Evaluating & improving fault localization techniques

Spencer Pearson*, José Campos**, René Just†, Gordon Fraser**, Rui Abreu‡, Michael D. Ernst*, Deric Pang*, Benjamin Keller*

*U. of Washington, USA **U. of Sheffield, UK †U. of Massachusetts, USA ‡Palo Alto Research Center, USA
U. of Porto, Portugal

suspense@cs.washington.edu, jose.campos@sheffield.ac.uk, rjust@cs.umass.edu, gordon.fraser@sheffield.ac.uk, rui@computer.org, mernst@cs.washington.edu, dericp@cs.washington.edu, bjkeller@cs.washington.edu

***Abstract*—A fault localization technique takes as input a faulty program, and it 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; they score the technique based on where in its output list the defective code 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, 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 performed a replication study to evaluate 10 claims in the literature that compared fault localization techniques. We used 2273 artificial faults in 5 real-world programs. Our results refute 3 of the previous claims. Then, we evaluated the same 10 claims, using 297 *real* faults from the 5 programs. Every previous result was refuted or was statistically insignificant. In other words, our experiments show that artificial faults are not useful for predicting which fault localization techniques perform best on real faults.

In light of these results, we identified a design space that includes many previously-studied fault localization techniques as well as hundreds of new techniques. We experimentally determined which factors in the design space are most important. Then, we extended it with new techniques. Several of our novel techniques outperform all existing techniques, notably in terms of ranking defective code in the top-5 or top-10 reports.

I. 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 — that is, 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]. Another use is to focus a defect repair tool on the parts of the code that are most likely to be buggy.

Dozens of fault localization techniques have been proposed [47]. 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. Although some real faults (mostly 35 faults in the single small numerical program space [43]) have been used in previous comparisons [47] of fault localization techniques, the vast majority of faults used in such comparisons are fake faults, mostly 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`. The artificial faults were sometimes mutants automatically created by a tool [28], [29], [52], and were sometimes mutant-like 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) [46], [48] or by researchers (in the Siemens set of 7 tiny programs) [17].

Artificial faults such as mutants differ from real faults in many respects, including their size, their distribution in code, and their difficulty of being detected by tests [22]. 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 should 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 [39].

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, 297 real faults, and 2273 artificial faults. Our results do *not* indicate that techniques that do well on artificial faults also do well on real faults.

The contributions of this paper include:

- A replication study that repeats and extends previous experiments, comparing 7 fault localization techniques on artificial faults. We mitigated threats to internal validity by re-implementing all the techniques in a single infrastructure and using the same experimental scripts, mutations, and other experimental variables. Our results confirm 70% of previously-reported comparisons (such as "Ochiai is better than Tarantula" [28], [29], [34], [46], [52]) and refute 30%.
- A new study that compares the 7 fault localization tech-

niques on *real* faults. The ranking does not agree with any previous results from artificial faults! 40% of the previous results are reversed; for example, Metallaxis is better than Ochiai on artificial faults [36], but Ochiai is better than Metallaxis on real faults. The other 60% of the results are statistically insignificant; for example, DStar is better than Tarantula on artificial faults [20], [28], [46], but on real faults there is no statistically significant difference between the two techniques. These results indicate that artificial faults (e.g., mutants) are not an adequate substitute for real faults, for the task of evaluating a fault localization tool.

- An explication of the design space of fault localization techniques. Previous work had made different, sometimes undocumented, choices for factors other than the formula. We exhaustively evaluated all these factors. We found that formula, which most papers have exclusively focused on, is not the most important factor. We also added new factors to the design space, thereby creating new hybrid fault localization techniques that combine the best of previous techniques.
- An evaluation of all the FL techniques generated by the design space, with respect to how well they localize real faults. We found new techniques that are statistically significantly better than any previous technique, though with small effect sizes. More importantly, they do much better in terms of including the correct answer (the actual faulty line) within the top-5 or top-10 lines of their output. Our results indicate how to make the most of current approaches, and they indicate that significant advances in fault localization will come from focusing on different issues than in the past.
- Our methodology addresses multi-line faults, faults of omission, and other real-world issues, both in the design of FL techniques and in the experimental protocol for evaluating them.

The rest of the paper is structured as follows. Section II describes the traditional approaches that have been commonly used to evaluate fault localization techniques and our extensions to make them applicable to real faults. Section III describes the subjects of investigation, including the studied fault localization techniques and the used programs, real faults, and test suites. Sections IV–VII describe our four empirical studies in detail. More specifically, Section IV reports on our replication study using artificial faults, Section V reports on our replication study using real faults, Section VI defines and explores a design space for fault localization techniques, and Section VII proposes and evaluates new fault localization techniques. Section VIII discusses threats to validity regarding all four studies. Section IX describes related work, Section X highlights lessons learned, and Section XI concludes. An appendix contains tables and figures that support claims made in the body of the paper.

II. Evaluating Fault Localization

Many studies have evaluated and compared FL techniques [3]–[6], [19], [20], [28], [29], [33], [34], [36], [41], [46], [52]. Table I summarizes these studies in terms of the programs used, the number of real and artificial faults considered, and the resulting ranking of techniques. The majority of studies revolve around the same set of programs and use largely artificial faults. One of this paper's aims is to investigate to what extent the use of artificial faults influences the findings of such experiments. This section explains how a fault localization technique's output can be evaluated.

A. 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 [38], [42]: 1) run the FL technique to compute the sorted list of suspicious lines; 2) let n be the rank of d in the list; 3) use one of the metrics proposed in the literature to evaluate the effectiveness of a FL technique, e.g., LIL [33], T-score [30], Expense [19], or *EXAM* score [45].[1] For concreteness this paper uses *EXAM* score, which is the most popular metric, but our results generalize to the others.

LIL [33] measures the effectiveness of a fault localization technique for automated program repair. T-score [30] computes the percentage of components that a developer *would not have* to inspect before finding the first faulty one. By contrast, Expense [19] and *EXAM* score [45] 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 N is the total number of statements in the program for *EXAM* score. The score ranges between 0 and 1, and smaller numbers are better.

B. Extensions to fault localization evaluation

The standard technique for evaluating fault localization, described in section II-A, handles defects that consist of a single change to an executable statement in the program, as is the case for mutants. In order to evaluate fault localization on real faults, we had 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.

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

[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.

TABLE I
SELECTED FAULT-LOCALIZATION STUDIES. OUR RESULTS (BOTTOM) UPHOLD PREVIOUS RESULTS ON ARTIFICIAL FAULTS BUT NOT ON REAL FAULTS.

Ref.	Lang.	Ranking (from best to worst)	Projects	kLOC	Artif. faults	Real faults
[19]	C	Tarantula	Siemens	3	122👤	-
[4]	C	Ochiai, Tarantula	Siemens	3	120👤	-
[3]	C	Ochiai, Tarantula	Siemens, space	12	128👤	34
[5]	C	Barinel, Ochiai, Tarantula	Siemens, space, gzip, sed	31	141👤	38
[6]	C	Tarantula	Concordance	2	200♣	13
[34]	C	Op2, Ochiai, Tarantula	Siemens, space	12	132👤	32
[36]	C	Metallaxis, Ochiai	Siemens, space, flex, grep, gzip	45	859 👤,♣	12
[29]	C	Ochiai, Tarantula	Siemens, space, NanoXML, XML-Security	41	164👤	35
[46]	C	DStar, Ochiai, Tarantula	Siemens, space, ant, flex, grep, gzip, make, sed, Unix	155	436👤	34
[33]	C	MUSE, Op2, Ochiai	space, flex, grep, gzip, sed	54	11👤	3
[52]	Java	Ochiai, Tarantula	JExel, JParsec, Jaxen, Commons Codec, Commons Lang, Joda-Time	108	1800♣	-
[20]	C/Java	DStar, Tarantula	printtokens, printtokens2, schedule, schedule2, totinfo, Jtcas, Sorting, NanoXML, XML-Security	32	104👤	-
[28]	C	DStar, Ochiai, Tarantula	Siemens, space, NanoXML, XML-Security	41	165👤	35
this	Java	Metallaxis, Op2, DStar, Ochiai, Barinel, Tarantula, MUSE	JFreeChart, Closure, Commons Lang, Commons Math, Joda-Time	321	2273♣	-
this	Java	{DStar ≈ Ochiai ≈ Barinel ≈ Tarantula}, Op2, Metallaxis, MUSE	JFreeChart, Closure, Commons Lang, Commons Math, Joda-Time	321	-	297

👤 represents manually-seeded artificial faults, and ♣ represents mutation-based artificial faults. The Siemens set is printtokens, printtokens2, replace, schedule, schedule2, tcas, and totinfo. The Unix set is Cal, Checkeq, Col, Comm, Crypt, Look, Sort, Spline, Tr, and Uniq.

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 [42], [47].

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. In our experiments, there were never duplicate reports for a single statement, but it would be possible to remove all but the first one.

3) Multi-statement faults: Existing evaluation metrics, such as the *EXAM* score, are limited as they assume that a single program statement is defective. This assumption holds for mutants and often other artificial faults. However, 76% of real-world bug fixes span multiple statements [21]. To evaluate a fault localization technique on a multi-statement fault, we need to define when the fault is considered to be localized: is it sufficient for the technique to propose any defective statement, or must it propose all defective statements? Rather than making assumptions about a particular *debugging scenario* (e.g., experienced developer vs. novice developer vs. automated repair tool), our study evaluates the fault localization techniques for three debugging scenarios:

1) **Best-case:** Any one defective statement needs to be localized to understand and repair the defect.
2) **Worst-case:** All defective statements need to be localized to understand and repair the defect.
3) **Average-case:** 50% of the defective statements need to be localized to understand and repair the defect.

Note that these debugging scenarios are equivalent for single-statement faults.

```
1  public boolean updateState(State s) {
2+   if (!flag)
3+     return false;
4    setState(s);                // goal report
5    return true;
6  }
```

```
1  public boolean updateState(State s) {
2+   if (flag)
3      setState(s);              // goal report
4    return true;
5  }
```

```
1  public boolean updateState(State s) {
2+   if (flag) {
3      setState(s);  // goal report for "if (flag)"
4      otherStatement();
5      return true;  // goal report for "return false"
6+   }
7+   return false;
8  }
```

Fig. 1. Faults of omission and goal statements for reporting them. For the third example, line 8 (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.

4) Faults of omission: In 19% of cases [21], 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. Previous studies on the effectiveness of fault localization have not reported whether and how this issue was addressed. Wong et al. [14] suggested that analyzing the code around statements with a 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 the line in the source code the programmer needs to make a change, such as the line following where the programmer should insert the code. Therefore, when a bug fix involves inserting code, 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. To avoid disadvantaging such techniques, we also 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 Closure-15 in Defects4J [21]:

```
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* locations at which a code block could be inserted to fix the defect (lines 1, 3, and 5 in the example above). We consider a fault localization technique to identify an omitted statement as soon any candidate location appears in the FL technique's output. (A multi-statement fault may require localizing multiple omitted or fixed statements.)

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 [21], 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. Some faulty, executable statements are not mutatable due to compiler restrictions. For example, deleting or incorrectly moving a `break` or `return` statement might cause compilation errors due to the violation of control-flow properties enforced by the compiler. In the Defects4J database of real-world faults [21], 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, by adding any missing statement at the end of the ranking. 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 III-B2) 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, which is the expected rank of that statement, if the remaining 100 statements were appended to the ranking in arbitrary order.

6) Multiple defects: Large real-world programs, like those in Defects4J, almost always contain multiple defects coexisting with each other. However, no action is needed to correct for this when performing fault localization, as long as the failing tests only reveal one of these defects (as is the case in Defects4J).

III. Subjects of Investigation

A. Fault localization techniques

This paper evaluates 2 families of fault localization techniques: spectrum-based fault localization (SBFL techniques for short) [4], [19], [34], [46], which is the most studied and evaluated FL technique; and mutation-based fault localization (MBFL technique for short), which is reported to significantly outperform SBFL techniques [33], [36]. A survey paper lists other types of fault localization techniques [47].

Most fault localization techniques, including all that we examine in this paper, yield a ranked list of program statements sorted by the suspiciousness score $S(s)$ of the statement s. A high suspiciousness score means the statement is more likely to be defective and the root cause of the failures.

1) Spectrum-based FL techniques: Spectrum-based fault localization techniques [4], [19], [34], [46] depend on statement execution frequencies. The more often a statement is executed by failing tests, and the less often it is executed by passing tests, the more suspicious the statement is considered.

This paper considers 5 of the best-studied SBFL formulas [47]. In the following, let $totalpassed$ be the number of passed test cases and $passed(s)$ be the number of those that executed statement s (similarly for $totalfailed$ and $failed(s)$).

Tarantula[19]:	$S(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed+passed(s)/totalpassed}$
Ochiai[3]:	$S(s) = \frac{failed(s)}{\sqrt{totalfailed\cdot(failed(s)+passed(s))}}$
Op2[34]:	$S(s) = failed(s) - \frac{passed(s)}{totalpassed+1}$
Barinel[5]:	$S(s) = 1 - \frac{passed(s)}{passed(s)+failed(s)}$
DStar† [46]:	$S(s) = \frac{failed(s)^*}{passed(s)+(totalfailed-failed(s))}$

†variable $* > 0$. We chose $* = 2$, as that value is most thoroughly explored by Wong et al. [46].

2) Mutation-based FL techniques: Mutation-based fault localization techniques [33], [36] 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 assign suspiciousnesses to 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. Our study considered two well-known MBFL techniques: MUSE [33] and Metallaxis [36]. Each one generates a set of mutants $mut(s)$ for each statement s, assigns each mutant a suspiciousness $M(m)$, and aggregates the $M(m)$ to yield a statement suspiciousness score $S(s)$.

MUSE [33] sets $M(m) = \mathit{failed}(m) - \frac{\mathit{f2p}}{\mathit{p2f}} \cdot \mathit{passed}(m)$ 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. MUSE sets $S(s) = \text{avg}_{m \in mut(s)} M(m)$. The MUSE approach is based on the assumption that mutating faulty statements is more likely than mutating correct statements to cause failing tests to pass, and mutating correct statements is more likely than mutating faulty statements to cause passing tests to fail.

Metallaxis [36] uses the Ochiai formula for the suspiciousness of each mutant:

$$M(m) = \frac{failed(m)}{\sqrt{totalfailed \cdot (failed(m) + passed(m))}}$$

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. The suspiciousness of statement s is $S(s) = \max_{m \in mut(s)} M(m)$.

Since MBFL requires running the test suite once per possible mutant, it is much more expensive than SBFL: even with the optimizations desribed in section X-C that reduced the runtime by more than an order of magnitude, running every test necessary to compute every MBFL technique's score on all 2502 faults took over 25000 CPU-hours.

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. We collected coverage data using an improved version of GZoltar [9]. We collected mutation analysis data from a tool built upon the Major mutation framework [25].

B. Programs

We used the programs in the Defects4J [21] dataset (v1.0.1), which consists of 357 real faults from 5 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.

1) 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 independently minimized the patches, and agreed on a patch that preserved the spirit of the programmer's fix.

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.

2) Fault-relevant classes: To reduce CPU costs, we applied each fault localization technique only to the *fault-relevant classes*. A fault-relevant class for a defect is any class that is loaded by any fault-triggering test for that defect. This optimization is sound, and a programmer could use it with little or no effort when debugging a failure. We did not use slicing, impact analysis, or other approaches to further localize or isolate the defective code.

TABLE II
AVERAGE NUMBER OF TEST CASES PER PROJECT AND PER TYPE OF FAULT.

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

For each fault that we localized, we obtained the set of fault-relevant classes as follows. First, we used Defects4J's infrastructure to independently execute each fault-triggering test and monitor the class loader. This yielded a set of all loaded classes of which we retained only the project-related classes (ignoring classes from libraries and the runtime environment). This set of fault-relevant classes is a sound approximation of the set of classes that contain a defect.

C. 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.

Table II shows, per project, the average number of test cases that load any class containing a defective statement, and how many of them pass or fail with that defect. Artificial and real faults have similar numbers of passing and failing tests, confirming that our results are not confounded by different effective test suite sizes for real and artificial faults.

IV. Replication: Artificial Faults

One goal of our work is to repeat previous evaluations of fault localization techniques on artificial faults, using shared infrastructure between all of the techniques to ensure that differences reflect true differences in the techniques and not differences in implementation. This section describes the techniques and faults we studied, our methodology for assessing them, and the results of the comparison.

A. Methodology

1) Research questions:

RQ 1: Which FL techniques are significantly better than which others, on artificial faults?

RQ 2: Do the answers to RQ1 agree with previous results? (For example, "Ochiai outperforms Tarantula" [4], "Metallaxis outperforms Ochiai" [36].)

2) Data: artificial faults: We used the Major tool to generate 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. To reduce CPU costs, 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.

In more detail: each real fault in Defects4J is associated with a faulty and a 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 statements modified or deleted by this patch the *fixed statements* of the real fault. We generated artificial faults by mutating only the fixed statements. This means that if there are no fixed statements (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, 14% of artificial faults could not be exposed by any test case; we discarded these, as well as the 30 real faults for which no artificial faults caused any test failures.

Overall, the output of this process was a set of 2273 artificial faults, corresponding to 229 different real faults, each artificial fault existing in a fixed statement of the corresponding real fault and detectable by the same developer-written test suite. We computed the *EXAM* score for each artificial fault and FL technique.

3) Experimental Design: We answered our research questions through the following analyses:

RQ 1: Which FL techniques are significantly better than which others, on artificial faults? We used three independent, complementary evaluation metrics to rank these 7 FL techniques from best to worst:

1) mean *EXAM* score across all artificial faults.
2) tournament ranking: comparing the sets of *EXAM* scores of each pair of techniques, awarding 1 point to the winner if it is statistically significantly better, and ranking by number of points.
3) mean rank among FL techniques (FLT rank): using each fault to rank the techniques from 1 to 7, and averaging across all artificial faults.

RQ 2: Do the answers to RQ1 agree with previous results? For each pair of techniques that prior work has compared (see table I), we determined whether the two techniques' distributions of *EXAM* scores are significantly different.

For all statistical comparisons between any two techniques in this paper, we performed a paired t-test for two reasons. First, our experiments have a matched pairs design — fault localization results are grouped per defect. Second, while the exam scores are not normally distributed, the differences between the exam scores of any two techniques are close to being normally distributed. Given that we have a large sample size and no serious violation of the normality assumption, we chose the t-test for its statistical power.

B. Results

1) Best FL technique on artificial faults: The left-hand columns in table III show the FL technique rankings produced by our three metrics. According to our tournament ranking, Metallaxis is the best technique on artificial faults, being statistically significantly better than all other techniques.

All three metrics we used are perfectly consistent with each other, except that MUSE does best by mean FLT rank and worst by the other two metrics. This is because its distribution of scores is bimodal: as fig. 2 shows, although it usually does better than other techniques, it often does much worse. Whether it does better or worse is tightly correlated with whether a fault is caused by a "reversible" mutant (a mutant that can be canceled by a second mutant, such as `a&&b` $\rightarrow$ `a||b`, but not `a&&b` $\rightarrow$ `true`). Recall that MUSE assumes that mutating a faulty statement will sometimes cause a failing test to pass. This is why it does well on artificial faults caused by reversible mutants: there exists a mutant which will repair the fault, thus causing all failing tests to pass. This mutant will be located in the faulty line, and it will achieve the maximum possible suspiciousness, putting that line near the top of the ranking and explaining MUSE's excellent performance on the majority of artificial faults. Irreversible mutants are responsible for the cluster of large scores in fig. 2, explaining why MUSE is best by mean FLT rank but worst by mean *EXAM* score.

MUSE's success is so tightly connected to the fault's reversibility because of its very restrictive definition for whether a failing test kills a mutant (the mutant must cause the test to pass). This criterion is very rarely met, *except* for when a mutant reverses an artificial fault. If MUSE used a different kill-definition, it would lose its pinpoint accuracy on reversible faults, but it would also have fewer large outlier scores on irreversible faults, as shown in fig. 4.

As shown in fig. 2, MBFL techniques very often rank the artificially-faulty line in the top 5. One reason for this is that most artificial faults we generate are caused by "reversible" mutants: mutants which can be exactly cancelled by inserting a second mutant (e.g., `a+b`$\rightarrow$`a-b`$\rightarrow$`a+b`). Reversible artificial faults guarantee that a mutant in the faulty line will fix every failing test, and so be guaranteed a very high suspiciousness score.

2) Agreement with previous results: For each of the 10 pairs of techniques that the prior work in table I has compared, we performed a two-tailed t-test comparing the two techniques' scores for artificial faults. The left column of table IV shows the results of prior comparisons, and the middle columns show our results. Notable features include:

Small effect sizes. All of these pairs of techniques have *statistically* significant differences: the "agree?" column of table IV is unparenthesized. However, the practical differences (that is, effect sizes) are all small or negligible: the "d" column is parenthesized. As seen in fig. 2, all spectrum-based techniques (except Tarantula) are nearly indistinguishable. We only see statistical significance because of our large number of artificial faults.

Consistency with prior SBFL-SBFL comparisons. We agree with all previous comparisons between SBFL techniques, except the claim that Barinel outperforms Ochiai, which we contradict. The difference may be due to our different set of faults, or differences between our implementation and that in [5].

Disagreement with prior SBFL-MUSE comparisons. Prior comparisons between MUSE and SBFL techniques found MUSE to be superior. We disagree: although MUSE does better on many artificial faults, it does much worse on others. Overall, we find that the differences are practically insignificant.

TABLE III
FAULT LOCALIZATION TECHNIQUES SORTED BY AVERAGE PERFORMANCE.

Artificial Faults			Real Faults		
Technique	*EXAM*	# Worse	Technique	*EXAM*	# Worse
Metallaxis	0.0197	6	DStar	0.0443	3
Op2	0.0437	5	Ochiai	0.0445	3
DStar	0.0442	4	Barinel	0.0453	2
Ochiai	0.0448	3	Tarantula	0.0477	2
Barinel	0.0503	1	Op2	0.0527	2
Tarantula	0.0512	0	Metallaxis	0.0768	1
MUSE	0.0574	0	MUSE	0.2186	0

(a) Techniques sorted by mean *EXAM* score or tournament ranking.

Artificial Faults		Real Faults	
Technique	FLT rank	Technique	FLT rank
MUSE	2.86	Metallaxis	3.35
Metallaxis	2.93	DStar	3.78
Op2	3.81	Ochiai	3.81
DStar	3.96	Op2	3.89
Ochiai	4.08	Barinel	4.07
Barinel	5.18	Tarantula	4.11
Tarantula	5.18	MUSE	4.98

(b) Techniques sorted by mean FLT rank

V. Replication: Real Faults

A. Subjects

We evaluated the same techniques, programs, and test suites as described in section III, except that we evaluated each technique on 229 *real* faults provided by Defects4J, rather than on 2273 artificial faults.

B. Methodology

Our methodology here is also exactly like that described in section IV-A, except evaluated on real faults, to answer:

RQ 3: Which FL techniques are significantly better than which others, on real faults?
RQ 4: Do the answers to RQ3 agree with previous results?

C. Results

1) Best FL techniques on real faults: The right-hand columns in the top part of table III show the FL technique rankings for real faults produced by either the mean *EXAM* score metric or the tournament ranking metric. The mean FLT rank (bottom part of table III) gives a similar ranking, except that Metallaxis ranks first instead of nearly last. Metallaxis usually has slightly higher scores than any other technique (as shown in fig. 2), giving it a good FLT rank, but it also has more extreme outliers than SBFL techniques, greatly damaging its mean *EXAM* score.

2) Agreement with previous results: The right-hand columns of table IV compare our results on real faults to the results of the studies we replicated. Notably:

Insignificant differences between SBFL techniques. Once again, there are no large effect sizes (column "d" is parenthesized).

TABLE IV
PREVIOUSLY-REPORTED COMPARISONS (AS LISTED IN TABLE I), AND OUR RESULTS FOR THOSE COMPARISONS. THE COLUMN "95% CI" GIVES THE CONFIDENCE INTERVAL FOR THE DIFFERENCE IN MEANS AND d GIVES THE EFFECT SIZE (COHEN'S D). THE COLUMN "(B – EQ – W)" GIVES THE COUNTS FOR: PER DEFECT, WAS THE WINNER BETTER, EQUAL TO, OR WORSE COMPARED TO THE LOSER, IGNORING THE MAGNITUDE OF THE DIFFERENCE.

Previous comparisons	**Our study on artificial faults**				**Our study on real faults**			
Winner > loser	agree?	d (eff. size)	95% CI	(b – eq – w)	agree?	d (eff. size)	95% CI	(b – eq – w)
Ochiai > Tarantula [28], [29], [34], [46], [52]	**yes**	(-0.25)	[-0.007, -0.005]	(997–1275–1)	*(insig.)*	*(-0.11)*	[-0.007, 0.001]	(37–184–8)
Barinel > Ochiai [5]	**no**	(0.26)	[0.005, 0.006]	(1–1277–995)	*(insig.)*	*(0.06)*	[-0.001, 0.003]	(8–186–35)
Barinel > Tarantula [5]	**yes**	*(-0.06)*	[-0.001, -0.000]	(8–2265–0)	*(insig.)*	*(-0.09)*	[-0.006, 0.001]	(2–227–0)
Op2 > Ochiai [34]	**yes**	*(-0.13)*	[-0.001, -0.001]	(314–1955–4)	no	*(0.14)*	[0.001, 0.016]	(27–178–24)
Op2 > Tarantula [33], [34]	**yes**	(-0.26)	[-0.009, -0.006]	(1034–1235–4)	*(insig.)*	*(0.08)*	[-0.004, 0.014]	(38–166–25)
DStar > Ochiai [28], [46]	**yes**	*(-0.14)*	[-0.001, -0.000]	(205–2068–0)	*(insig.)*	*(-0.02)*	[-0.002, 0.001]	(16–205–8)
DStar > Tarantula [20], [28], [46]	**yes**	(-0.25)	[-0.008, -0.006]	(1005–1267–1)	*(insig.)*	*(-0.11)*	[-0.007, 0.001]	(37–182–10)
Metallaxis > Ochiai [36]	**yes**	*(-0.16)*	[-0.032, -0.019]	(1509–192–572)	**no**	(0.2)	[0.012, 0.053]	(125–17–87)
MUSE > Op2 [33]	**no**	*(0.08)*	[0.006, 0.021]	(1502–106–665)	**no**	**0.81**	[0.139, 0.193]	(77–2–150)
MUSE > Tarantula [33]	*(insig.)*	*(0.03)*	[-0.001, 0.014]	(1659–74–540)	**no**	**0.86**	[0.145, 0.197]	(75–2–152)

Emphasis on whether our study *agrees* indicates p-value: **p<0.01**, p<0.05, *(p≥0.05)*.
Emphasis on Cohen's d indicates effect size: **large**, medium, (small), *(negligible)*.

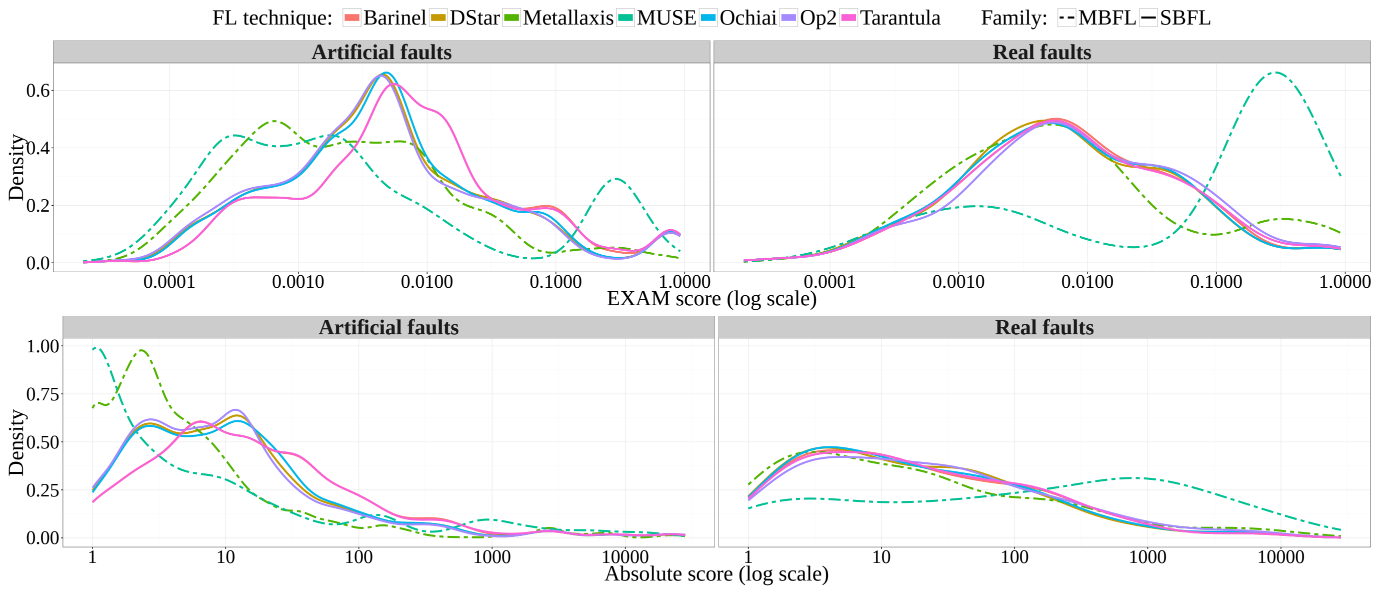

Fig. 2. Distributions of *EXAM* and absolute scores for all FL techniques, considering the best-case debugging scenario and artificial vs. real faults. The absolute score is the first location of any defective statement in the suspiciousness ranking of program statements, computed by a fault localization technique.

Practical significance: MUSE performs poorly. The only practically significant differences show that MUSE performs poorly on real faults (see fig. 2). This is due to almost no real faults being reversible by a single mutant (see section IV-B1).

Independence of debugging scenario. Our conclusions hold in all debugging scenarios, as shown in table XIII in the appendix.

D. Comparison to Artificial Faults

1) Relative Performances: The most important feature of tables III and IV is that ***there is no significant relationship between the results for real and artificial faults.*** This means that artificial faults are not useful for the purpose of determining which FL technique is best at localizing mistakes that programmers actually make.

Another notable feature is that while Metallaxis performs best on artificial faults, it does worse than spectrum-based techniques on real faults. One reason for this may be that 10% of real-world faults involve 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 the best mean FLT rank but one of the worst mean *EXAM* scores.

These findings are again independent of the debugging scenario as shown in table XIII in the appendix.

VI. EXPLORING A DESIGN SPACE

To better understand these differences in performance and their causes, we developed a design space that encompasses all of these techniques, and evaluated the techniques in that space on our set of real faults.

A. Subjects

All 7 of the techniques described in section III-A have the same basic structure:

1) for each program element (i.e., statements or mutants), count the number of passing/failing tests that interact with (i.e., execute or kill) that element;
2) calculate a suspiciousness for each element, by applying a formula to the numbers of passing/failing tests that interact;

3) if necessary, group those elements by statement, and aggregate across the elements' suspiciousnesses to compute the statement's suspiciousness;
4) rank statements by their suspiciousness.

We developed a taxonomy for describing any of these techniques in terms of 4 different parameters:

Formula: the formula used to compute elements' suspiciousness values (e.g., Ochiai)

Total Definition: the method for weighting passing/failing test interactions in the formula

Kill Definition: what it means for a test to *interact* with an element (i.e., coverage for SBFL, killing for MBFL)

Aggregate Definition: for MBFL, the way of aggregating elements' suspiciousness by statement (e.g., max, average)

These parameters are described in more detail below. For SBFL techniques, the "elements" are simply statements: a test interacts with a statement by executing it, and no aggregation of elements by statement is necessary, so the only two relevant parameters are formula and total definition.

The following subsections detail the possible values for each of these parameters. By taking all sensible combinations of them, we arrive at a design space containing 156 techniques.

1) Formula: We consider the formulas for the SBFL techniques Tarantula, Ochiai, DStar, Barinel, and Op2, as well as the formula used by MUSE, which can be cast as

$$S(s) = \mathit{failed}(s) - \frac{\mathit{totalfailed}}{\mathit{totalpassed}} \cdot \mathit{passed}(s) \ .$$

When combined with the appropriate values of the other parameters, this formula produces MUSE's statement-ranking.

2) Total Definition: Almost all of the prior FL techniques make use of *totalpassed* and *totalfailed* in their suspiciousness formulas, representing the numbers of passing/failing tests. MUSE, though (recall from section III-A2), instead refers to *p2f* and *f2p*, representing the number of *mutants killed by* passing/failing tests. Motivated by the resemblance between these quantities, we introduced a parameter that determines whether *totalpassed*, in the FL technique's formula, refers to the number of *tests* or the number of *elements interacted with* by the tests (and similarly for *totalfailed*).

3) Kill Definition: For SBFL there is one clear definition for whether a test interacts with a statement: coverage, or executing the statement. For MBFL, the definition of whether a test "kills" a mutant is not firmly established. MUSE requires that the mutant change whether the test passes or fails, while Metallaxis merely requires that the mutant cause any change to the test's output (for example, change the message of the error thrown by a failing test). We used the following framework to describe the spectrum of possible definitions.

A test kills a mutant if it changes the test outcome — more specifically, if it changes the outcome's *equivalence class*. There are 6 ways to define the equivalence classes. All of them define one class for "pass," one class for "timeout," one class for "JVM crash," and several classes for "exception" (including `AssertionError`). The 6 definitions differ in how they partition exceptions:

1) **exact**: exceptions with the exactame stack trace are equivalent;
2) **type+fields+location**: exceptions with the same type, message, and location are equivalent;
3) **type+fields**: exceptions with the same type and same message are equivalent;
4) **type**: exceptions with the same type are equivalent;
5) **all**: all exceptions are equivalent;
6) **passfail**: all exceptions are equivalent to one another and to the "time out" and "crash" classes (so there are only two possible equivalence classes: "pass" and "fail").

Metallaxis uses the "exact" definition. MUSE uses the "passfail" definition.

4) Aggregate Definition: For MBFL, an aggregate statement suspiciousness $S(s)$ is computed from the suspiciousnesses of individual mutants by taking either the average (like Metallaxis) or the maximum (like MUSE). Unmutatable statements are not assigned any suspiciousness, and therefore do not appear in the technique's ranking. (Approximately 10% of Defects4J's faults contain at least one unmutatable faulty line. This causes MBFL to do quite poorly when its goal is to find the position of *all* faulty lines (the worst-case debugging scenario) for these faults.)

B. Methodology

RQ 5: Which technique in this design space performs best on real faults? By how much is it better than other techniques?

RQ 6: What are the most significant design decisions for a FL technique?

1) Experimental Design:

RQ 5: For each of our evaluation metrics (*EXAM* score, FLT rank) and debugging scenarios (best-case, average-case, worst-case), we identified the technique that performed best, averaged across all real faults.
To quantify how often these techniques significantly outperform others, we performed pairwise comparisons between them and each of the other 155 techniques, using a paired t-test.

RQ 6: We performed an analysis of variance to determine the influence of the 4 design space parameters, the debugging scenario, and the defect on the *EXAM* score. In other words, we compute how much variance in the *EXAM* score is explained by each factor.

C. Results

1) What is the best fault localization technique?: For the best-case debugging scenario, DStar has the smallest mean *EXAM* score (see Table V); DStar is statistically significantly better than almost every other technique in the design space (see table VII), and its score is almost twice as good as the best MBFL technique, which closely resembles Metallaxis. For the other two debugging scenarios, Barinel is best, when instantiated with the "number of statements covered" definition of *totalpassed* and *totalfailed.*

Judged by mean *EXAM* score, all 12 SBFL techniques are better than the best MBFL technique. However, judged by mean FLT rank (shown in table VI), this reverses, and many MBFL techniques are better than any SBFL technique. As

TABLE V
BEST FL TECHNIQUES PER FAMILY ACCORDING TO MEAN *EXAM* SCORE.

	Family	Formula	Total def.	Kill def.	Agg. def	*EXAM* score
best-case debugging scenario (localize any defective statement)						
1	SBFL	dstar2	tests	–	–	0.041
13	MBFL	ochiai	elements	exact	max	0.077
worst-case debugging scenario (localize all defective statements)						
1	SBFL	barinel	elements	–	–	0.202
13	MBFL	ochiai	elements	exact	max	0.248
average-case debugging scenario (localize 50% of the defective statements)						
1	SBFL	barinel	elements	–	–	0.095
13	MBFL	barinel	elements	exact	max	0.134

TABLE VI
BEST FL TECHNIQUES PER FAMILY ACCORDING TO THE MEAN FLT RANK.

	Family	Formula	Total def.	Kill def.	Agg. def.	Rank
best-case debugging scenario (localize any defective statement)						
1	MBFL	muse	elements	exact	avg	72.8
49	SBFL	dstar2	tests	–	–	79.7
worst-case debugging scenario (localize all defective statements)						
1	MBFL	muse	elements	exact	avg	71.1
117	SBFL	dstar2	tests	–	–	85.9
average-case debugging scenario (localize 50% of the defective statements)						
1	MBFL	muse	elements	exact	max	68.3
25	SBFL	ochiai	elements	–	–	76.1

seen in table VII, the best MBFL technique by FLT rank uses MUSE's formula, total-definition, and aggregation definition, but Metallaxis's kill-definition. (Recall from section IV-B that MUSE's kill-definition was what tied its performance to fault reversibility: using a different kill-definition damages its performance on reversible faults, but as shown in fig. 4, makes it much better on real faults.)

As table VII shows, DStar is statistically significantly better than all but six techniques in the design space, and the best MBFL technique is statistically significantly better than about 80% of the design space.

DStar is the best fault localization technique in the design space. However, it is statistically indistinguishable from six other SBFL techniques, including Ochiai, Tarantula, and Barinel.

This pattern remains true if we re-run the analysis on all 357 faults in Defects4J, rather than the 297 for which we were also able to perform mutation analyses.

2) Which parameters matter in the design of a FL technique?: We analyzed the influence of the 4 different parameters on the *EXAM* score. Table VIII shows the results, indicating that all factors (including all FL technique parameters, as well as the defect and debugging scenario) have a significant effect on the *EXAM* score.

It is unsurprising that most of the variance in scores ("sum of squares" column) is accounted for by *which defect* is being localized: some faults are easy to localize and some are difficult.

Interestingly, although prior studies have mostly focused on the formula and neglected other factors, we find that the formula has relatively little effect on how well a FL technique performs.

TABLE VII
PAIR-WISE COMPARISON OF THE BEST TECHNIQUE PER EVALUATION METRIC WITH ALL OTHER TECHNIQUES IN THE DESIGN SPACE. "BETTER" GIVES THE NUMBER OF COMPARISONS FOR WHICH THE BEST TECHNIQUE IS SIGNIFICANTLY BETTER, AND "$\bar{d}$" GIVES ITS AVERAGE EFFECT SIZE.

Evaluation metric	Best FL technique					Better than	$\bar{d}$
	Family	Formula	Total def.	Kill def.	Agg. def.		
best-case debugging scenario (localize any defective statement)							
Mean *EXAM* score	SBFL	dstar2	tests	–	–	149/155	−0.47
Mean Rank	MBFL	muse	elements	exact	avg	115/155	−0.16
worst-case debugging scenario (localize all defective statements)							
Mean *EXAM* score	SBFL	barinel	elements	–	–	149/155	−0.41
Mean Rank	MBFL	muse	elements	exact	avg	128/155	−0.19
average-case debugging scenario (localize 50% of the defective statements)							
Mean *EXAM* score	SBFL	barinel	elements	–	–	149/155	−0.48
Mean Rank	MBFL	muse	elements	exact	max	138/155	−0.24

TABLE VIII
ANOVA ANALYSIS OF THE EFFECT OF ALL FACTORS ON THE *EXAM* SCORE FOR REAL FAULTS. R^2 GIVES THE COEFFICIENT OF DETERMINATION.

Factor	Deg. of freedom	Sum of squares	F-value	p
sbfl *($R^2 = 0.66$)*				
Defect	296	326	60	<0.05
Debugging scenario	2	48.9	1336	<0.05
Formula	5	0.28	3	<0.05
Total definition	1	0.00314	0	*(insig.)*
mbfl *($R^2 = 0.67$)*				
Defect	296	4455	746	<0.05
Debugging scenario	2	571	14155	<0.05
Kill definition	5	284	2816	<0.05
Formula	5	16.7	166	<0.05
Aggregate definition	1	1.26	63	<0.05
Total definition	1	0.172	9	<0.05
sbfl + mbfl *($R^2 = 0.65$)*				
Defect	296	4570	718	<0.05
Debugging scenario	2	620	14411	<0.05
Family	12	362	1401	<0.05
Formula	5	16.6	154	<0.05
Total definition	1	0.172	8	<0.05

The choice of the formula accounts for no more than 1% of the non-defect variation in the *EXAM* scores. Furthermore, a post-hoc Tukey test (Table IX) showed that the differences between all formulas are insignificant for SBFL techniques.

All studied parameters have a statistically significant effect on the *EXAM* score, but the only FL technique parameters with a practically significant effect are family (SBFL vs. MBFL) and *kill definition* (for MBFL only).

VII. NEW TECHNIQUES

Beyond the quantitative results discussed so far, our studies exposed three limitations of MBFL techniques. MBFL techniques perform poorly on defects that involve unmutatable statements. MBFL techniques perform poorly when some mutants are covered but not killed. The run time of MBFL techniques is several orders of magnitude larger than for SBFL techniques, because mutation analysis requires running the

TABLE IX
PAIR-WISE COMPARISON (TUKEY POST-HOC TEST) OF ALL *Formulas*. FOR EACH PAIR, THE P VALUE INDICATES WHETHER THE CHOICE OF THE FORMULA SIGNIFICANTLY AFFECTS THE *EXAM* SCORE FOR REAL FAULTS.

Formula 1	Formula 2	p (*sbfl*)	p (*mbfl*)	p (*sbfl*+*mbfl*)
dstar2	barinel	*(insig.)*	**<0.01**	**<0.01**
muse	barinel	*(insig.)*	**<0.01**	**<0.01**
ochiai	barinel	*(insig.)*	*(insig.)*	*(insig.)*
opt2	barinel	*(insig.)*	**<0.01**	**<0.01**
tarantula	barinel	*(insig.)*	*(insig.)*	*(insig.)*
muse	dstar2	*(insig.)*	**<0.01**	**<0.01**
ochiai	dstar2	*(insig.)*	**<0.01**	**<0.01**
opt2	dstar2	*(insig.)*	**<0.01**	**<0.01**
tarantula	dstar2	*(insig.)*	*(insig.)*	*(insig.)*
ochiai	muse	*(insig.)*	**<0.01**	**<0.01**
opt2	muse	*(insig.)*	*(insig.)*	*(insig.)*
tarantula	muse	*(insig.)*	**<0.01**	**<0.01**
opt2	ochiai	*(insig.)*	**<0.01**	**<0.01**
tarantula	ochiai	*(insig.)*	*(insig.)*	*(insig.)*
tarantula	opt2	*(insig.)*	**<0.01**	**<0.01**

entire test suite many times (once per mutant), which can be prohibitive for larger-scale projects.

This section describes three approaches we explored with the aim to address these limitations and to design better fault localization techniques:

1) **MCBFL:** To address the limitation of covered yet not killed mutants, we explore an improved MBFL technique that uses mutation coverage information in addition to mutation kill information.
2) **MCBFL-hybrid:** To address the limitation of unmutatable statements, we explore a family of hybrid techniques that combine MCBFL with SBFL to further improve MCBFL.
3) **MRSBFL:** To address the scalability issue of MBFL techniques, we explore a faster MBFL technique that uses only mutation coverage information.

A. MCBFL

One contributor to the poor performance of MBFL techniques on real faults is that they consider only which tests *kill* which mutants, and disregard coverage information. For example, on Defects4J's bug `Chart-1`, one mutant changes the faulty statement from `if (this.dataset != null)` ... to `if (true)` ... — this mutant is covered (executed) by the sole failing test, but not killed by it. Intuitively, a mutant covered by a failing test should be more suspicious than a mutant that is not even executed.

To study this intuition, we introduced a new family of techniques, which operate in exactly the same way as MBFL techniques, except that the suspiciousness of each mutant is increased if it is covered by failing tests. We identified three ways to accomplish this:

1) **numerator** (small increase): if a mutant is covered by any failing test, increase its suspiciousness slightly by incrementing (or decrementing) the numerator of the fraction in the suspiciousness formula. This typically makes such mutants very slightly more suspicious, thus differentiating between mutants that are completely irrelevant to failing tests and mutants that are covered but not killed by failing tests.
2) **constant** (large increase): if a mutant is covered by any failing test, increase its suspiciousness by 1. For the formulas we consider, 1 is a fairly substantial increase in suspiciousness: this will typically ensure that every mutant covered by a failing test is more suspicious than any mutant not covered by a failing test.
3) **mirror** (variable increase): recall that in MBFL, each mutant's suspiciousness is computed by applying one of the formulas in section III-A1 to the numbers of passing/failing tests that kill it. The mirror technique applies the same formula also to the numbers of passing/failing tests that *cover* the mutant, and calculates the overall suspiciousness as the sum of the formula applied to kill and coverage information.

We applied each of these three modifications to the MBFL technique with the best mean FLT rank, to generate three variants of a new technique that we call "mutant-and-coverage-based fault localization" (MCBFL) technique.

1) Results: We evaluated the three MCBFL techniques on Defects4J's real defects, using the same methodology as in section VI. The **mirror** technique always outperformed the other two MCBFL techniques.

Tables X and XI show the relative performances of the best technique in each family (SBFL, MBFL, MCBFL, and the other techniques introduced in section VII). The best MCBFL technique significantly outperforms the MBFL techniques. Moreover, the best MCBFL technique does better than SBFL judging by mean FLT rank, but slightly worse in terms of mean *EXAM* score. This suggests that MCBFL techniques are better than spectrum-based ones at localizing most defects, but that they also have more outlier scores, causing them to do worse on average. If there was a way to improve the MCBFL techniques on those outliers, the resultant technique would likely outperform all other techniques in our design space.

B. MCBFL-hybrid

This section describes three hybrid approaches that aim to mitigate outlier scores, which drag down the effectiveness of mutation-based fault localization techniques.

One cause of outliers for MCBFL techniques is unmutatable statements. Since MCBFL, like MBFL, assigns suspiciousness values to statements by aggregating the suspiciousness values of their mutants, it assigns no suspiciousness score to unmutatable statements, so those statements appear at the end of the technique's statement-ranking as described in section II-B. Therefore, when only unmutatable lines are faulty, MCBFL techniques perform exceptionally badly. Note that unmutatable lines appear at the end, not because the MCBFL technique ranks them as very unsuspicious, but because the technique simply *has no opinion* about them. By using a different technique to assign suspiciousness values to those statements, we aim to make better guesses at how they should be ranked.

1) Failover: A *failover* technique assigns a suspiciousness to every statement, using:

1) If possible (i.e., if the statement is mutatable): the technique with the best mean FLT rank (i.e., MCBFL);
2) otherwise: the technique with the best mean *EXAM* score (i.e., DStar).

2) Suspiciousness-Averaging and Maxing: Another approach to improving MCBFL's scores for outliers is to, for each statement, combine the suspiciousness assigned by MCBFL with that assigned by SBFL. Assuming that both component techniques *usually* assign high suspiciousness to faulty statements, and outliers occur when they accidentally assign very low suspiciousness to those statements, combining the two suspiciousness values should mitigate the damage done by those mistakes. (Missing suspiciousness scores, such as for MCBFL on unmutatable statements, are taken to be 0.)

We experimented with two methods of combining the suspiciousness scores: averaging and taking the maximum.

3) Results: As evidenced by tables X and XI, the MCBFL-hybrid-failover technique does somewhat better than the best MCBFL technique, as measured by mean *EXAM* score. It does somewhat worse as measured by mean FLT rank, which is not surprising: it is based on the MCBFL technique with the best mean FLT rank, and adding SBFL was intended to improve the *mean EXAM* score, without regard for the FLT rank. The mean rank most likely gets worse because, even though the failover SBFL technique raises faulty unmutatable statements in the ranking produced by the MCBFL technique, it raises non-faulty unmutatable statements as well: the enormous improvement for a few bugs comes at the cost of very slightly worse scores on many more.

The other two hybrid techniques are displayed as *MCBFL-hybrid-avg* and *MCBFL-hybrid-max* in tables X and XI. MCBFL-hybrid-avg does better by every metric than the best SBFL technique.

C. MRSBFL

Mutation analysis is computationally expensive (see also discussion in section X-C). We therefore created less expensive mutant-based techniques by substituting mutant *coverage* information for the mutant *kill* information used by MBFL techniques: these "mutant-resolution spectrum-based" techniques (MRSBFL techniques) require only one run of the test suite, and therefore are comparable in execution time to the SBFL techniques.

1) Results: We evaluated MRSBFL versions of the best MBFL technique the other three experimental techniques. As table X shows, these techniques achieve nearly the same scores as their MCBFL-based equivalents, and at substantially lower computational cost.

D. The best new technique

Overall, MCBFL-hybrid-avg does better than any other technique in all debugging scenarios, but the difference in *EXAM* score and FLT rank is not practically significant. This can be seen in fig. 3, which compares the distributions of *EXAM* and absolute scores of the new MCBFL and MRSBFL techniques with the best, existing MBFL and SBFL techniques.

TABLE X
BEST FL TECHNIQUES PER FAMILY ACCORDING TO MEAN *EXAM* SCORE.

	Family	Formula	Total def.	Kill def.	Agg. def.	*EXAM* score
best-case debugging scenario (localize any defective statement)						
1	MCBFL-hybrid-avg	–	–	–	–	0.038 716
2	MCBFL-hybrid-max	–	–	–	–	0.039 247
3	MRSBFL-hybrid-max	–	–	–	–	0.040 315
4	MRSBFL-hybrid-avg	–	–	–	–	0.040 994
5	SBFL	dstar2	tests	–	–	0.041 321
12	MCBFL-hybrid-failover	–	–	–	–	0.044 556
13	MRSBFL-hybrid-failover	–	–	–	–	0.046 942
19	MCBFL	–	–	–	–	0.060 678
20	MRSBFL	–	–	–	–	0.063 442
21	MBFL	ochiai	elements	exact	max	0.077 431
worst-case debugging scenario (localize all defective statements)						
1	MRSBFL-hybrid-max	–	–	–	–	0.198 761
2	MCBFL-hybrid-avg	–	–	–	–	0.198 916
3	MCBFL-hybrid-max	–	–	–	–	0.198 985
4	MRSBFL-hybrid-avg	–	–	–	–	0.199 647
5	MRSBFL-hybrid-failover	–	–	–	–	0.200 300
6	MCBFL-hybrid-failover	–	–	–	–	0.200 379
7	SBFL	barinel	elements	–	–	0.201 780
19	MCBFL	–	–	–	–	0.221 530
20	MRSBFL	–	–	–	–	0.221 851
21	MBFL	ochiai	elements	exact	max	0.248 259
average-case debugging scenario (localize 50% of the defective statements)						
1	MCBFL-hybrid-avg	–	–	–	–	0.094 533
2	MCBFL-hybrid-max	–	–	–	–	0.095 183
3	MCBFL-hybrid-failover	–	–	–	–	0.095 428
4	SBFL	barinel	elements	–	–	0.095 465
8	MRSBFL-hybrid-max	–	–	–	–	0.095 822
10	MRSBFL-hybrid-avg	–	–	–	–	0.096 629
11	MRSBFL-hybrid-failover	–	–	–	–	0.097 010
19	MCBFL	–	–	–	–	0.112 649
20	MRSBFL	–	–	–	–	0.114 967
21	MBFL	barinel	elements	exact	max	0.134 167

TABLE XI
BEST FL TECHNIQUES PER FAMILY ACCORDING TO MEAN FLT RANK.

	Family	Formula	Total def.	Kill def.	Agg. def.	Rank
best-case debugging scenario (localize any defective statement)						
1	MCBFL	–	–	–	–	65.3
2	MCBFL-hybrid-avg	–	–	–	–	67.0
3	MBFL	muse	elements	exact	avg	72.8
6	MRSBFL	–	–	–	–	73.8
8	MCBFL-hybrid-failover	–	–	–	–	74.0
11	MRSBFL-hybrid-avg	–	–	–	–	74.6
54	MCBFL-hybrid-max	–	–	–	–	79.6
55	SBFL	dstar2	tests	–	–	79.7
87	MRSBFL-hybrid-max	–	–	–	–	81.1
97	MRSBFL-hybrid-failover	–	–	–	–	82.1
worst-case debugging scenario (localize all defective statements)						
1	MCBFL	–	–	–	–	70.1
2	MBFL	muse	elements	exact	avg	71.1
9	MRSBFL	–	–	–	–	73.4
61	MCBFL-hybrid-avg	–	–	–	–	81.6
115	MRSBFL-hybrid-avg	–	–	–	–	85.1
121	SBFL	dstar2	tests	–	–	85.9
123	MCBFL-hybrid-failover	–	–	–	–	86.1
151	MCBFL-hybrid-max	–	–	–	–	88.7
152	MRSBFL-hybrid-failover	–	–	–	–	88.9
154	MRSBFL-hybrid-max	–	–	–	–	89.2
average-case debugging scenario (localize 50% of the defective statements)						
1	MCBFL	–	–	–	–	63.2
2	MCBFL-hybrid-avg	–	–	–	–	67.9
3	MBFL	muse	elements	exact	max	68.3
5	MRSBFL	–	–	–	–	69.8
14	MCBFL-hybrid-failover	–	–	–	–	72.8
18	MRSBFL-hybrid-avg	–	–	–	–	74.2
30	SBFL	ochiai	elements	–	–	76.1
39	MCBFL-hybrid-max	–	–	–	–	76.6
69	MRSBFL-hybrid-failover	–	–	–	–	78.8
71	MRSBFL-hybrid-max	–	–	–	–	79.1

TABLE XII
RATIO OF DEFECTS WHOSE DEFECTIVE STATEMENTS APPEAR WITHIN THE TOP-5 AND TOP-10 OF THE TECHNIQUES' SUSPICIOUSNESS RANKING.

Technique	Best-case		Worst-case		Average-case	
	Top-5	Top-10	Top-5	Top-10	Top-5	Top-10
MCBFL-hybrid-avg	38%	46.8%	20.9%	27.3%	24.6%	32.3%
Metallaxis	31.3%	42.4%	17.8%	24.2%	19.9%	29%
DStar	30.3%	41.4%	16.5%	24.6%	17.8%	26.9%

Table XII compares our new MCBFL-hybrid-avg technique to the best SBFL and MBFL techniques in terms of how often they report defective statements near the top (top 5 or top 10) statements. This is relevant because a recent study [26] showed that 98% of practitioners consider a fault localization technique to be useful only if it reports the defective statement(s) within the top-10 of the suspiciousness ranking.

While the SBFL and MBFL techniques perform equally well under this light, they complement each other. This leads our new MCBFL-hybrid-avg technique to clearly report more defective statements near the top of the suspiciousness ranking than any previous technique.

VIII. THREATS TO VALIDITY

Generalization. Defects4J's data set spans five projects, written by different developers and targeting different applications. This suggests that our results may generalize to other projects and other test suites.

Based on the consistency of our results so far (table III), we believe that artificial faults are not good proxies for real faults, for evaluating any SBFL or MBFL techniques. However, slice-based or model-based techniques (see section IX) are sufficiently different that our results may not carry over to them.

Applicability. *EXAM* score may not be the best metric for comparing FL techniques: in one study, expert programmers diagnosed faults more quickly with FL tools than without, but better *EXAM* scores did not always result in significantly faster debugging [37]. This problem is somewhat mitigated here because our study revolves around the *comparison* of FL techniques rather than their absolute performances. Furthermore, our "mean FL technique rank" metric is agnostic to whether absolute or relative scores are being compared. Other metrics may be better correlated with programmer performance, such as defective statements in the top-10 (section VII-D). It is unknown which metrics are best for other uses of fault localization, such as automated program repair. Even for the use case of human debugging, our study yields insights into the construction and evaluation of FL techniques.

Verifiability. All of our results can be reproduced by an interested reader. Our data and scripts are publicly available at https://bitbucket.org/rjust/fault-localization-data. Our methodology builds upon other tools, which are also publicly available. Notable examples are the Defects4J database of real faults (https://github.com/rjust/defects4j), the GZoltar fault localization tool (http://www.gzoltar.com/), and the Major mutation framework (http://mutation-testing.org/).

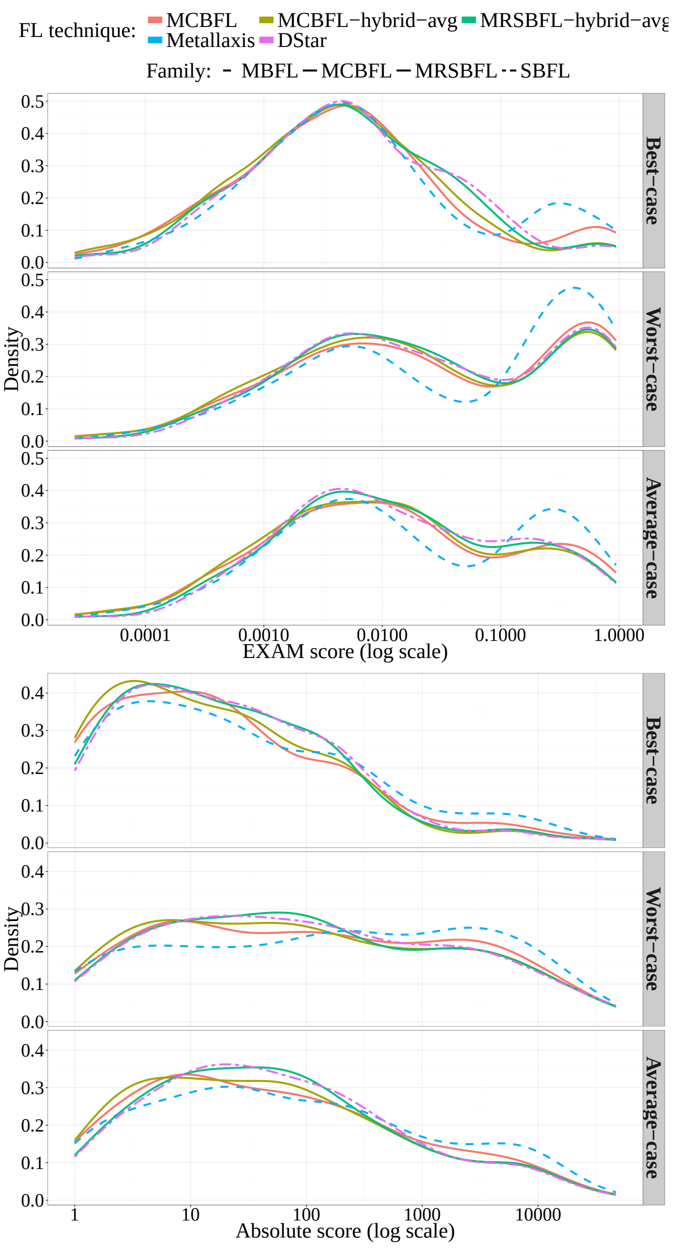

Fig. 3. Distributions of *EXAM* and absolute scores for the new MCBFL and MRSBFL FL techniques. The absolute score is the location of a defective statement in the suspiciousness ranking of program statements, computed by a fault localization technique. The best existing SBFL and MBFL techniques are included as a baseline.

IX. RELATED WORK

According to a recent survey [47], the most studied and evaluated fault localization techniques are spectrum-based [4], [19], [27], [46], slice-based [44], [51], model-based [2], [49], and mutation-based [33], [36].

Slice-based techniques [51] 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. 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], [31] 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. 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] achieved a better diagnosis when compared to SBFL techniques alone. However, model-based techniques are computationally intractable [32] and do not scale to large software programs.

A. Evaluation of fault localization techniques

a) Evaluation Studies: Comparisons of SBFL techniques have been conducted by Jones and Harrold [19], Abreu et al. [4], Le et al. [29], Wong et al. [46], and Santelices et al. [41]. Jones and Harrold compared five spectrum- and slice-based techniques, using the Siemens set of tiny programs to evaluate each. They found their Tarantula formula to be the best, doing significantly better than Nearest-Neighbor [40] (which looks for statements covered by a failing test but not by the most-similar passing test), Set-Union and Set-Intersection (which look for statements executed only by failing tests, or only by passing tests), and Cause Transitions (which looks for statements that change memory in a way that causes failures when transplanted into passing tests during execution). The techniques we studied all resemble Tarantula much more than these other formulas, leading to a more narrowly scoped but more thorough investigation that focuses on the most promising approaches. We found that on artificial faults, Tarantula does significantly worse than any other SBFL technique, and worse than Metallaxis. However, on real faults there is no statistically significant difference between Tarantula and the best SBFL technique.

Abreu et al. [4] considered several different spectrum-based techniques, all identical to Tarantula except in their formulas. They found that Ochiai outperformed all others on the Siemens programs, regardless of the number of passing/failing tests. We compared Tarantula and Ochiai to more formulas and extended them to mutant-based FL as well. Ochiai has also been evaluated in the context of an embedded TV software stack containing 450KLOC [53]. We confirmed that Ochiai does relatively well on artificial faults, although outperformed by Op2 and DStar.

Le et al. [29] also compared several formulas over the Siemens set, as well on NanoXML, XML-security, and Space; they too find that Ochiai is the best, though not always statistically significantly. It even outperforms three theoretically optimal formulas derived by Xie et al. [50], because the optimality assumptions (e.g., 100% code coverage) are unmet in the studied programs. We do not consider the same set of FL techniques, but we do find Ochiai to be among the best techniques on real faults.

Wong et al. [46] compared over twenty formulas' performances on the Siemens set, grep, make, gzip, and several other real-world programs. They introduce the D^* formula, which typically outperforms all other formulas on all projects they study. They most thoroughly explore D^2, so we selected it for use here, finding that it outperformed all other techniques in our design space, usually statistically significantly.

Santelices et al. [41] also explored variations on spectrum-based FL, but instead of comparing formulas they compared types of coverage data: statements, branches, and data dependencies. There is no clear winner, but a hybrid of all three modestly outperformed any one individually, evaluated on the Siemens set, NanoXML, and XML-security.

There have been fewer comparisons between spectrum- and mutant-based FL techniques. Papadakis and Le Traon [36], when introducing their mutant-based FL technique Metallaxis, compared it to the spectrum-based equivalent (the "Ochiai" technique in this paper), using the Siemens set. They found that their Metallaxis was consistently superior across several different test suites. We confirmed this finding on artificial faults, but refuted it on real faults.

Moon et al. [33] also performed a cross-family comparison, between their mutation-based technique MUSE and three spectrum-based techniques (Jaccard, Ochiai, and Op2) on flex, grep, gzip, sed, and space. The authors concluded that their MUSE technique typically performs better than the other three. We disagree with this finding, due to the fact that mutants rarely fix failing tests on either real faults or many artificial faults.

B. Using developers to evaluate fault localization techniques

Metrics such as T-score, Expense, or *EXAM* score (see Section II-A 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.[2] To understand the perfect fault understanding assumption, Parnin and Orso conducted a preliminary user study on debugging effectiveness, involving 34 graduate students [37]. 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 [37]) could help developers on debugging. One group used the authors' tool GZoltar [9] and the other group did not. The group that used GZoltar's graphical visualizations

[2]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.

of the ranking found the faulty component more often than the control group, and spent less time on debugging.

A recent study [26] surveyed 386 practitoners to access their expectations of automated fault localization techniques. Statement granularity is in the top-3 granularity preferences of practitioners. All fault localization techniques used in this paper (published ones, sections III-A1 and III-A2, and new ones, section VII) are statement-based, i.e., in line with practitioners' opinion. 98% of practitioners say they are not willing to inspect more than 10 statements before finding the true faulty one. For 48% of the total faults, MCBFL-hybrid-avg ranked the faulty line in the top-10. That is, for almost half of the faults used in our study, the 10 first positions of a ranking produced by MCBFL-hybrid-avg included the faulty line.

C. Mutants vs. Real Faults

It has been very common to evaluate and compare fault localization techniques using manually-seeded artificial faults (e.g., Siemens set [4], [19], [36], [41]) or mutations (e.g., [12], [41], [46]) 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 [43], 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 (22–96 KLOC), and we independently evaluated the performance of each FL technique on a larger number of real faults and artificial faults (see Section V-A).

The use of mutants as a replacement for real faults has been investigated in other domains. 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. [22] studied the real faults of the Defects4J [21] 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 [21] are not coupled with commonly used mutation operators [18], suggesting a need for stronger mutation operators.

However, Namin et al. [35] studied the same set of programs as previous studies [7], and cautioned 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.

X. Lessons Learned

A. Complexity of real faults

Real faults vary in their characteristics far more than artificial faults do, which requires considerable revisions to existing methodology for evaluating FL techniques. For example, with multi-line faults it is nontrivial to identify which statements in the buggy code should be considered defective: declarations and scoping braces are not ranked by most FL techniques, but localizing a closely related statement may be good enough.

B. Debugging scenarios

There may be no single, proper cost model for fault localization effort. A debugging scenario is very likely to be different for humans and automated techniques. For example, a developer might be able to repair a defect after encountering the first defective statement or after encountering 50% of the defective statements on average. An automated repair technique, however, might need to go through all defective statements before being able to generate a patch. Our evaluation acknowledges this challenge and studies the effectiveness of fault localization techniques for multiple debugging scenarios.

C. Efficiency of fault localization techniques

Our analysis does not consider the time it takes to execute each FL technique. MBFL is computationally very costly: our mutation analysis completed in under 168 hours for only 297/357 faults, and in under 32 hours for only 229/357 faults. Note that our implementation avoids unnecessary mutant executions by exploiting several sound optimizations such as mutation coverage information [23] and non-redundant mutation operators [24]. Moreover, for Metallaxis, our implementation is able to determine that the suspiciousness of a mutant is 0 without running all tests on that mutant if no fault-triggering test in the test suite kills that mutant. This optimization reduces Metallaxis's runtime by a factor of 10. (This can not be done for MUSE, since a mutant's suspiciousness can be affected by passing tests regardless of whether failing tests kill it.)

The best techniques we found mostly require a mutation analysis of the target program, and for most users, the slight improvement in output quality (compared to SBFL) will not be worth this computational expense. The MRSBFL-hybrid-avg technique we developed is consistently on par with (or slightly superior to) SBFL techniques, but that is the full extent to which we find mutants are useful without incurring these computational costs.

Recall that one of MBFL's limitations is its inability to rank unmutatable statements. In theory, this could be addressed by using a large enough set of mutation operators, including higher-order mutation operators, such that all statements are mutatable. This is, however, a very costly solution and might render MBFL as being prohibitively expensive, even for mid-sized programs. Given that an optimized implementation of MBFL did not finish within 168 CPU hours for 17% of real faults, adding many more mutants will only amplify this scalability issue. Furthermore, the benefit of mutating more statements would still be limited to the 10% of real faults that involve statements not mutatable by Major's mutation operators.

XI. Conclusion

Fault localization techniques' performance has mostly been evaluated using artificial faults (e.g., mutants). Artificial faults differ from real faults, so previous studies do not establish which techniques are best at finding real faults.

This paper evaluates the performance of 7 previously-studied fault localization techniques. We replicated previous studies in a systematic way on a larger number of *artificial* faults and on larger subject programs; this confirmed 70% of previous results and falsified 30%. We also evaluated the FL techniques on hundreds of *real* faults, and we found that artificial faults are not useful for predicting which fault localization techniques perform best on real faults. Of the previously-reported results on artificial faults, 60% were statistically insignificant on real faults and the other 40% were falsified; most notably, MBFL techniques are relatively less useful for real faults.

We analyzed the similarities and differences among the FL techniques to synthesize a design space that encompasses them. We evaluated 156 techniques to determine what aspects of a FL technique are most important.

We created new hybrid techniques that outperform previous techniques on the important metric of reporting defects in the top-10 slots of the ranking. The hybrids combine existing techniques in a way that preserves the complementary strengths of each while mitigating their weaknesses.

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. Technical Report 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, 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.

[22] 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.

[23] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–20, 2012.

[24] R. Just and F. Schweiggert. Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability*, 25(5-7):490–507, 2015.

[25] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 612–615, November 9–11 2011.

[26] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International*

Symposium on Software Testing and Analysis, ISSTA 2016, pages 165–176, New York, NY, USA, 2016. ACM.

[27] G. Laghari, A. Murgia, and S. Demeyer. Improving spectrum based fault localisation techniques. In *In Proceedings of the 14th Belgian-Netherlands Software Evolution Seminar (BENEVOL'2015)*, December 2015.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

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

[39] 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.

[40] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, Oct. 2003.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31:1–31:40, Oct. 2013.

[51] 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.

[52] 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.

[53] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. C. van Gemund. Diagnosis of embedded software using program spectra. In *ECBS*, pages 213–220, Mar. 2007.

Appendix

Figure 4 shows how the *Kill definition* affects MUSE's scores for artificial and real faults. Table XIII provides the results of the replication studies for all debugging scenarios, and Figures 5 and 6 show the distributions of *EXAM* and absolute scores per project. Tables XIV and XV show the top-25 fault localization techniques for real faults, considering all evaluation metrics and all debugging scenarios.

TABLE XIII

EXPANDED VERSION OF TABLE IV, WITH DATA FOR ALL DEBUGGING SCENARIOS. THE TABLE CONTAINS PREVIOUSLY-REPORTED COMPARISONS (AS LISTED IN TABLE I), AND OUR RESULTS FOR THOSE COMPARISONS. THE COLUMN "95% CI" GIVES THE CONFIDENCE INTERVAL FOR THE DIFFERENCE IN MEANS AND *d* GIVES THE EFFECT SIZE (COHEN'S D). THE COLUMN "(B – EQ – W)" GIVES THE COUNTS FOR: PER DEFECT, WAS THE WINNER BETTER, EQUAL TO, OR WORSE COMPARED TO THE LOSER, IGNORING THE MAGNITUDE OF THE DIFFERENCE.

Previous comparisons	**Our study on artificial faults**				**Our study on real faults**			
Winner > loser	agree?	d (eff. size)	95% CI	(b – eq – w)	agree?	d (eff. size)	95% CI	(b – eq – w)
best-case debugging scenario (localize any defective statement)								
Ochiai > Tarantula [28], [29], [34], [46], [52]	**yes**	(-0.25)	[-0.007, -0.005]	(997–1275–1)	*(insig.)*	*(-0.11)*	[-0.007, 0.001]	(37–184–8)
Barinel > Ochiai [5]	**no**	(0.26)	[0.005, 0.006]	(1–1277–995)	*(insig.)*	*(0.06)*	[-0.001, 0.003]	(8–186–35)
Barinel > Tarantula [5]	**yes**	*(-0.06)*	[-0.001, -0.000]	(8–2265–0)	*(insig.)*	*(-0.09)*	[-0.006, 0.001]	(2–227–0)
Op2 > Ochiai [34]	**yes**	*(-0.13)*	[-0.001, -0.001]	(314–1955–4)	no	*(0.14)*	[0.001, 0.016]	(27–178–24)
Op2 > Tarantula [33], [34]	**yes**	(-0.26)	[-0.009, -0.006]	(1034–1235–4)	*(insig.)*	*(0.08)*	[-0.004, 0.014]	(38–166–25)
DStar > Ochiai [28], [46]	**yes**	*(-0.14)*	[-0.001, -0.000]	(205–2068–0)	*(insig.)*	*(-0.02)*	[-0.002, 0.001]	(16–205–8)
DStar > Tarantula [20], [28], [46]	**yes**	(-0.25)	[-0.008, -0.006]	(1005–1267–1)	*(insig.)*	*(-0.11)*	[-0.007, 0.001]	(37–182–10)
Metallaxis > Ochiai [36]	**yes**	*(-0.16)*	[-0.032, -0.019]	(1509–192–572)	**no**	(0.2)	[0.012, 0.053]	(123–16–90)
MUSE > Op2 [33]	**no**	*(0.08)*	[0.006, 0.021]	(1502–106–665)	**no**	**0.81**	[0.143, 0.197]	(77–1–151)
MUSE > Tarantula [33]	*(insig.)*	*(0.03)*	[-0.001, 0.014]	(1659–74–540)	**no**	**0.87**	[0.149, 0.201]	(74–1–154)
worst-case debugging scenario (localize all defective statements)								
Ochiai > Tarantula [28], [29], [34], [46], [52]	**yes**	(-0.25)	[-0.007, -0.005]	(997–1275–1)	*(insig.)*	*(-0.03)*	[-0.005, 0.003]	(16–202–11)
Barinel > Ochiai [5]	**no**	(0.26)	[0.005, 0.006]	(1–1277–995)	*(insig.)*	*(-0.06)*	[-0.004, 0.002]	(11–204–14)
Barinel > Tarantula [5]	**yes**	*(-0.06)*	[-0.001, -0.000]	(8–2265–0)	*(insig.)*	*(-0.09)*	[-0.006, 0.001]	(2–227–0)
Op2 > Ochiai [34]	**yes**	*(-0.13)*	[-0.001, -0.001]	(314–1955–4)	**no**	(0.21)	[0.006, 0.026]	(27–164–38)
Op2 > Tarantula [33], [34]	**yes**	(-0.26)	[-0.009, -0.006]	(1034–1235–4)	**no**	*(0.18)*	[0.004, 0.026]	(30–158–41)
DStar > Ochiai [28], [46]	**yes**	*(-0.14)*	[-0.001, -0.000]	(205–2068–0)	*(insig.)*	*(-0.02)*	[-0.000, 0.000]	(8–214–7)
DStar > Tarantula [20], [28], [46]	**yes**	(-0.25)	[-0.008, -0.006]	(1005–1267–1)	*(insig.)*	*(-0.03)*	[-0.005, 0.003]	(17–200–12)
Metallaxis > Ochiai [36]	**yes**	*(-0.16)*	[-0.032, -0.019]	(1509–192–572)	**no**	(0.25)	[0.023, 0.074]	(140–13–76)
MUSE > Op2 [33]	**no**	*(0.08)*	[0.006, 0.021]	(1502–106–665)	**no**	0.62	[0.121, 0.186]	(88–2–139)
MUSE > Tarantula [33]	*(insig.)*	*(0.03)*	[-0.001, 0.014]	(1659–74–540)	**no**	0.71	[0.138, 0.199]	(87–1–141)
average-case debugging scenario (localize 50% of the defective statements)								
Ochiai > Tarantula [28], [29], [34], [46], [52]	**yes**	(-0.25)	[-0.007, -0.005]	(997–1275–1)	*(insig.)*	*(-0.06)*	[-0.006, 0.002]	(32–180–17)
Barinel > Ochiai [5]	**no**	(0.26)	[0.005, 0.006]	(1–1277–995)	*(insig.)*	*(-0.03)*	[-0.003, 0.002]	(17–182–30)
Barinel > Tarantula [5]	**yes**	*(-0.06)*	[-0.001, -0.000]	(8–2265–0)	*(insig.)*	*(-0.09)*	[-0.006, 0.001]	(2–227–0)
Op2 > Ochiai [34]	**yes**	*(-0.13)*	[-0.001, -0.001]	(314–1955–4)	**no**	(0.21)	[0.005, 0.022]	(33–155–41)
Op2 > Tarantula [33], [34]	**yes**	(-0.26)	[-0.009, -0.006]	(1034–1235–4)	no	*(0.16)*	[0.002, 0.022]	(42–143–44)
DStar > Ochiai [28], [46]	**yes**	*(-0.14)*	[-0.001, -0.000]	(205–2068–0)	*(insig.)*	*(0.07)*	[-0.000, 0.001]	(16–202–11)
DStar > Tarantula [20], [28], [46]	**yes**	(-0.25)	[-0.008, -0.006]	(1005–1267–1)	*(insig.)*	*(-0.05)*	[-0.006, 0.003]	(33–178–18)
Metallaxis > Ochiai [36]	**yes**	*(-0.16)*	[-0.032, -0.019]	(1509–192–572)	**no**	(0.23)	[0.017, 0.060]	(140–12–77)
MUSE > Op2 [33]	**no**	*(0.08)*	[0.006, 0.021]	(1502–106–665)	**no**	0.72	[0.141, 0.203]	(68–1–160)
MUSE > Tarantula [33]	*(insig.)*	*(0.03)*	[-0.001, 0.014]	(1659–74–540)	**no**	**0.82**	[0.154, 0.213]	(64–1–164)

Emphasis on whether our study *agrees* indicates p-value: **p<0.01**, p<0.05, *($p \geq 0.05$)*.
Emphasis on Cohen's *d* indicates effect size: **large**, medium, (small), *(negligible)*.

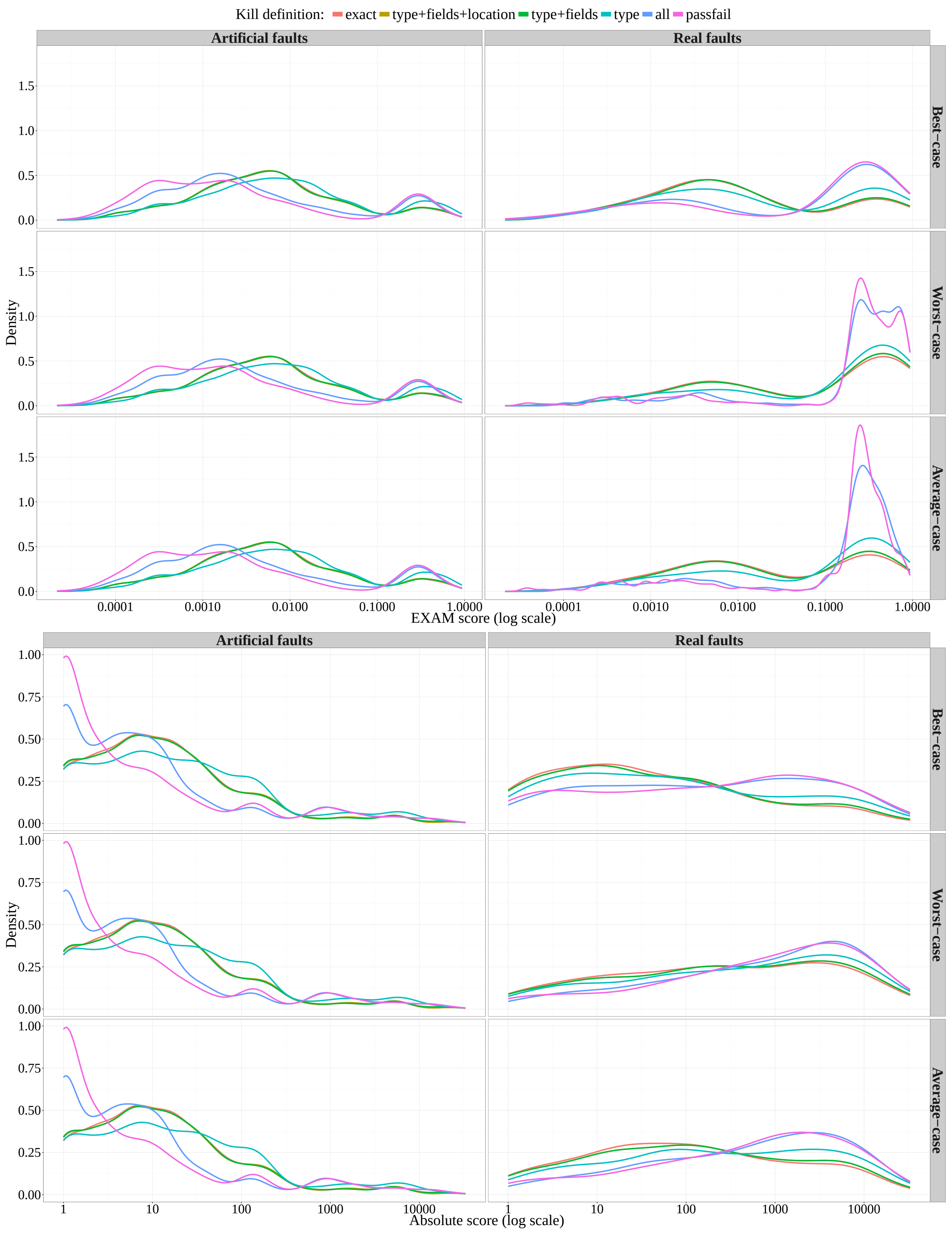

Fig. 4. The effect of the choice of the "Kill definition" on MUSE's *EXAM* and absolute scores for artificial vs. real faults.

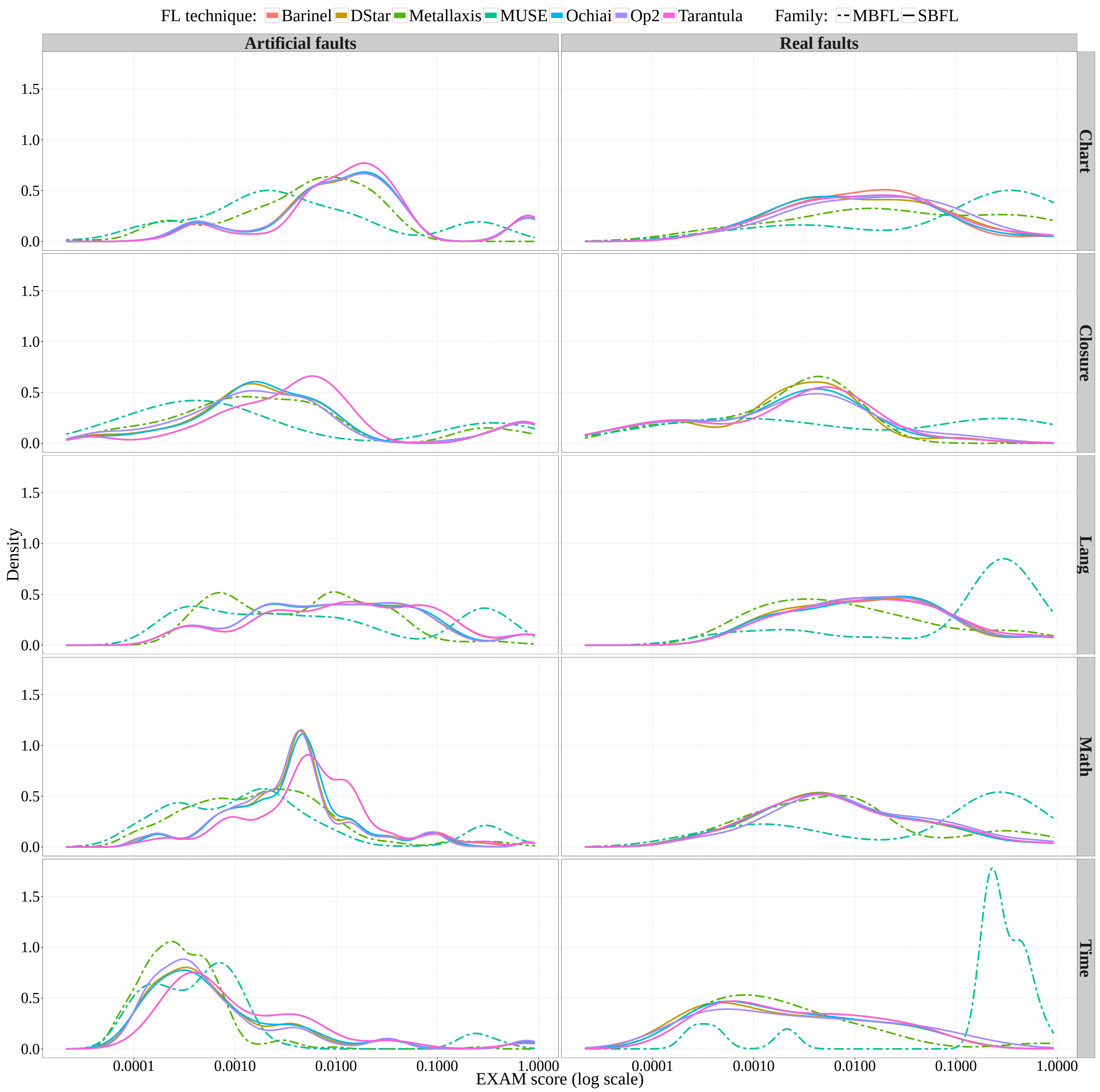

Fig. 5. Distribution of the *EXAM* scores per project for all fault localization techniques and artificial vs. real faults.

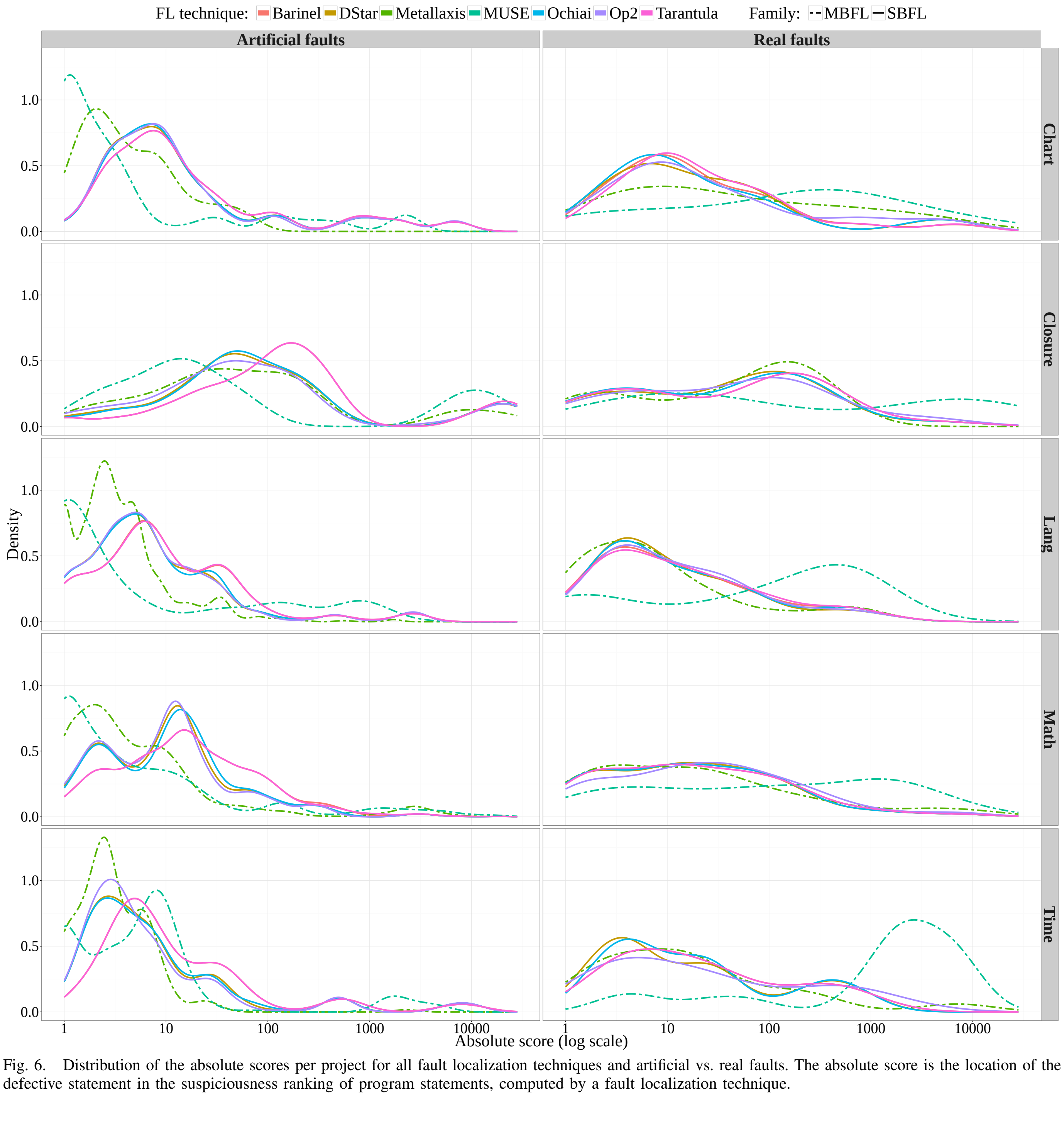

Fig. 6. Distribution of the absolute scores per project for all fault localization techniques and artificial vs. real faults. The absolute score is the location of the defective statement in the suspiciousness ranking of program statements, computed by a fault localization technique.

TABLE XIV
TOP-25 FL TECHNIQUES ACCORDING TO MEAN *EXAM* SCORE.

	Family	Formula	Total def.	Kill def.	Agg. def.	*EXAM* score
	best-case debugging scenario (localize any defective statement)					
1	MCBFL-hybrid-avg	–	–	–	–	0.038 716
2	MCBFL-hybrid-max	–	–	–	–	0.039 247
3	MRSBFL-hybrid-max	–	–	–	–	0.040 315
4	MRSBFL-hybrid-avg	–	–	–	–	0.040 994
5	SBFL	dstar2	tests	–	–	0.041 321
6	SBFL	ochiai	elements	–	–	0.041 491
7	SBFL	ochiai	tests	–	–	0.041 491
8	SBFL	barinel	elements	–	–	0.042 261
9	SBFL	barinel	tests	–	–	0.042 261
10	SBFL	tarantula	elements	–	–	0.044 072
11	SBFL	tarantula	tests	–	–	0.044 072
12	MCBFL-hybrid-failover	–	–	–	–	0.044 556
13	MRSBFL-hybrid-failover	–	–	–	–	0.046 942
14	SBFL	muse	tests	–	–	0.048 060
15	SBFL	opt2	elements	–	–	0.048 255
16	SBFL	opt2	tests	–	–	0.048 255
17	SBFL	dstar2	elements	–	–	0.048 271
18	SBFL	muse	elements	–	–	0.051 032
19	MCBFL	–	–	–	–	0.060 678
20	MRSBFL	–	–	–	–	0.063 442
21	MBFL	ochiai	elements	exact	max	0.077 431
22	MBFL	ochiai	tests	exact	max	0.077 431
23	MBFL	dstar2	tests	exact	max	0.077 564
24	MBFL	barinel	elements	exact	max	0.077 900
25	MBFL	barinel	tests	exact	max	0.077 900
	worst-case debugging scenario (localize all defective statements)					
1	MRSBFL-hybrid-max	–	–	–	–	0.198 761
2	MCBFL-hybrid-avg	–	–	–	–	0.198 916
3	MCBFL-hybrid-max	–	–	–	–	0.198 985
4	MRSBFL-hybrid-avg	–	–	–	–	0.199 647
5	MRSBFL-hybrid-failover	–	–	–	–	0.200 300
6	MCBFL-hybrid-failover	–	–	–	–	0.200 379
7	SBFL	barinel	elements	–	–	0.201 780
8	SBFL	barinel	tests	–	–	0.201 780
9	SBFL	dstar2	tests	–	–	0.202 712
10	SBFL	ochiai	elements	–	–	0.202 741
11	SBFL	ochiai	tests	–	–	0.202 741
12	SBFL	tarantula	elements	–	–	0.203 590
13	SBFL	tarantula	tests	–	–	0.203 590
14	SBFL	dstar2	elements	–	–	0.208 712
15	SBFL	muse	elements	–	–	0.215 287
16	SBFL	opt2	elements	–	–	0.217 729
17	SBFL	opt2	tests	–	–	0.217 729
18	SBFL	muse	tests	–	–	0.218 060
19	MCBFL	–	–	–	–	0.221 530
20	MRSBFL	–	–	–	–	0.221 851
21	MBFL	ochiai	elements	exact	max	0.248 259
22	MBFL	ochiai	tests	exact	max	0.248 259
23	MBFL	barinel	elements	exact	max	0.248 374
24	MBFL	barinel	tests	exact	max	0.248 374
25	MBFL	barinel	elements	exact	avg	0.249 260
	average-case debugging scenario (localize 50% of the defective statements)					
1	MCBFL-hybrid-avg	–	–	–	–	0.094 533
2	MCBFL-hybrid-max	–	–	–	–	0.095 183
3	MCBFL-hybrid-failover	–	–	–	–	0.095 428
4	SBFL	barinel	elements	–	–	0.095 465
5	SBFL	barinel	tests	–	–	0.095 465
6	SBFL	ochiai	elements	–	–	0.095 724
7	SBFL	ochiai	tests	–	–	0.095 724
8	MRSBFL-hybrid-max	–	–	–	–	0.095 822
9	SBFL	dstar2	tests	–	–	0.096 028
10	MRSBFL-hybrid-avg	–	–	–	–	0.096 629
11	MRSBFL-hybrid-failover	–	–	–	–	0.097 010
12	SBFL	tarantula	elements	–	–	0.097 276
13	SBFL	tarantula	tests	–	–	0.097 276
14	SBFL	dstar2	elements	–	–	0.103 783
15	SBFL	opt2	elements	–	–	0.108 650
16	SBFL	opt2	tests	–	–	0.108 650
17	SBFL	muse	elements	–	–	0.109 169
18	SBFL	muse	tests	–	–	0.110 579
19	MCBFL	–	–	–	–	0.112 649
20	MRSBFL	–	–	–	–	0.114 967
21	MBFL	barinel	elements	exact	max	0.134 167
22	MBFL	barinel	tests	exact	max	0.134 167
23	MBFL	ochiai	elements	exact	max	0.134 318
24	MBFL	ochiai	tests	exact	max	0.134 318
25	MBFL	dstar2	tests	exact	max	0.134 341

TABLE XV
TOP-25 FL TECHNIQUES ACCORDING TO MEAN FLT RANK.

	Family	Formula	Total def.	Kill def.	Agg. def.	FLT rank
	best-case debugging scenario (localize any defective statement)					
1	MCBFL	–	–	–	–	65.3
2	MCBFL-hybrid-avg	–	–	–	–	67.0
3	MBFL	muse	elements	exact	avg	72.8
4	MBFL	muse	elements	type+fields	avg	73.3
5	MBFL	muse	elements	type+fields+location	avg	73.3
6	MRSBFL	–	–	–	–	73.8
7	MBFL	dstar2	tests	exact	avg	73.9
8	MCBFL-hybrid-failover	–	–	–	–	74.0
9	MBFL	dstar2	tests	type	avg	74.4
10	MBFL	muse	tests	exact	avg	74.6
11	MRSBFL-hybrid-avg	–	–	–	–	74.6
12	MBFL	muse	tests	type+fields	avg	74.8
13	MBFL	muse	tests	type+fields+location	avg	74.8
14	MBFL	dstar2	tests	type+fields	avg	75.0
15	MBFL	dstar2	tests	type+fields+location	avg	75.0
16	MBFL	ochiai	elements	exact	avg	75.3
17	MBFL	ochiai	tests	exact	avg	75.3
18	MBFL	muse	elements	type	max	75.7
19	MBFL	ochiai	elements	type+fields	avg	75.8
20	MBFL	ochiai	elements	type+fields+location	avg	75.8
21	MBFL	ochiai	tests	type+fields	avg	76.0
22	MBFL	ochiai	tests	type+fields+location	avg	76.0
23	MBFL	barinel	elements	type	avg	76.4
24	MBFL	barinel	tests	type	avg	76.4
25	MBFL	muse	tests	type	max	76.8
	worst-case debugging scenario (localize all defective statements)					
1	MCBFL	–	–	–	–	70.1
2	MBFL	muse	elements	exact	avg	71.1
3	MBFL	muse	elements	exact	max	71.2
4	MBFL	muse	elements	type+fields	avg	71.9
5	MBFL	muse	elements	type+fields+location	avg	71.9
6	MBFL	muse	elements	type+fields	max	72.1
7	MBFL	muse	elements	type+fields+location	max	72.1
8	MBFL	muse	elements	type	max	72.5
9	MRSBFL	–	–	–	–	73.4
10	MBFL	ochiai	elements	passfail	max	75.7
11	MBFL	ochiai	tests	passfail	max	75.7
12	MBFL	dstar2	tests	passfail	max	75.7
13	MBFL	barinel	elements	passfail	max	75.9
14	MBFL	barinel	tests	passfail	max	75.9
15	MBFL	muse	tests	type	max	76.0
16	MBFL	muse	tests	exact	max	76.1
17	MBFL	tarantula	tests	passfail	max	76.3
18	MBFL	dstar2	tests	passfail	avg	76.3
19	MBFL	muse	tests	type+fields	avg	76.6
20	MBFL	muse	tests	type+fields+location	avg	76.6
21	MBFL	barinel	elements	passfail	avg	76.6
22	MBFL	barinel	tests	passfail	avg	76.6
23	MBFL	tarantula	elements	passfail	max	76.7
24	MBFL	muse	tests	type+fields	max	76.9
25	MBFL	muse	tests	type+fields+location	max	76.9
	average-case debugging scenario (localize 50% of the defective statements)					
1	MCBFL	–	–	–	–	63.2
2	MCBFL-hybrid-avg	–	–	–	–	67.9
3	MBFL	muse	elements	exact	max	68.3
4	MBFL	muse	elements	exact	avg	68.6
5	MRSBFL	–	–	–	–	69.8
6	MBFL	muse	elements	type+fields	max	70.0
7	MBFL	muse	elements	type+fields+location	max	70.0
8	MBFL	muse	elements	type+fields	avg	70.0
9	MBFL	muse	elements	type+fields+location	avg	70.0
10	MBFL	muse	tests	exact	avg	71.6
11	MBFL	muse	tests	exact	max	72.2
12	MBFL	muse	tests	type+fields	avg	72.4
13	MBFL	muse	tests	type+fields+location	avg	72.4
14	MCBFL-hybrid-failover	–	–	–	–	72.8
15	MBFL	dstar2	tests	exact	avg	73.4
16	MBFL	barinel	elements	exact	avg	73.5
17	MBFL	barinel	tests	exact	avg	73.5
18	MRSBFL-hybrid-avg	–	–	–	–	74.2
19	MBFL	dstar2	tests	type+fields	avg	74.2
20	MBFL	dstar2	tests	type+fields+location	avg	74.2
21	MBFL	barinel	elements	type+fields	avg	74.3
22	MBFL	barinel	tests	type+fields	avg	74.3
23	MBFL	barinel	elements	type+fields+location	avg	74.3
24	MBFL	barinel	tests	type+fields+location	avg	74.3
25	MBFL	muse	tests	type+fields	max	74.4