a patch that preserved the spirit of the programmer's fix.

[2]http://www.gzoltar.com/

[3]http://mutation-testing.org/

Given a minimized patch, we employed an automated analysis to obtain all removed, inserted, and changed lines, but ignoring changes to declarations without an initializer, addition and removal of compound statement delimiters (curly braces `{}`), annotations, and import statements. These statements do not affect the program's algorithm or are trivial to add, and therefore a FL tool should not report them. Any lines remaining in the patch are defective lines that a FL tool should report.

Fault-relevant classes.

As an optimization, we applied each fault localization technique only to the *fault-relevant classes.* A fault-relevant class for a defect is any class that is executed by any fault-triggering test for that defect. This optimization is sound, and it is one that a programmer could use when debugging a failure.

For each real and artificial fault, we obtained the set of fault-relevant classes are obtained as follows. First, independently executed each fault-triggering test, monitoring the class loader during execution. This yields a set of all loaded classes; we retain only the project-related classes (i.e., this ignores classes in libraries and the Java runtime environment). The set of fault-relevant classes is a sound approximation of the set of classes that contain a defect. We do not use slicing, impact analysis, or other approaches to further localize or isolate the defective code.

3.3 Defects

Real faults: Our experiments used the 357 real faults in the Defects4J dataset. For each fault, Defects4J provides a developer-written patch (minimized as explained in section 3.2) that fixes that fault. Any statement that is either *changed* or *removed* by the minimized patch is considered defective. Any consecutive block of *inserted* lines represents a fault of omission.

Artificial faults: Our experiments used the Major mutation framework [21] to generate artificial faults. Recall that Defects4J provides, for each real fault, the faulty and the fixed program version and a fault-fixing patch that indicates which classes were defective (and fixed by that patch). For each fixed program version, Major generated artificial faults for each class fixed by the patch. More specifically, Major generated many faulty variations of each fixed program version by inserting one artificial fault into the source code at a time. Note that Major only inserted the artificial fault and no additional instrumentation. Section section 4.3 describes the experimental set up in greater detail.

3.4 Test Suites

All investigated fault localization techniques require, as an input, at least one test case that can expose the fault. For each real fault, Defects4J provides a developer-written test suite containing at least one such fault-triggering test case. To verify that each artificial fault has at least one fault-triggering test case as well, we executed the corresponding developer-written test suite and discarded artificial faults that could not be triggered by any test case.

4. METHODOLOGY

4.1 Research questions

RQ 1: How do the *relative* performances of FL techniques on artificial faults compare to their *relative* performances on real faults? (If technique A outperforms B on artificial faults, will A outperform B on real faults?)

RQ 2: How do the *absolute* performances of FL techniques on artificial faults compare to the *absolute* performances on real faults? (Is the *EXAM* score for real faults predictable from the *EXAM* score for real faults?)

RQ 3: Do the answers to RQ1 agree with previous results? (For example, "Ochiai outperforms Tarantula" [4], "Metallaxis outperforms Ochiai" [32].)

RQ 4: Do the answers to RQ1 and RQ2 depend on the kind of fault? (Is behavior different for fencepost errors vs. faults of omission?)

4.2 Data: real and artificial faults

Recall from section 3.3 that we generated artificial faults by mutating the fixed program versions in Defects4J. We could have generated an artificial fault for every possible mutation of every statement in the program, but many of these artificial faults would be in parts of the program completely unrelated to the corresponding real fault, and the cost of running our experiments would be unacceptably high. As an optimization, we only generated artificial faults for formerly-defective statements of the fixed program version — that is, those that would need to be modified or deleted to reintroduce the real fault. (Faults of omission do not require special treatment in this process, since the omission corresponds to a well-defined segment of code in the fixed program version, which can be mutated to produce artificial faults.)

In more detail: each real fault in Defects4J is associated with a faulty and fixed program version. From each of these pairs of program versions, we generated a patch which, when applied to the *fixed* version, would reintroduce the real fault. We call the set of statements modified or deleted by this patch the *scope* of the real fault. We generated artificial faults by mutating only statements in this scope. This means that if the scope is the empty set (i.e., fixing the real fault only required the deletion of erroneous code), no artificial faults can be generated; we discarded 9 real faults whose fixes only deleted erroneous code. Furthermore, for some real faults, none of the artificial faults in scope of that real fault caused any test failures; we discarded these 30 real faults as well.

Overall, the output of this process was one set of 327 real faults and 327 sets of artificial faults, each of which exists in the scope of the corresponding real fault and is detectable by the same developer-written test suite. For each real fault and FL technique, we compute the *EXAM* score for the real fault and the average *EXAM* score for the corresponding set of artificial faults.

4.3 Experimental Design

We answered our research questions through the following analyses:

RQ 1: How do the *relative* performances of FL techniques on artificial faults compare to their *relative* performances on real faults?

We used real faults to produce a "tournament ranking" of FL techniques. We evaluated each FL technique on each real fault, to get a set of *EXAM* scores for each technique. Then, we performed ANOVA for each pair of sets of scores, awarding 1 point to whichever technique (if any) is statistically significantly superior. Finally, we ranked the techniques by number of points.

We repeated this procedure for artificial faults, thus obtaining two rankings of FL techniques from best to worst: one for real faults and one for artificial faults. We used Spearman's rank correlation test to determine whether the rankings were significantly different.

For comparison, we repeated this process, awarding tournament points according to a Wilcoxon rank-correlation test instead of an ANOVA, to roughly compare the techniques' medians instead of their means.

RQ 2: How do the *absolute* performances of FL techniques on artificial faults compare to the *absolute* performances on real faults?

We break this research question into the following three components.

(a) Are artificial faults much easier/harder to localize?

To answer the question of whether the differences between the scores on artificial and real faults are significant, we used a paired, non-parametric test (the Wilcoxon signed-rank test), pairing the score for each real fault with the mean score for its associated artificial faults.

(b) Is a FL technique's *EXAM* score for artificial faults positively correlated with its *EXAM* score for real faults?

To answer the question of whether the scores on artificial and real faults are positively correlated, we compute the p-value for independence of the two sets of scores (again, pairing each real-fault score with the mean score on corresponding artificial faults) using Spearman's ρ.

(c) Is the *EXAM* score for real faults predictable from the *EXAM* score for real faults?

While we do not necessarily suspect a linear relationship, we give Pearson's r for comparison.

RQ 3: Do the answers to RQ1 agree with previous results?

We determine whether the rankings that we obtained for artificial and real faults in RQ1 correspond to prior studies as shown in table 1.

RQ 4: Do the answers to RQ1 and RQ2 depend on the kind of fault?

To answer this research question, we categorize the real faults onto three categories: single-line faults that were not faults of omission (these correspond to artificial faults generated by mutation operators, as used in many previous studies); multi-line faults that included faults of omission; and multi-line faults that did not include faults of omission.

For each category, we repeated all the analyses described in RQ1 and RQ2.

Table 2: Average number of test cases per project and per type of fault.

	Real Faults			Artificial Faults		
Project	Total	Pass	Fail	Total	Pass	Fail
Chart	185	181	4	122	118	4
Closure	987	985	2	484	477	7
Math	160	158	2	290	283	7
Lang	95	93	2	118	115	3
Time	2525	2522	3	2826	2815	11

5. RESULTS

5.1 Do relative performances correlate?

Table 3 shows the relative performances of the 7 techniques, broken down by fault type and ranking metric. The most important feature of these tables is that *there is no significant relationship between the rankings for real and artificial faults.* This means that artificial faults are not useful for the purpose of determining whether one FL technique is better than another at localizing mistakes that programmers actually make. The tournament rankings are perfectly consistent with the techniques sorted by *EXAM* score, which validates both our methodology and our R scripts that compute the statistics.

Another notable feature is that while the mutation-based techniques (MUSE and Metallaxis) consistently perform best on artificial faults, they tend to do worse than spectrum-based techniques on real faults. One reason for this may be that about 5% of real-world faults involve only non-mutatable statements, which appear last in mutation-based techniques' suspiciousness rankings. These outlier scores greatly degrade the technique's mean score, explaining why, on real faults, Metallaxis has a relatively good median score but a poor mean score.

Another contributing factor is that the artificial faults were generated by the same mutation tool as used in the calculation of the Metallaxis and MUSE formulas. As a result, artificial faults can often be fixed by mutants. For example, if an artificial fault is generated by mutating `a+b` to `a-b`, the fault localization technique will be able to repair the program by changing it back to `a+b`. This is likely why MUSE performs so well on artificial faults but so poorly on real faults: for artificial faults, there is often a mutant that makes all the failing tests pass (resulting in a very high suspiciousness score), while that is seldom true of real faults.

5.2 Do absolute performances correlate?

Artificial faults, despite not being good proxies for real faults for comparing fault localization techniques, do offer some predictive power: as shown in table 4, the *EXAM* score of a fault localization technique on a real fault is somewhat positively correlated with the technique's score on the corresponding artificial faults.

Figure 2 displays the distribution of *EXAM* scores for each fault localization technique and fault type, on a log scale. The overall distributions of scores are very similar between real and artificial faults, excepting DStar, and excepting the mutation-based techniques (for reasons discussed in section 5.1).

5.3 Do previous results generalize?

Table 3: Fault localization techniques sorted by their performance on real and artificial faults.

Real Faults		Artificial Faults	
Technique	Mean	Technique	Mean
Ochiai	0.0042968	Metallaxis	0.0083165
Barinel	0.0043400	MUSE	0.0133091
Tarantula	0.0043642	Op2	0.0176151
Op2	0.0052150	Ochiai	0.0176708
DStar	0.0071811	Barinel	0.0179941
Metallaxis	0.0091936	Tarantula	0.0180353
MUSE	0.0316299	DStar	0.0333815

(a) Techniques sorted by mean *EXAM* score

Real Faults		Artificial Faults	
Technique	Median	Technique	Median
Ochiai	0.0004183	MUSE	0.0001074
Metallaxis	0.0004219	Metallaxis	0.0001377
Barinel	0.0004336	Op2	0.0002652
Tarantula	0.0004357	Ochiai	0.0002755
Op2	0.0005275	Barinel	0.0003994
DStar	0.0005979	Tarantula	0.0004022
MUSE	0.0067051	DStar	0.0020359

(b) Techniques sorted by median *EXAM* score

Real Faults		Artificial Faults	
Technique	# Worse	Technique	# Worse
Ochiai	2	Metallaxis	6
Barinel	2	MUSE	5
Tarantula	2	Ochiai	1
DStar	1	Barinel	1
Op2	1	Op2	1
Metallaxis	1	Tarantula	1
MUSE	0	DStar	0

(c) Techniques sorted by tournament ranking over means. These rankings are uncorrelated because $p = 0.43$.

Real Faults		Artificial Faults	
Technique	# Worse	Technique	# Worse
Ochiai	2	MUSE	5
Barinel	1	Metallaxis	5
DStar	1	Ochiai	3
Op2	1	Op2	3
Tarantula	1	Barinel	1
Metallaxis	1	Tarantula	1
MUSE	0	DStar	0

(d) Techniques sorted by tournament ranking over medians. These rankings are uncorrelated because $p = 0.55$.

For each of the 10 pairs of techniques that the prior work in table 1 has compared, we performed an ANOVA comparing the two techniques' scores for both real and artificial faults. Table 5 shows the results of these replications.

Notably, we find almost no statistically significant differences between any of the spectrum-based techniques, for either real or artificial faults. In other words, our experimental results suggest that most previously-reported results are not statistically significant. Our experiments were on a

Figure 2: Distributions of *EXAM* scores.

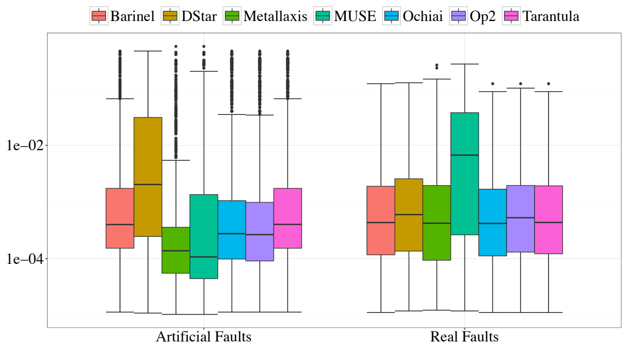

Table 4: For each technique, the p-value for the hypothesis that real faults' *EXAM* scores are the means of their corresponding artificial faults' scores; and the p-value for the hypothesis that the two are independent, and associated Pearson's r.

Technique	p (equal)	p (independent)	r
Ochiai	<0.01	<0.01	0.10
Barinel	<0.01	<0.01	0.11
MUSE	0.50	<0.01	0.36
DStar	<0.01	0.01	0.16
Op2	<0.01	<0.01	0.08
Tarantula	<0.01	<0.01	0.11
Metallaxis	<0.01	<0.01	0.03

Table 5: Previously-reported comparisons (as listed in table 1), and our results for those comparisons.

	Comparison		Agree?	
Citation	Better	Worse	Real	Artificial
[24, 25, 30, 41, 46]	Ochiai	Tarantula	insig.	insig.
[5]	Barinel	Ochiai	insig.	insig.
[5]	Barinel	Tarantula	insig.	insig.
[30]	Op2	Ochiai	insig.	insig.
[29, 30]	Op2	Tarantula	insig.	insig.
[32]	Metallaxis	Ochiai	**no**	**yes**
[24, 41]	DStar	Ochiai	insig.	**no**
[20, 24, 41]	DStar	Tarantula	insig.	**no**
[29]	MUSE	Op2	**no**	**yes**
[29]	MUSE	Tarantula	**no**	**yes**

larger dataset and used consistent implementations of the techniques, which we feel reduces the likelihood of mistaken conclusions.

Our results contradict previous reports that DStar outperforms Ochiai and Tarantula on artificial faults.

However, we confirm previous reports that mutation-based techniques (Metallaxis and MUSE) do better than spectrum-based techniques, but only on artificial faults. On *real* faults, the results are inverted, and the spectrum-based techniques do significantly better than the mutation-based ones. This is likely for the reasons described in section 5.1: mutations can fix artificial faults, and real faults sometimes include unmutatable statements.

Table 6: Techniques sorted by various metrics considering only single-line faults that were not faults of omission, and on their corresponding artificial faults.

Real Faults		Artificial Faults	
Technique	Mean	Technique	Mean
Op2	0.0041347	Metallaxis	0.0117046
Ochiai	0.0041569	Op2	0.0135273
Barinel	0.0041667	Ochiai	0.0136061
Tarantula	0.0042733	Barinel	0.0139500
DStar	0.0086908	Tarantula	0.0139500
Metallaxis	0.0100369	MUSE	0.0175548
MUSE	0.0324617	DStar	0.0288024

(a) Techniques sorted by mean *EXAM* score

Real Faults		Artificial Faults	
Technique	Median	Technique	Median
Metallaxis	0.0003610	Metallaxis	0.0001510
Op2	0.0003735	MUSE	0.0001744
Ochiai	0.0004336	Op2	0.0004352
Barinel	0.0004336	Ochiai	0.0005275
Tarantula	0.0004357	Barinel	0.0009813
DStar	0.0007010	Tarantula	0.0009813
MUSE	0.0033679	DStar	0.0021218

(b) Techniques sorted by median *EXAM* score

Real Faults		Artificial Faults	
Technique	# Worse	Technique	# Worse
Ochiai	1	Ochiai	1
Barinel	1	Barinel	1
DStar	1	MUSE	1
Op2	1	Op2	1
Tarantula	1	Tarantula	1
Metallaxis	1	Metallaxis	1
MUSE	0	DStar	0

(c) Techniques sorted by tournament ranking over means. These rankings are uncorrelated because $p = 0.72$.

Real Faults		Artificial Faults	
Technique	# Worse	Technique	# Worse
Ochiai	2	MUSE	5
Barinel	2	Metallaxis	5
Op2	2	Ochiai	3
Tarantula	1	Op2	3
Metallaxis	1	Barinel	1
MUSE	0	Tarantula	1
DStar	0	DStar	0

(d) Techniques sorted by tournament ranking over medians. These rankings are uncorrelated because $p = 0.98$.

5.4 Does type of fault affect correlation?

Tables 6 to 8 repeat the analysis of RQ1, restricted to each category of real faults described in RQ4. Again, there are no statistically significant relationships between rankings on real and artificial faults.

The ranking of spectrum-based techniques is exactly the same on both real and artificial faults, for single-line faults — which one would expect to be the most similar to mutants — but this is as strong a relationship as we see.

Table 7: Techniques sorted by various metrics considering only multi-line faults that included faults of omission, and on their corresponding artificial faults.

Real Faults		Artificial Faults	
Technique	Mean	Technique	Mean
Barinel	0.0042599	Metallaxis	0.0080336
Tarantula	0.0042599	MUSE	0.0133316
Ochiai	0.0044110	Op2	0.0190395
DStar	0.0056097	Ochiai	0.0190759
Op2	0.0067196	Barinel	0.0193352
Metallaxis	0.0091738	Tarantula	0.0193352
MUSE	0.0358144	DStar	0.0361518

(a) Techniques sorted by mean *EXAM* score

Real Faults		Artificial Faults	
Technique	Median	Technique	Median
Ochiai	0.0002782	MUSE	0.0001053
DStar	0.0002926	Metallaxis	0.0001419
Barinel	0.0003695	Op2	0.0002525
Tarantula	0.0003695	Ochiai	0.0002659
Metallaxis	0.0004169	Barinel	0.0003356
Op2	0.0006020	Tarantula	0.0003356
MUSE	0.0068916	DStar	0.0023385

(b) Techniques sorted by median *EXAM* score

Real Faults		Artificial Faults	
Technique	# Worse	Technique	# Worse
Ochiai	1	Metallaxis	6
Barinel	1	MUSE	5
DStar	1	Ochiai	1
Op2	1	Barinel	1
Tarantula	1	Op2	1
Metallaxis	1	Tarantula	1
MUSE	0	DStar	0

(c) Techniques sorted by tournament ranking over means. These rankings are uncorrelated because $p = 0.31$.

Real Faults		Artificial Faults	
Technique	# Worse	Technique	# Worse
Ochiai	1	MUSE	6
Barinel	1	Metallaxis	5
DStar	1	Ochiai	3
Op2	1	Op2	3
Tarantula	1	Barinel	1
Metallaxis	1	Tarantula	1
MUSE	0	DStar	0

(d) Techniques sorted by tournament ranking over medians. These rankings are uncorrelated because $p = 0.13$.

Table 8: Techniques sorted by various metrics considering only multi-line faults that did not include faults of omission, and on their corresponding artificial faults.

Real Faults		Artificial Faults	
Technique	Mean	Technique	Mean
Op2	0.0025056	Metallaxis	0.0075830
Ochiai	0.0034721	MUSE	0.0110307
Barinel	0.0040578	Op2	0.0114975
Tarantula	0.0040578	Ochiai	0.0115969
DStar	0.0089081	Barinel	0.0122952
Metallaxis	0.0098250	Tarantula	0.0126113
MUSE	0.0232786	DStar	0.0236207

(a) Techniques sorted by mean *EXAM* score

Real Faults		Artificial Faults	
Technique	Median	Technique	Median
Metallaxis	0.0004537	Metallaxis	0.0000718
Op2	0.0005668	MUSE	0.0001268
Ochiai	0.0005963	Op2	0.0001981
Barinel	0.0005963	Ochiai	0.0003170
Tarantula	0.0005963	DStar	0.0005176
DStar	0.0008638	Barinel	0.0005706
MUSE	0.0080034	Tarantula	0.0005706

(b) Techniques sorted by median *EXAM* score

Real Faults		Artificial Faults	
Technique	# Worse	Technique	# Worse
Op2	2	Ochiai	1
Ochiai	1	Barinel	1
Barinel	1	MUSE	1
DStar	1	Op2	1
Tarantula	1	Tarantula	1
Metallaxis	1	Metallaxis	1
MUSE	0	DStar	0

(c) Techniques sorted by tournament ranking over means. These rankings are uncorrelated because $p = 1.00$.

Real Faults		Artificial Faults	
Technique	# Worse	Technique	# Worse
Op2	2	Metallaxis	6
Ochiai	1	MUSE	5
Barinel	1	Op2	4
Tarantula	1	Ochiai	3
Metallaxis	1	Barinel	0
MUSE	0	DStar	0
DStar	0	Tarantula	0

(d) Techniques sorted by tournament ranking over medians. These rankings are uncorrelated because $p = 0.83$.

6. RELATED WORK

6.1 Fault localization techniques

According to a recent survey [42], the most studied and evaluated fault localization techniques are spectrum-based [4, 19, 41] (see Section 3.1.1), slice-based [39, 45], model-based [2, 44], and mutation-based [29, 32] (see Section 3.1.2).

Slice-based techniques [45] use data- and control-flow of a failing execution to identify components that are *not* responsible for triggering a failure and exclude them from a slice. I.e., rather than setting a suspiciousness score to each component as SBFL or MBFL techniques, the aim of slice-based techniques is reduce the number of components that a developer would have to inspect.

Model-based techniques [2, 27] for fault localization observe executions to infer a model of the software under test; they infer which components could be responsible for faulty behavior using, for example, the oracles provided by test cases. Model-based techniques can achieve high diagnostic quality. For instance, by first applying a spectrum-based technique to remove the components that are likely not related to the observed failure, and then applying a model-based technique on the remaining components, Abreu et al. [1] were able to achieve a better diagnose when compare to SBFL techniques. However, the computationally intractable [28] of model-based techniques does not scale to large software programs.

6.2 Using developers to evaluate fault localization techniques

Metrics such as T-score, Expense, or *EXAM* score (see Section 2.1 for more information) were inspired by *perfect fault understanding*: a developer examines statements one-by-one and detects the faulty component when encountering it. Researchers do not believe that programmers actually work this way! Rather, researchers use it as a proxy; they hypothesize that this approach ranks techniques in terms of their true quality[4]. To understand the perfect fault understanding assumption, Parnin and Orso conducted a preliminary user study on debugging effectiveness, involving 34 graduate students [33]. They divided the study participants into two groups, one that used the Tarantula fault localization technique [19] and one that did not. While the results indicate that a fault localization technique helps experienced developers debugging small programs, they do not support the hypothesis that a better ranking significantly affects debugging effectiveness.

More recently, Gouveia et al. [16] also performed a user study to evaluate whether graphical visualizations of the ranking (rather than text [33]) could help developers on debugging. One group used the authors' tool GZoltar [9] and the other group did not. The authors concluded that the group which used the graphical visualizations of the ranking, as provided by GZoltar, were able to pinpoint the faulty component and to spend less time on debugging than the control group.

6.3 Mutants vs. Real Faults

It has been very common to evaluate and compare fault localization techniques using manually-seeded faults (e.g., `Siemens` set [4, 19, 32, 36]), or faulty versions of real-world programs generated by mutating correct versions (e.g., [12, 36, 41]) as a proxy to *real faults*. However, it remains an open question whether results on small artificial faults (whether hand-seeded or automatically-generated) are characteristic of results on real faults.

To the best of our knowledge, previous evaluations of FL techniques on real faults only used one small numerical subject program with simple control flow: `space` [38], in which 35 real faults were detected during development. Those faults were characterized as: logic omitted or incorrect (e.g., missing condition), computation problems (e.g., incorrect equations), incomplete or incorrect interfaces, and data handling problems (e.g., incorrectly access/storage data). In previous studies, `space`'s real faults have been considered alongside artificially inserted faults, but no comparison between the two kinds was done. In contrast, we used larger programs (ranging from 22KLOC to 96KLOC); and we independently evaluated the performance of each FL technique on a higher number of real faults (which are more complex) and artificial faults (see Section 3).

[4]This is exactly like researchers do not believe that all real bugs are mutants. Rather, researchers use mutants as a proxy; they hypothesize that ranking based on mutants gives the same results as ranking based on real faults.

The use of mutants as a replacement for real faults has been investigated in other domains. In particular, in the context of test prioritization, Do et al. [13] concluded from experiments on six Java programs that mutants are better suited than manually seeded faults for studies of prioritization techniques, as small numbers of hand-selected faults may lead to inappropriate assessments of those techniques.

The more general question of whether mutants are representative of real faults has been subject to thorough investigation. While Gopinath et al. [15] found that mutants and real faults are not syntactically similar, several independent studies have provided evidence that mutants and real faults are coupled. DeMillo et al. [11] studied 296 errors in TeX and found simple mutants to be coupled to complex faults. Daran et al. [10] found that the errors caused by 24 mutants on an industrial software program are similar to those of 12 real faults. Andrews et al. [7] compared manually-seeded faults with mutants and concluded that mutants are a good representation of real faults for testing experiments, in contrast to manually-seeded faults. Andrews et al. [8] further evaluated the relation of real faults from the `space` program and 736 mutants using four mutation operators, and again found that mutants are representative of real faults. Just et al. [23] studied the real faults of the Defects4J [22] data set, and identified a positive correlation of mutant detection with real fault detection. However, they also found that 27% of the real faults in Defects4J [22] are not coupled with commonly used mutation operators [18], suggesting a need for stronger mutation operators.

However, Namin et al. [31] studied the same set of programs as previous studies [7], and caution of the substantial external threats to validity when using mutants for experiments. Therefore, it is important to study the impact of the use of mutants for specific types of software engineering experiments, such as fault localization, as conducted in this paper.

7. CONCLUSION

Fault localization techniques' performance has mostly been evaluated using artificial faults (i.e., mutants). As mutants may differ from real faults, previous studies fall short in establishing which techniques are best at finding real faults.

This paper evaluates the performance of 7 previously-studied fault localization techniques. We applied the 7 FL techniques to 251 real faults and 2831 artificial faults, distributed over 5 programs. Our results, in particular the differences between the mean and median *EXAM* scores for MBFL and SBFL techniques, suggest that MBFL techniques are less applicable to real faults, and that artificial faults in general are not useful for predicting which fault localization techniques perform best on real faults.

8. REFERENCES

[1] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund. Refining spectrum-based fault localization rankings. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 409–414, New York, NY, USA, 2009. ACM.

[2] R. Abreu and A. J. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Symposium on Abstraction, Reformulation, and Approximation (SARA)*, volume 9, pages 2–9, 2009.

[3] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[4] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[5] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. Spectrum-based multiple fault localization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 88–99. IEEE, 2009.

[6] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang. Evaluating the accuracy of fault localization techniques. In *ASE*, pages 76–87, Nov. 2009.

[7] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, May 2005.

[8] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Trans. Softw. Eng.*, 32(8):608–624, Aug. 2006.

[9] J. Campos, A. Riboira, A. Perez, and R. Abreu. GZoltar: An Eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 378–381, New York, NY, USA, 2012. ACM.

[10] M. Daran and P. Thévenod-Fosse. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 158–171, New York, NY, USA, 1996. ACM.

[11] R. A. DeMillo and A. P. Mathur. On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis in Detecting Errors in Production Software. techreport SERC-TR-92-P, Purdue University, West Lafayette, Indiana, 1992.

[12] N. DiGiuseppe and J. A. Jones. Fault density, fault types, and spectra-based fault localization. *Empirical Softw. Engg.*, 20(4):928–967, Aug. 2015.

[13] H. Do and G. Rothermel. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Trans. Softw. Eng.*, 32(9):733–752, Sep. 2006.

[14] W. Eric Wong, V. Debroy, and B. Choi. A Family of Code Coverage-based Heuristics for Effective Fault Localization. *Journal of Systems and Software*, 83(2):188–208, Feb. 2010.

[15] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 189–200. IEEE, 2014.

[16] C. Gouveia, J. Campos, and R. Abreu. Using HTML5 visualizations in software fault localization. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, ICSM 2013, Washington, DC, USA, 2013. IEEE Computer Society.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the Effectiveness of Dataflow-and Controlflow-Based Test Adequacy Criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[18] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sep. 2011.

[19] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, pages 273–282, Nov. 2005.

[20] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao. HSFal: Effective Fault Localization Using Hybrid Spectrum of Full Slices and Execution Slices. *Journal of Systems and Software*, 90:3–17, Apr. 2014.

[21] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ISSTA*, pages 433–436, July 2014.

[22] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*, pages 437–440, July 2014. Tool demo.

[23] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, Nov. 2014.

[24] T.-D. B. Le, D. Lo, and F. Thung. Should i follow this fault localization tool's output? *Empirical Softw. Engg.*, 20(5):1237–1274, Oct. 2015.

[25] T.-D. B. Le, F. Thung, and D. Lo. Theory and practice, do they match? A case with spectrum-based fault localization. In *ICSM*, pages 380–383, Sep. 2013.

[26] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *Software Engineering, IEEE Transactions on*, 32(10):831–848, 2006.

[27] W. Mayer and M. Stumptner. Modeling programs with unstructured control flow for debugging. In *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, AI '02, pages 107–118, London, UK, UK, 2002. Springer-Verlag.

[28] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 128–137, Washington, DC, USA, 2008. IEEE Computer Society.

[29] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, pages 153–162, Apr. 2014.

[30] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[31] A. S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International*

Symposium on Software Testing and Analysis, ISSTA '11, pages 342–352, New York, NY, USA, 2011. ACM.

[32] M. Papadakis and Y. Le Traon. Metallaxis-FL: Mutation-based fault localization. *STVR*, 25(5-7):605–628, Aug.–Nov. 2015.

[33] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, July 2011.

[34] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *AADEBUG*, pages 273–276, Sep. 2003.

[35] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 191–201, New York, NY, USA, 2013. ACM.

[36] R. Santelices, J. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 56–66, 2009.

[37] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*, pages 314–324, July 2013.

[38] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 44–, Washington, DC, USA, 1998. IEEE Computer Society.

[39] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[40] E. Wong, T. Wei, Y. Qi, and L. Zhao. A Crosstab-based Statistical Method for Effective Fault Localization. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 42–51, Washington, DC, USA, 2008. IEEE Computer Society.

[41] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar method for effective software fault localization. *IEEE Trans. Reliab.*, 63(1):290–308, Mar. 2014.

[42] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey of software fault localization. *IEEE Transactions on Software Engineering (TSE)*, 2016.

[43] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of Test Set Minimization on Fault Detection Effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[44] F. Wotawa, M. Stumptner, and W. Mayer. 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, pages 746–757, London, UK, UK, 2002. Springer-Verlag.

[45] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

[46] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM.