Evaluating and improving fault localization

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/HASLab, 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*—Most fault localization techniques take as input a faulty program, and produce 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 typically evaluate it on programs with known faults. The technique is scored based on where in its output list the defective code appears. This enables the 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 (from the spectrum-based and mutation-based families). We used 2995 artificial faults in 6 real-world programs. Our results support 7 of the previous claims as statistically significant, but only 3 as having non-negligible effect sizes. Then, we evaluated the same 10 claims, using 310 *real* faults from the 6 programs. Every previous result was refuted or was statistically and practically insignificant. 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, using an overall set of 395 *real* faults. Then, we extended this design space 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 (typically, 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 [15]. 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 [45]. 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 [41]) have been used in previous comparisons [45] of fault localization techniques, the vast majority of faults used in such comparisons are fake faults, mostly mutants. The artificial faults were mutants automatically created by a tool [26], [27], [49], or mutant-like manually-seeded faults created by students [44], [46] or researchers [16].

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. If so, 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 [37].

This paper compares fault localization techniques on real vs. artificial faults. Techniques that localize artificial faults best do *not* perform best on real faults. Our experiments are based on 7 previously-studied fault localization techniques from the spectrum-based and mutation-based families.

The contributions of this paper include:

- A replication study that repeats and extends previous experiments, comparing 7 fault localization techniques on 2995 artificial faults. We mitigated threats to internal validity by re-implementing all the techniques in a single infrastructure and using the same experimental scripts, faults, and other experimental variables. Our results confirm 70% of previously-reported comparisons (such as "Ochiai is better than Tarantula" [26], [27], [31], [44], [49]) and refute 30%.
- A new study that compares the 7 fault localization techniques on 310 *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 [33], 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 [19], [26], [44], but on real faults there is no 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 technique.

- An explication of the design space of fault localization techniques. Previous work 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 one of the least important factors. 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 statement) within the top-5 or top-10 statements 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.

A technical report [35] has more examples, data, & analyses.

II. Evaluating fault localization

Many studies have evaluated and compared FL techniques [2]–[5], [18], [19], [26], [27], [30], [31], [33], [38], [44], [49]. Table I summarizes these studies. The majority of studies revolve around the same set of programs and use largely artificial faults. 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 other levels of granularity.

Given a fault localization technique T and a program P of size N with a single known defective statement d, a numerical measure of the quality of the fault localization technique can be computed as follows [36], [40]: (1) run the FL technique to compute the sorted list of suspicious statements; (2) let n be the rank of d in the list; (3) use a metric proposed in the literature to evaluate the effectiveness of a FL technique, e.g., LIL [30], T-score [28], Expense [18], or *EXAM* score [43].[1] For concreteness this paper uses *EXAM* score, which is the most popular metric, but our results generalize to the others. The *EXAM* score is n/N, where N is the number of statements in the program. 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 change to one executable statement in the program, as is the case for mutants. 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-statement faults, faults of omission, and defective non-executable code such as declarations.

1) Ties in the suspiciousness score: We assume that the sorting function breaks ties arbitrarily. 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 [40], [45].

2) Multi-line 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.

3) Multi-statement faults: 76% of real-world bug fixes span multiple statements [21]. 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. All of our the experimental results are generally consistent for the three scenarios.

4) Faults of omission: In 30% of cases [21], a bug fix consists of adding new code rather than changing existing code. The defective program contains no defective statement, but some are missing. Previous studies have not reported whether and how this issue was addressed.

A FL technique communicates with the programmer in terms of the program's representation: statements of source code. A FL technique is most useful if it identifies the statement in the source code the programmer needs to change. However, many FL techniques have a serious limitation: they do not 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. A technical report [35] 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]:

```
  if (n.isCall() && ...)
    return true;
  if (n.isNew() && ...)
    return true;
+ if (n.isDelProp())
+   return true;
  for (Node c = n.getFirstChild(); ...) {
    ...
```

[1] These scores are different than the "suspiciousness score" the FL 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)	**Programs**	**kLOC**	**Artif. faults**	**Real faults**
[18]	C	Tarantula	Siemens	3	122🚹	-
[3]	C	Ochiai, Tarantula	Siemens	3	120🚹	-
[2]	C	Ochiai, Tarantula	Siemens, space	12	128🚹	34
[4]	C	Barinel, Ochiai, Tarantula	Siemens, space, gzip, sed	31	141🚹	38
[5]	C	Tarantula	Concordance	2	200☢	13
[31]	C	Op2, Ochiai, Tarantula	Siemens, space	12	132🚹	32
[33]	C	Metallaxis, Ochiai	Siemens, space, flex, grep, gzip	45	859 🚹,☢	12
[27]	C	Ochiai, Tarantula	Siemens, space, NanoXML, XML-Security	41	164🚹	35
[44]	C	DStar, Ochiai, Tarantula	Siemens, space, ant, flex, grep, gzip, make, sed, Unix	155	436🚹	34
[30]	C	MUSE, Op2, Ochiai	space, flex, grep, gzip, sed	54	11🚹	3
[49]	Java	Ochiai, Tarantula	JExel, JParsec, Jaxen, Commons Codec, Commons Lang, Joda-Time	108	1800☢	-
[19]	C & Java	DStar, Tarantula	printtokens, printtokens2, schedule, schedule2, totinfo, Jtcas, Sorting, NanoXML, XML-Security	32	104🚹	-
[26]	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	2995☢	-
this	Java	{DStar ≈ Ochiai ≈ Barinel ≈ Tarantula}, Op2, Metallaxis, MUSE	JFreeChart, Closure, Commons Lang, Commons Math, Joda-Time	321	-	310

🚹 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.

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 real 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 as any candidate location appears in the FL technique's output.

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) such as a supertype declaration or the data type in a field or variable declaration. In the Defects4J 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-mutatable statements: the mutation-based FL 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. In the Defects4J database of real-world faults [21], 10% of real faults involve a non-mutatable yet executable statement.

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.

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) [3], [18], [31], [44], which is the most studied and evaluated FL technique; and mutation-based fault localization (MBFL techniques for short), which is reported to significantly outperform SBFL techniques [30], [33]. A survey paper lists other types of fault localization techniques [45].

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—that is, to be the root cause of the failures.

1) Spectrum-based FL techniques: Spectrum-based fault localization techniques [3], [18], [31], [44] 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 techniques [45]. 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[18]: $S(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed+passed(s)/totalpassed}$

Ochiai[2]: $S(s) = \frac{failed(s)}{\sqrt{totalfailed\cdot(failed(s)+passed(s))}}$

Op2[31]: $S(s) = failed(s) - \frac{passed(s)}{totalpassed+1}$

Barinel[4]: $S(s) = 1 - \frac{passed(s)}{passed(s)+failed(s)}$

DStar†[44]: $S(s) = \frac{failed(s)^{*}}{passed(s)+(totalfailed-failed(s))}$

†variable $* > 0$. We used $* = 2$, the most thoroughly-explored value.

2) Mutation-based FL techniques: Mutation-based fault localization techniques [30], [33] 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 outcome. 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 yields a different test outcome than executing it on the original program. Our study considered two well-known MBFL techniques: MUSE [30] and Metallaxis [33]. 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's ranking [30] can be obtained by setting $M(m) = failed(m) - \frac{f2p}{p2f} \cdot passed(m)$ 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 $S(s) = \text{avg}_{m \in mut(s)} M(m)$.

Metallaxis [33] uses the same suspiciousness formula as Ochiai: Ochiai^f (the superscript f denoting a reference to the formula rather than the SBFL technique) 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 (*passed(m)* and *totalpassed* are defined similarly). 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 described in our technical report [35] that reduced the runtime by more than an order of magnitude, running every test necessary to compute every MBFL technique's score on all 3390 faults took over 100,000 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 [8]. We collected mutation analysis data using the Major mutation framework [20] (v1.2.1), using all mutation operators it offers.

B. Programs

We used the programs in the Defects4J [21] dataset (v1.1.0), which consists of 395 real faults from 6 open source projects: JFreeChart, Google Closure compiler, Apache Commons Lang, Apache Commons Math, Mockito, and Joda-Time. 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.

TABLE II
HOW WE SELECTED FAULTS FOR OUR REPLICATION STUDIES.

	# faults	
Action	Real	Artif.
Consider all faults from Defects4J	395	4834
Every real fault must correspond to some artificial fault		
Discard faults with deletion-only fix	386	4834
Discard undetectable artificial faults	358	3723
Impose 100,000h timeout	310	2995
Final	**310**	**2995**

Defects4J's patch minimization was performed by three authors of this paper, using both automated analysis (such as delta debugging [50]) and manual analysis to find a minimal patch that they agreed preserved the spirit of the programmer's fix.

Given a minimized patch, we used 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 other statement modified by the patch is a defective statement that a FL tool should report.

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.

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. To verify that each artificial fault has at least one fault-triggering test case as well, we executed the corresponding developer-written test suite and discarded the 23% of artificial faults that were not exposed by any test case. This is on the same order as the results of a study [39] finding that 16% of mutants were undetectable, 45% of which do not change the program semantics.

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 reflect differences in the techniques, not in their implementations. 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?

2) Data: artificial faults: We used the Major tool [23] 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, where fault localization might be easier or harder. Therefore, 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. For each of these pairs of program versions, Defects4J provides 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 the fixed statements.

Our methodology of comparing real faults to artificial faults requires that for every real fault, there is at least one artificial fault for comparison. We discarded 37 real faults that did not fit this criterion, as shown in table II. First, we discarded 9 real faults whose fixes only deleted erroneous code, so there were no fixed statements and no artificial faults can be generated. Then, we discarded artificial faults that are not detected by any tests; we also discarded artificial faults that cause the test suite to time out (e.g., because they introduce infinite loops). This discarded all artificial faults for 28 real faults, so we removed those 28 real faults from the study.

Some faults that do not introduce an infinite loop nonetheless take a very long time during mutation testing. We ran our experiments for about 100,000 hours, and discarded faults whose MBFL analysis had not yet completed.

The output of this process was a set of 2995 artificial faults, corresponding to 310 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: We used three independent, complementary evaluation metrics to rank 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 FLT rank: using each fault to rank the techniques from 1 to 7, and averaging across all artificial faults. "FLT" stands for "fault localization technique".

TABLE III
FAULT LOCALIZATION TECHNIQUES SORTED BY MEAN *EXAM* SCORE OR TOURNAMENT RANKING. "# WORSE" IS THE NUMBER OF OTHER TECHNIQUES THAT ARE STATISTICALLY SIGNIFICANTLY WORSE IN THE TOURNAMENT RANKING.

Artificial Faults			**Real Faults**		
Technique	*EXAM*	# Worse	Technique	*EXAM*	# Worse
Metallaxis	0.0432	5	DStar	0.0404	4
Op2	0.0503	5	Ochiai	0.0405	4
DStar	0.0510	4	Barinel	0.0416	3
Ochiai	0.0514	3	Tarantula	0.0429	2
Barinel	0.0562	2	Op2	0.0476	2
Tarantula	0.0569	1	Metallaxis	0.0753	1
MUSE	0.0781	0	MUSE	0.2061	0

RQ 2: For each pair of techniques compared by prior work (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 mean *EXAM* score metric and tournament ranking metric were perfectly consistent with each other, and produce the ordering shown in the left half of table III. The third metric (mean FLT rank) also agrees perfectly with the others, except that MUSE does best by mean FLT rank and worst by the other two metrics.

As shown by the peaks for the dotted lines in fig. 1, MBFL techniques very often rank the artificially-faulty statement in the top 5. One reason for this is that many artificial faults we generate are caused by "reversible" mutants: mutants that can be exactly canceled by applying a second mutant (e.g., a+b→a-b→a+b). Reversible artificial faults guarantee that MBFL will consider some mutant in the faulty statement that fixes every failing test and that receives 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 10 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. All spectrum-based techniques (except Tarantula) are nearly indistinguishable in fig. 1. We only see statistical significance because of our large number of artificial faults.

Consistency with prior SBFL-SBFL comparisons. Our results agree with all previous comparisons between SBFL

TABLE IV

PREVIOUSLY-REPORTED COMPARISONS, AND OUR RESULTS FOR THOSE COMPARISONS, IN THE BEST-CASE DEBUGGING SCENARIO.

The conclusions are the same for all debugging scenarios [35]. 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)*. The column "95% CI" gives the confidence interval for the difference in means. 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 [26], [27], [31], [44], [49]	**yes**	(-0.23)	[-0.006, -0.005]	(1255–1739–1)	*(insig.)*	*(-0.1)*	[-0.005, 0.000]	(66–232–12)
Barinel > Ochiai [4]	**no**	(0.25)	[0.004, 0.005]	(1–1742–1252)	*(insig.)*	*(0.09)*	[-0.000, 0.003]	(12–233–65)
Barinel > Tarantula [4]	**yes**	*(-0.05)*	[-0.001, -0.000]	(9–2986–0)	*(insig.)*	*(-0.06)*	[-0.003, 0.001]	(1–309–0)
Op2 > Ochiai [31]	**yes**	*(-0.08)*	[-0.002, -0.001]	(456–2525–14)	no	*(0.14)*	[0.002, 0.013]	(60–219–31)
Op2 > Tarantula [30], [31]	**yes**	(-0.23)	[-0.008, -0.006]	(1316–1665–14)	*(insig.)*	*(0.09)*	[-0.001, 0.012]	(71–206–33)
DStar > Ochiai [26], [44]	**yes**	*(-0.12)*	[-0.001, -0.000]	(259–2736–0)	*(insig.)*	*(-0.02)*	[-0.001, 0.001]	(26–273–11)
DStar > Tarantula [19], [26], [44]	**yes**	(-0.24)	[-0.007, -0.005]	(1265–1729–1)	*(insig.)*	*(-0.1)*	[-0.005, 0.000]	(66–229–15)
Metallaxis > Ochiai [33]	yes	*(-0.04)*	[-0.016, -0.001]	(1900–228–867)	**no**	(0.2)	[0.012, 0.042]	(168–18–124)
MUSE > Op2 [30]	**no**	*(0.12)*	[0.020, 0.036]	(1874–120–1001)	**no**	**0.8**	[0.131, 0.173]	(115–2–193)
MUSE > Tarantula [30]	**no**	*(0.09)*	[0.013, 0.029]	(2055–79–861)	**no**	**0.86**	[0.137, 0.178]	(109–2–199)

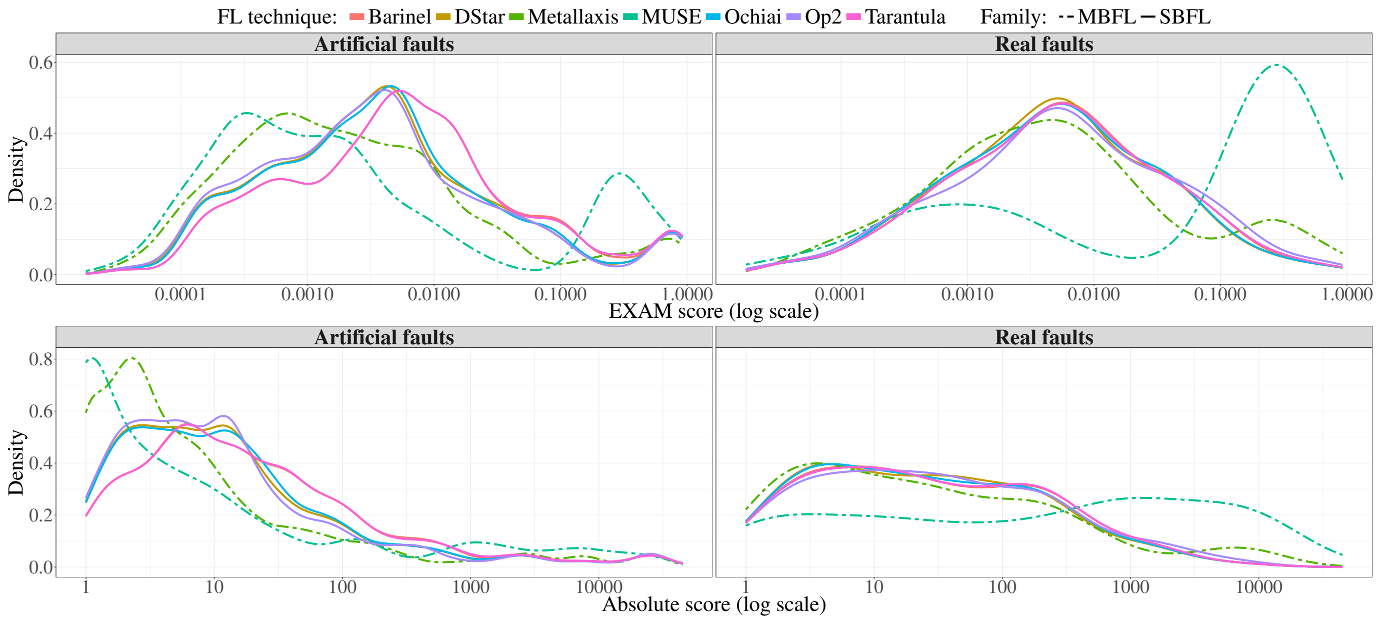

Fig. 1. Distributions of *EXAM* and absolute scores for all FL techniques, considering the best-case debugging scenario and artificial vs. real faults. The absolute score is the first location of any defective statement in the suspiciousness ranking of program statements, computed by a fault localization technique.

techniques, except the claim that Barinel outperforms Ochiai, which our results contradict.

Disagreement with prior SBFL-MUSE comparisons. Prior comparisons found MUSE superior to SBFL techniques. Our results do not support that finding: although MUSE is better on many artificial faults, it does much worse on others. Overall, the differences are practically insignificant.

V. REPLICATION: REAL FAULTS

A. *Subjects*

We evaluated the same techniques, programs, and test suites as described in section III, except that instead of evaluating each technique on the 2995 artificial faults described in section IV, we evaluate them on the corresponding 310 *real* faults.

B. *Methodology*

Our methodology is 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 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 gives almost the same ranking, except that Metallaxis ranks first instead of nearly last. Metallaxis usually has slightly higher scores than any other technique (as shown in fig. 1), giving it a good FLT rank, but it also has more extreme outliers than SBFL techniques, greatly damaging its mean *EXAM* score. For more details, see our technical report [35].

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. All effect sizes are negligible: column "d" is italicized.

Practical significance: MUSE performs poorly. The only practically significant differences show that MUSE performs poorly on real faults (see fig. 1). This is due to almost no real faults being reversible by a single mutant (see section IV-B1).

D. Comparison to artificial faults

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 suggests 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. On real faults, Metallaxis has the best mean FLT rank but one of the worst mean *EXAM* scores.

We repeated the analysis of RQ1, restricted to different categories of real faults. Again, there are no statistically significant relationships between rankings on real and artificial faults, even for single-line faults—which one would expect to be the most similar to mutants—or faults of omission, on which some FL techniques might perform poorly.

E. Controlling for number of samples

Table IV shows statistically significant results for artificial faults, but mostly insignificant results on real faults. It is possible that the results are insignificant because there are so few data points—there are many more artificial than real faults. To investigate this, we averaged results for each artificial fault that corresponds to a single real fault, so there are only 310 datapoints for artificial faults. The results for artificial faults remained statistically significant [35], showing that having 310 data points does not prevent statistically significant results. Furthermore, the effect sizes for real faults are negligible for SBFL comparisons, and the confidence intervals are tight (table IV). Artificial faults are not a good proxy for any single technique: the correlation between each technique's performance on artificial faults and real faults is at most moderate—mostly weak or negligible.

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 overall set of 395 real faults.

A. Subjects

All the techniques of section III-A have the same 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^f)

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

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

Aggregation 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) Interaction 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 an exception 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*. We give 6 ways to define the equivalence classes. All of them define one class each for "pass", "timeout", "JVM crash", and several classes for "exception" (including `AssertionError`). The 6 definitions differ in how they partition exceptions:

1) **exact**: exceptions with the same 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) Aggregation definition: MBFL computes an aggregate statement suspiciousness $S(s)$ 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 statement. This causes MBFL to do quite poorly in the worst-case debugging scenario, when its goal is to find the position of *all* faulty statements.)

B. Methodology

RQ 5: Which technique in this design space performs best on real faults? 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 395 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: What are the most significant design decisions for a FL technique? 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, the DStar technique has the smallest mean *EXAM* score (see table V, with more details in [35]); DStar is statistically significantly better than almost every other technique in the design space (see table VI), and its score is almost twice as good as the best MBFL technique, which closely resembles Metallaxis. For the other two debugging scenarios, Barinel and Ochiai perform best, when instantiated with the "number of statements covered" definition of *totalpassed* and *totalfailed*.

Judged by the mean *EXAM* score, all 12 SBFL techniques are better than the best MBFL technique. However, judged by mean FLT rank, this reverses, and many MBFL techniques are better than any SBFL technique. As seen in table VI, the best MBFL technique by FLT rank for two debugging scenarios uses MUSE's formula and total-definition, but Metallaxis's aggregation and interaction (kill)-definition. (Recall from section IV-B that MUSE's kill-definition tied its performance to fault reversibility. Using a different kill-definition damages its performance on reversible faults, but makes it much better on real faults [35].)

TABLE V
BEST FL TECHNIQUES PER FAMILY ACCORDING TO MEAN *EXAM* SCORE. THE FIRST COLUMN IS RANK AMONG THE 156 TECHNIQUES.

	Family	Formula	Total def.	Interact. def.	Agg. def	*EXAM* score
best-case debugging scenario (localize any defective statement)						
1	SBFL	DStar^f	tests	–	–	0.040
13	MBFL	Ochiai^f	elements	exact	max	0.078
worst-case debugging scenario (localize all defective statements)						
1	SBFL	Barinel^f	elements	–	–	0.191
13	MBFL	Ochiai^f	elements	exact	max	0.245
average-case debugging scenario (localize 50% of the defective statements)						
1	SBFL	Ochiai^f	elements	–	–	0.088
13	MBFL	DStar^f	tests	exact	max	0.129

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

Evaluation metric	Best FL technique					# Better than	$\bar{d}$
	Family	Formula	Total def.	Inter. def.	Agg. def.		
best-case debugging scenario (localize any defective statement)							
Mean *EXAM* score	SBFL	DStar^f	tests	–	–	151/155	−0.47
Mean Rank	MBFL	DStar^f	tests	type	avg	128/155	−0.18
worst-case debugging scenario (localize all defective statements)							
Mean *EXAM* score	SBFL	Barinel^f	elements	–	–	149/155	−0.42
Mean Rank	MBFL	MUSE^f	elements	exact	max	135/155	−0.22
average-case debugging scenario (localize 50% of the defective statements)							
Mean *EXAM* score	SBFL	Ochiai^f	elements	–	–	149/155	−0.47
Mean Rank	MBFL	MUSE^f	elements	exact	max	146/155	−0.25

DStar is statistically significantly better than all but four 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 four other SBFL techniques, including Ochiai and Barinel.

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 VII shows the results, indicating that all factors (including all FL technique parameters, as well as the defect and debugging scenario) have a statistically 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. The choice of the formula accounts for no more than 2% of the non-defect variation in the *EXAM* scores. Furthermore, a post-hoc Tukey test showed that the differences between all formulas are insignificant for SBFL techniques.

TABLE VII
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 *($R^2 = 0.65$)*				
Defect	394	387	58	<0.05
Debugging scenario	2	57.9	1703	<0.05
Formula	5	0.374	4	<0.05
Total definition	1	0.00623	0	*(insig.)*
mbfl *($R^2 = 0.67$)*				
Defect	394	5508	725	<0.05
Debugging scenario	2	718	18614	<0.05
Interaction definition	5	324	3357	<0.05
Formula	5	20.4	211	<0.05
Aggregation definition	1	1.33	69	<0.05
Total definition	1	0.145	8	<0.05
sbfl + mbfl *($R^2 = 0.64$)*				
Defect	394	5595	686	<0.05
Debugging scenario	2	776	18734	<0.05
Family	12	426	1716	<0.05
Formula	5	20.3	196	<0.05
Total definition	1	0.15	7	<0.05

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 *interaction (kill) definition* (for MBFL only).

VII. New techniques

Beyond the quantitative results discussed so far, our studies exposed three limitations of MBFL techniques. (i) MBFL techniques perform poorly on defects that involve unmutatable statements. (ii) MBFL techniques perform poorly when some mutants are covered but not killed. (iii) The run time of MBFL techniques is several orders of magnitude larger than for SBFL techniques, because mutation analysis requires running the entire test suite many times (once per mutant).

We designed several new variants of MBFL to address these limitations, by using coverage information to augment the mutation information:

- MCBFL ("mutant-and-coverage-based FL"), which increases the suspiciousness of mutants covered by failing tests, thus ensuring that mutants covered-but-not-killed by failing tests are more suspicious than ones not even covered;
- MCBFL-hybrid-failover, which uses SBFL to assign suspiciousnesses to unmutatable statements, thus placing them more accurately in the ranking than MBFL can;
- MCBFL-hybrid-avg, which averages each statement's MBFL suspiciousness with the suspiciousness calculated by a SBFL technique;
- MCBFL-hybrid-max, which does the same, but takes the greater of the two suspiciousnesses; and
- MRSBFL, which uses mutation *coverage* information to replace the *kill* information in MBFL, thus requiring only a single run of the test suite, making it as inexpensive as SBFL.

TABLE VIII
PERCENTAGE OF DEFECTS WHOSE DEFECTIVE STATEMENTS APPEAR WITHIN THE TOP-5, TOP-10, AND TOP-200 OF THE TECHNIQUES' SUSPICIOUSNESS RANKING.

	Debugging scenario								
	Best-case dbg. scen.			Worst-case dbg. scen.			Avg-case dbg. scen.		
Technique	Top-5	Top-10	Top-200	Top-5	Top-10	Top-200	Top-5	Top-10	Top-200
MCBFL-hybrid-avg	36%	45%	85%	19%	26%	58%	23%	31%	71%
Metallaxis	29%	39%	77%	16%	22%	47%	18%	27%	63%
DStar	30%	39%	82%	17%	23%	57%	18%	26%	69%

We evaluated these novel techniques on our overall set of 395 real faults, considering all debugging scenarios. Overall, MCBFL-hybrid-avg is better than any other technique in all debugging scenarios, but the difference in *EXAM* score and FLT rank is not practically significant (a technical report gives full experimental results [35]).

Table VIII compares our new MCBFL-hybrid-avg technique to the best SBFL and MBFL techniques in terms of how often they report defective statements in the top 5, 10, or 200 statements. This is relevant because a recent study [24] 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. Another analysis [29] shows that automatic program repair systems perform best when they consider only the top-200 suspicious statements.

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 6 programs, written by different developers and targeting different application domains. Our set of 395 real faults is much larger than all previous studies combined (less than 60 faults). Nonetheless, future research should verify whether our results generalize to other programs and 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. The *EXAM* score may not be the best metric for comparing usefulness of FL techniques by humans: 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 [34]. Our study revolves around the *comparison* of FL techniques rather than their absolute performances. Furthermore, our "mean FLT 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). 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, and what user studies should be done in the future.

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/).

IX. Related work

According to a recent survey [45], the most studied and evaluated fault localization techniques are spectrum-based [3], [18], [25], [44], slice-based [42], [48], model-based [1], [47], and mutation-based [30], [33]. For reasons of space, we discuss the most closely related work; our technical report [35] contains extensive additional discussion.

A. Evaluation of fault localization techniques

Table I references many previous comparisons of FL techniques. Most compare only SBFL techniques, though Jones and Harrold [18] compare Tarantula against other families (slice-based and cause transitions), and the papers presenting MBFL techniques compare against prior SBFL techniques. These studies predominantly use artificial faults. Our findings agree with the results of almost all of these studies on artificial faults, but differ dramatically on real faults.

B. Artificial vs. real faults

It has been very common to evaluate and compare fault localization techniques using manually-seeded artificial faults (e.g., Siemens set [3], [18], [33], [38]) or mutations (e.g., [12], [38], [44]) 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 [41], 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/store 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. Cifuentes et al. [9] found that 5 static bug detection tools achieved an average accuracy of 20% on real bugs but 46% on synthetic bugs.

The more general question of whether mutants are representative of real faults has been subject to thorough investigation. While Gopinath et al. [14] 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. [6] 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. [7] 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 [17], suggesting a need for stronger mutation operators.

However, Namin et al. [32] studied the same set of programs as previous studies [6], 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. 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.

Acknowledgments

This material is based on research sponsored by Air Force Research Laboratory and DARPA under agreement numbers FA8750-12-2-0107, FA8750-15-C-0010, and FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This material is based upon work supported by the ERDF's COMPETE 2020 Programme under project No. POCI-01-0145-FEDER-006961 and FCT under project No. UID/EEA/50014/2013.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. BegBunch: Benchmarking for C bug detection tools. In *DEFECTS*, pages 16–20, July 2009.

[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] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713. ACM, 2016.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating & improving fault localization techniques. Technical Report UW-CSE-16-08-03, U. Wash. Dept. of Comp. Sci. & Eng., Seattle, WA, USA, Sep. 2016. Revised Feb. 2017.

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

[37] 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.

[38] 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.

[39] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability*, 23(5):353–374, 2013.

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TSE*, 28(3):183–200, Feb. 2002.