An Empirical Study of Fault Localization in Python Programs

Mohammad Rezaalipour · Carlo A. Furia

Abstract Despite its massive popularity as a programming language, especially in novel domains like data science programs, there is comparatively little research about fault localization that targets Python. Even though it is plausible that several findings about programming languages like C/C++ and Java—the most common choices for fault localization research—carry over to other languages, whether the dynamic nature of Python and how the language is used in practice affect the capabilities of classic fault localization approaches remain open questions to investigate.

This paper is the first multi-family large-scale empirical study of fault localization on real-world Python programs and faults. Using Zou et al.'s recent large-scale empirical study of fault localization in Java [78] as the basis of our study, we investigated the effectiveness (i.e., localization accuracy), efficiency (i.e., runtime performance), and other features (e.g., different entity granularities) of seven well-known fault-localization techniques in four families (spectrum-based, mutation-based, predicate switching, and stack-trace based) on 135 faults from 13 open-source Python projects from the BugsInPy curated collection [67].

The results replicate for Python several results known about Java, and shed light on whether Python's peculiarities affect the capabilities of fault localization. The replication package that accompanies this paper includes detailed data about our experiments, as well as the tool FauxPy that we implemented to conduct the study.

Keywords Fault localization · Debugging · Python · Empirical study

1 Introduction

It is commonplace that debugging is an activity that takes up a disproportionate amount of time and resources in software development [41]. This also explains the popularity of *fault localization* as a research subject in software engineering: identifying locations in a program's source code that are implicated in some observed failures (such as crashes or other kinds of runtime errors) is a key step of debugging. This paper contributes to the empirical knowledge about the capabilities of fault localization techniques, targeting the Python programming language.

Despite the massive amount of work on fault localization (see Section 3) and the popularity of the Python programming language,[1][2] most empirical studies of fault localization target languages like Java or C. This leaves open the question of whether Python's characteristics—such as the fact that it is dynamically typed, or that it is dominant in certain application domains such as data science—affect the capabilities of classic fault localization techniques—developed and tested primarily on different kinds of languages and programs.

This paper fills this knowledge gap: to our knowledge, it is the first multi-family large-scale empirical study of fault localization in real-world Python programs. The starting point is Zou et al.'s recent extensive study [78] of fault localization for Java. This paper's main contribution is a differentiated conceptual replication [30] of Zou et al.'s study, sharing several of its features: *i*) it experiments with several different families (spectrum-based, mutation-based, predicate switching, and stack-trace-based) of fault localization techniques; *ii*) it targets a large number of faults in real-world projects (135 faults in 13 projects) ; *iii*) it studies fault localization effectiveness at different

M. Rezaalipour
Software Institute, Università della Svizzera italiana, Lugano, Switzerland
E-mail: rezaam@usi.ch

C. A. Furia
Software Institute, Università della Svizzera italiana, Lugano, Switzerland
`https://bugcounting.net/`

[1] TIOBE language popularity index: `https://www.tiobe.com/tiobe-index/`
[2] Popularity of Programming Language Index: `https://pypl.github.io/PYPL.html`

granularities (statement, function, and module); *iv*) it considers combinations of complementary fault localization techniques. The fundamental novelty of our replication is that it targets the Python programming language; furthermore, *i*) it analyzes fault localization effectiveness of different kinds of faults and different categories of projects; *ii*) it estimates the contributions of different fault localization features by means of regression statistical models; *iii*) it compares its main findings for Python to Zou et al.'s [78] for Java.

The main *findings* of our Python fault localization study are as follows:

1. Spectrum-based fault localization techniques are the most effective, followed by mutation-based fault localization techniques.
2. Predicate switching and stack-trace fault localization are considerably less effective, but they can work well on small sets of faults that match their characteristics.
3. Stack-trace is by far the fastest fault localization technique, predicate switching and mutation-based fault localization techniques are the most time consuming.
4. Bugs in data-science related projects tend to be harder to localize than those in other categories of projects.
5. Combining fault localization techniques boosts their effectiveness with only a modest hit on efficiency.
6. The main findings about relative effectiveness still hold at all granularity levels.
7. Most of Zou et al. [78]'s findings about fault localization in Java carry over to Python.

A practical challenge to carry out a large-scale fault localization study of Python projects was that, at the time of writing, there were no open-source tools that support a variety of fault localization techniques for this programming language. Thus, to perform this study, we implemented FauxPy: a fault-localization tool for Python that supports seven fault localization techniques in four families, is highly configurable, and works with the most common Python unit testing frameworks (such as Pytest and Unittest). The present paper is not a paper about FauxPy, which we plan to present in detail in a separate publication. Nevertheless, we briefly discuss the key features of FauxPy, and make the tool available as part of this paper's replication package—which also includes all the detailed experimental artifacts and data that support further independent analysis and replicability.

The rest of the paper is organized as follows. Section 2 presents the fault localization techniques that fall within the scope of the empirical study, and outlines FauxPy's features. Section 3 summarizes the most relevant related work in fault localization, demonstrating how Python is underrepresented in this area. Section 4 presents in detail the paper's research questions, and the experimental methodology that we followed to answer them. Section 5 details the experimental results for each investigated research question, and presents any limitations and threats to the validity of the findings. Section 6 concludes with a high-level discussion of the main results, and of possible avenues for future work.

Replication package. For reproducibility, all experimental artifacts of this paper's empirical study, and the implementation of the FauxPy tool, are available:

`https://doi.org/10.6084/m9.figshare.23254688`

2 Fault Localization and FauxPy

Fault localization techniques [73, 71] relate program failures (such as crashes or assertion violations) to faulty locations in the program's source code that are responsible for the failures. Concretely, a fault localization technique L assigns a *suspiciousness score* $L_T(e)$ to any program entity e—usually, a location, function, or module—given test inputs T that trigger a failure in the program. The suspiciousness score $L_T(e)$ should be higher the more likely e is the location of a fault that is ultimately responsible for the failure. Thus, a list of all program entities $e_1, e_2, \ldots$ ordered by decreasing suspiciousness score $L_T(e_1) \geq L_T(e_2) \geq \ldots$ is fault localization technique L's overall output.

Let $T = P \cup F$ be a set of tests partitioned into passing P and failing F, such that $F \neq \emptyset$—there is at least one failing test—and all failing tests originate in the same fault. Tests T and a program p are thus the target of a single fault localization run. Then, fault localization techniques differ in what kind of information they extract from T and p to compute suspiciousness scores. A fault localization *family* is a group of techniques that combine the same kind of information according to different formulas. Sections 2.1–2.4 describe four common FL families that comprise a total of seven FL families. As Section 2.5 further explains, a FL technique's *granularity* denotes the kind of program entities it analyzes for suspiciousness—from individual program locations to functions or files/modules. Some FL techniques are only defined for a certain granularity level, whereas others can be applied to different granularities.

While FL techniques are usually applicable to any programming language, we could not find any comprehensive implementation of the most common fault localization techniques for Python at the time of writing. Therefore, we implemented FauxPy—an automated fault localization tool for Python implementing several widely used techniques—and used it to perform the empirical study described in the rest of the paper. Section 2.6 outlines FauxPy's main features and some details of its implementation.

$$\mathsf{Tarantula}_T(e) = \frac{F^+(e)/|F|}{F^+(e)/|F| + P^+(e)/|P|} \tag{1}$$

$$\mathsf{Ochiai}_T(e) = \frac{F^+(e)}{\sqrt{|F| \times (F^+(e) + P^+(e))}} \tag{2}$$

$$\mathsf{DStar}_T(e) = \frac{(F^+(e))^2}{P^+(e) + F^-(e)} \tag{3}$$

Figure 1: SBFL formulas to compute the suspiciousness score of an entity e given tests $T = P \cup F$ partitioned into passing P and failing F. All formulas compute a score that is higher the more failing tests $F^+(e)$ cover e, and lower the more passing tests $P^+(e)$ cover e—capturing the basic heuristics of SBFL.

$$\mathsf{Metallaxis}_T(m) = \frac{F^k_\sim(m)}{\sqrt{|F| \times (F^k_\sim(m) + P^k(m))}} \qquad \mathsf{Metallaxis}_T(e) = \max_{\substack{m \in M \\ m \text{ mutates } e}} \mathsf{Metallaxis}_T(m) \tag{4}$$

$$\mathsf{Muse}_T(m) = \frac{F^k(m) - P^k(m) \times \sum_{n \in M} F^k(n) / \sum_{n \in M} P^k(n)}{|F|} \qquad \mathsf{Muse}_T(e) = \operatorname*{mean}_{\substack{m \in M \\ m \text{ mutates } e}} \mathsf{Muse}_T(m) \tag{5}$$

Figure 2: MBFL formulas to compute the suspiciousness score of a mutant m given tests $T = P \cup F$ partitioned into passing P and failing F. All formulas compute a score that is higher the more failing tests $F^k(m)$ kill m, and lower the more passing tests $P^k(m)$ kill m—capturing the basic heuristics of mutation analysis. On the right, MBFL formulas to compute the suspiciousness score of a program entity e by aggregating the suspiciousness score of all mutants $m \in M$ that modified e in the original program.

2.1 Spectrum-Based Fault Localization

Techniques in the spectrum-based fault localization (SBFL) family compute suspiciousness scores based on a program's spectra [52]—in other words, its concrete execution traces. The key heuristics of SBFL techniques is that a program entity's suspiciousness is higher the more often the entity is covered (reached) by failing tests and the less often it is covered by passing tests. The various techniques in the SBFL family differ in what formula they use to assign suspiciousness scores based on an entity's coverage in passing and failing tests.

Given tests $T = P \cup F$ as above, and a program entity e: *i)* $P^+(e)$ is the number of passing tests that cover e; *ii)* $P^-(e)$ is the number of passing tests that do not cover e; *iii)* $F^+(e)$ is the number of failing tests that cover e; *iv)* and $F^-(e)$ is the number of failing tests that do not cover e. Figure 1 shows how Tarantula [26], Ochiai [1], and DStar [70]—three widely used SBFL techniques [49]—compute suspiciousness scores given this coverage information. DStar's formula (3), in particular, takes the second power of the numerator, as recommended by other empirical studies [78, 70].[3]

2.2 Mutation-Based Fault Localization

Techniques in the mutation-based fault localization (MBFL) family compute suspiciousness scores based on mutation analysis [25], which generates many *mutants* of a program p by applying random transformations to it (for example, change a comparison operator $<$ to $\leq$ in an expression). A mutant m of p is thus a variant of p whose behavior differs from p's at, or after, the location where m differs from p. The key idea of mutation analysis is to collect information about p's runtime behavior based on how it differs from its mutants'. Accordingly, when a test t behaves differently on p than on m (for example, p passes t but m fails it), we say that *t kills m*.

To perform fault localization on a program p, MBFL techniques first generate a large number of mutants $M = \{m_1, m_2, \ldots\}$ of p by systematically applying each mutation operator to each statement in p that is executed in any failing test F. Then, given tests $T = P \cup F$ as above, and a mutant $m \in M$: *i)* $P^k(m)$ is the number of tests that p passes but m fails (that is, the tests in P that *kill m*); *ii)* $F^k(m)$ is the number of tests that p fails but m passes (that is, the tests in F that *kill m*); *iii)* and $F^k_\sim(m)$ is the number of tests that p fails and behave differently on m, either

[3]Suspiciousness score formulas are typically expressed as *ratios*; when a denominator is zero, this leads to an undefined score. There are different strategies to account for suspiciousness scores in these degenerate cases [57]. In this paper, we implicitly add a small constant $\epsilon = 0.1$ to the denominator of every suspiciousness score formula. When the denominator is not zero, adding ϵ is practically irrelevant; when the denominator is zero and the numerator is also zero, adding ϵ gives a very low suspiciousness of zero (which reflects that the entity is hardly covered by any tests); when the denominator is zero and the numerator is positive, adding ϵ gives a large suspiciousness (which reflects that the entity is only covered by failing tests).

because they pass on *m* or because they still fail but lead to a different stack trace (this is a *weaker* notion of tests that *kill m* [45]). Figure 2 shows how Metallaxis [45] and Muse [42]—two widely used MBFL techniques—compute suspiciousness scores of each *mutant* in M.

Metallaxis's formula (4) is formally equivalent to Ochiai's—except that it is computed for each mutant and measures *killing* tests instead of *covering* tests. In Muse's formula (5), $\sum_{n \in M} F^k(n)$ is the total number of failing tests in F that kill any mutant in M, and $\sum_{n \in M} P^k(n)$ is the total number of passing tests in P that kill any mutant in M (these are called *f2p* and *p2f* in Muse's paper [42]).

Finally, MBFL computes a suspiciousness score for a program entity e by aggregating the suspiciousness scores of all mutants that modified e in the original program p; when this is the case, we say that a mutant *m mutates e*. The right-hand side of Figure 2 shows Metallaxis's and Muse's suspiciousness formulas for entities: Metallaxis (4) takes the largest (maximum) mutant score, whereas Muse (5) takes the average (mean) of the mutant scores.

2.3 Predicate Switching

The predicate switching (PS) [74] fault localization technique identifies *critical predicates*: branching conditions (such as those of **`if`** and **`while`** statements) that are related to a program's failure. PS's key idea is that if forcibly changing a predicate's value turns a failing test into passing one, the predicate's location is a suspicious program entity.

For each failing test $t \in F$, PS starts from t's execution trace (the sequence of all statements executed by t), and finds t's subsequence $b^t_1\, b^t_2\, \ldots$ of *branching* statements. Then, by instrumenting the program p under analysis, it generates, for each branching statement b^t_k, a new execution of t where the *predicate* (branching condition) c^t_k evaluated by statement b^t_k is forcibly *switched* (negated) at runtime (that is, the new execution takes the *other* branch at b^t_k). If switching predicate c^t_k makes the test execution pass, then c^t_k is a *critical predicate*. Finally, PS assigns a (positive) suspiciousness score to all critical predicates in all tests F: $\mathsf{PS}_F(c^t_k)$ is higher, the fewer critical predicates are evaluated between c^t_k and the failure location when executing $t \in F$ [78].[4] For example, the most suspicious program entity e will be the location of the last critical predicate evaluated before any test failure.

PS has some distinctive features compared to other FL techniques. First, it only uses failing tests for its dynamic analysis; any passing tests P are ignored. Second, the only program entities it can report as suspicious are locations of predicates; thus, it usually reports a shorter list of suspicious locations than SBFL and MBFL techniques. Third, while MBFL mutates program code, PS dynamically mutates individual *program executions*. For example, suppose that a loop **`while`** $c\!:\!B$ executes its body B twice—and hence, the loop condition c is evaluated three times—in a failing test. Then, PS will generate three variants of this test execution: *i)* one where the loop body never executes (c is false the first time it is evaluated); *ii)* one where the loop body executes once (c is false the second time it is evaluated); *iii)* one where the loop body executes three or more times (c is true the third time it is evaluated).

2.4 Stack Trace Fault Localization

When a program execution fails with a crash (for example, an uncaught exception), the language runtime usually prints its stack trace (the chain of methods active when the crash occurred) as debugging information to the user. In fact, it is known that stack trace information helps developers debug failing programs [5]; and a bug is more likely to be fixed if it is close to the top of a stack trace [59]. Based on these empirical findings, Zou et al. [78] proposed the stack trace fault localization technique (ST), which uses the simple heuristics of assigning suspiciousness based on how close a program entity is to the top of a stack trace.

Concretely, given a failing test $t \in F$, its *stack trace* is a sequence $f_1\, f_2\, \ldots$ of the stack frames of all functions that were executing when t terminated with a failure, listed in reverse order of execution; thus, f_1 is the most recently called function, which was directly called by f_2, and so on. ST assigns a (positive) suspiciousness score to any program entity e that belongs to any function f_k in t's stack trace: $\mathsf{ST}_t(e) = 1/k$, so that e's suspiciousness is higher, the closer to the failure e's function was called.[5] In particular, the most suspicious program entities will be all those in the function f_1 called in the top stack frame. Then, the overall suspiciousness score of e is the maximum in all failing tests F: $\mathsf{ST}_F(e) = \max_{t \in F} \mathsf{ST}_t(e)$.

2.5 Granularities

Fault localization *granularity* refers to the kinds of program entity that a FL technique ranks. The most widely studied granularity is *statement-level*, where each statement in a program may receive a different suspiciousness

[4]The actual value of the suspiciousness score is immaterial, as long as the resulting ranking is consistent with this criterion. In FAUXPY's current implementation, $\mathsf{PS}_F(c^t_k) = 1/10^d$, where d is the number of critical predicates *other than* c^t_k evaluated after c^t_k in t.

[5]As in PS, the actual value of the suspiciousness score is immaterial, as long as the resulting ranking is consistent with this criterion.

score [49, 70]. However, coarser granularities have also been considered, such as *function-level* (also called method-level) [3, 72] and *module-level* (also called file-level) [55, 77].

In practice, implementations of FL techniques that support different levels of granularity focus on the finest granularity (usually, statement-level granularity), whose information they use to perform FL at coarser granularities. Namely, the suspiciousness of a function is the maximum suspiciousness of any statements in its definition; and the suspiciousness of a module is the maximum suspiciousness of any functions belonging to it.[6]

2.6 FauxPy: Features and Implementation

Despite its popularity as a programming language, we could not find off-the-shelf implementations of fault localization techniques for Python at the time of writing [57]. The only exception is CharmFL [21]—a plugin for the PyCharm IDE—which only implements SBFL techniques. Therefore, to conduct an extensive empirical study of FL in Python, we implemented FauxPy: a fault localization tool for Python programs.

FauxPy supports all seven FL techniques described in Sections 2.1–2.4; it can localize faults at the level of statements, functions, or modules (Section 2.5). To make FauxPy a flexible and extensible tool, easy to use with a variety of other commonly used Python development tools, we implemented it as a stand-alone command-line tool that works with tests in the formats supported by Pytest, Unittest, and Hypothesis [40]—three popular Python testing frameworks.

While running, FauxPy stores intermediate analysis data in an SQLite database; upon completing a FL localization run, it returns to the user a human-readable summary—including suspiciousness scores and ranking of program entities. The database improves performance (for example by caching intermediate results) but also facilitates *incremental* analyses—for example, where we provide different batches of tests in different runs.

FauxPy's implementation uses Coverage.py [4]—a popular code-coverage measurement library—to collect the execution traces needed for SBFL and MBFL. It also uses the state-of-the-art mutation-testing framework Cosmic Ray [8] to generate mutants for MBFL; since Cosmic Ray is easily configurable to use some or all of its mutation operators—or even to add new user-defined mutation operators—FauxPy's MBFL implementation is also fully configurable. To implement PS in FauxPy, we developed an instrumentation library that can selectively change the runtime value of predicates in different runs as required by the PS technique. The implementation of FauxPy is available as open-source (see this paper's replication package).

3 Related Work

Fault localization has been an intensely researched topic for over two decades, whose popularity does not seem to wane [71]. This section summarizes a selection of studies that are directly relevant for the paper; Wong's recent survey [71] provides a broader summary for interested readers.

Spectrum-based fault localization. The Tarantula SBFL technique [26] was one of the earliest, most influential FL techniques, also thanks to its empirical evaluation showing it is more effective than other competing techniques [51, 7]. The Ochiai SBFL technique [1] improved over Tarantula, and it often still is considered the "standard" SBFL technique.

These earlier empirical studies [26, 1], as well as other contemporary and later studies of FL [45], used the Siemens suite [20]: a set of seven small C programs with seeded bugs. Since then, the scale and realism of FL empirical studies has significantly improved over the years, targeting real-world bugs affecting projects of realistic size. For example, Ochiai's effectiveness was confirmed [33] on a collection of more realistic C and Java programs [12]. When Wong et al. [70] proposed DStar, a new SBFL technique, they demonstrated its capabilities in a sweeping comparison involving 38 other SBFL techniques (including the "classic" Tarantula and Ochiai). In contrast, numerous empirical results about fault localization in Java based on experiments with artificial faults were found not to hold to experiments with real-world faults [49] using the Defects4J curated collection [28].

Mutation-based fault localization. With the introduction of novel fault localization families—most notably, MBFL—empirical comparison of techniques belonging to different families became more common [42, 45, 49, 78]. The Muse MBFL technique was introduced to overcome a specific limitation of SBFL techniques: the so-called "tie set problem". This occurs when SBFL assigns the same suspiciousness score to different program entities, simply because they belong to the same simple control-flow block (see Section 2.1 for details on how SBFL works). Metallaxis-FL [45] (which we simply call "Metallaxis" in this paper) is another take on MBFL that can improve over SBFL techniques.

[6]Other approaches aggregate the suspiciousness scores of finer-granularity entities by average or by minimum. We take the maximum for consistency with Zou et al. [78].

The comparison between MBFL and SBFL is especially delicate given how MBFL works. As demonstrated by Pearson et al. [49], MBFL's effectiveness crucially depends on whether it is applied to bugs that are "similar" to those introduced by its mutation operators. This explains why the MBFL studies targeting artificially seeded faults [42, 45] found MBFL to outperform SBFL; whereas studies targeting real-world faults [49, 78] found the opposite to be the case—a result also confirmed by the present paper in Section 5.1.

Mutation testing. MBFL techniques rely on mutation testing to generate mutants of a faulty program that may help locate the fault. Therefore, the selection of mutation operators that are used for mutation testing impacts the effectiveness of MBFL techniques. Research in mutation testing has grown considerably in the last decade, developing a large variety of mutation operators tailored to specific programming languages, applications, and faults [46]. Despite these recent developments, the fundamental set of mutation operators introduced in Offut et al.'s seminal work [44] remains the basis of basically every application to mutation testing. These fundamental operators generate mutants by modifying or removing arithmetic, logical, and relational operators, as well as constants and variables in a program, and hence are widely applicable and domain-agnostic. Notably, the Cosmic Ray [8] Python mutation testing framework (used in our implementation of FauxPy), the two other popular Python mutation testing frameworks MutPy [11] and mutmut,[7] as well as the popular Java mutation testing frameworks Pitest[8], MuJava [39] and Major [27] (the latter used in Zou et al.'s MBFL experiments [78]) all offer Offut et al.'s fundamental operators. This helps make experiments with mutation testing techniques meaningfully comparable.

Empirical comparisons. This paper's study design is based on Zou et al.'s empirical comparison of fault localization on Java programs [78]. We chose their study because it is fairly recent (it was published in 2021), it is comprehensive (it targets 11 fault localization techniques in seven families, as well as combinations of some of these techniques), and it targets realistic programs and faults (357 bugs in five projects from the Defects4J curated collection).

Ours is a differentiated conceptual replication [30] of Zou et al.'s study [78]. We target a comparable number of subjects (135 BugsInPy [67] bugs vs. 357 Defects4J [28] bugs) from a wide selection of projects (13 real-world Python projects vs. five real-world Java projects). We study [78]'s four main fault localization families SBFL, MBFL, PS, and ST, but we exclude three other families that featured in their study: DS (dynamic slicing [18]), IRBFL (Information retrieval-based fault localization [77]), and HBFL (history-based fault localization [50]). IRBFL and HBFL were shown to be scarcely effective by Zou et al. [77], and rely on different kinds of artifacts that may not always be available when dynamically analyzing a program as done by the other "mainstream" fault localization techniques. Namely, IRBFL analyzes bug reports, which may not be available for all bugs; HBFL mines commit histories of programs. In contrast, our study only includes techniques that solely rely on *tests* to perform fault localization; this help make a comparison between techniques consistent. Finally, we excluded DS for practical reasons: implementing it requires accurate data- and control-dependency static analyses [73]. These are available in languages like Java through widely used frameworks like Soot [64, 32]; in contrast, Python currently offers few mature static analysis tools (e.g, Scalpel [34]), none with the features required to implement DS. Unfortunately, dynamic slicing has been implemented for Python in the past [6] but no implementation is publicly available; and building it from scratch is outside the present paper's scope.

Python fault localization. Despite Python's popularity as a programming language, the vast majority of fault localization empirical studies target other languages—mostly C, C++, and Java. To our knowledge, CharmFL [63, 21] is the only available implementation of fault localization techniques for Python; the tool is limited to SBFL techniques. We could not find any realistic-size empirical study of fault localization using Python programs comparing techniques of different families. This gap in both the availability of tools [57] and the empirical knowledge about fault localization in Python motivated the present work.

Note that numerous recent empirical studies looked into fault localization for deep-learning models implemented in Python [13, 17, 76, 75, 58, 65]. This is a very different problem, using very different techniques, than "classic" program-based fault localization, which is the topic of our paper.

Deep learning-based fault localization. Deep learning models have recently been applied to the software fault localization problem. The key idea of techniques such as DeepFL [36], GRACE [38], and DEEPRL4FL [37] is to train a deep learning model to identify suspicious locations, giving it as input coverage information, as well as other encoded information about the source code of the faulty programs (such as the data and control-flow dependencies). While these approaches are promising, we could not include them in our empirical study since they do not have the same level of maturity as the other "classic" FL techniques we considered. First, DeepFL and GRACE only work at function-level granularity, whereas the bulk of FL research targets statement-level granularity. Second,

[7] `https://mutmut.readthedocs.io`

[8] `https://pitest.org`

there are no reference implementations of techniques such as DEEPRL4FL that we can use for our experiments.[9] Third, the performance of a deep learning-based technique usually depends on the training set. Fourth, training a deep learning model is usually a time consuming process; how to account for this overhead when comparing efficiency is tricky.

Nevertheless, our empirical study does feature one FL technique that is based on machine learning: CombineFL, which is Zou et al.'s application of learning to rank to fault localization [78]. The same paper also discusses how CombineFL outperforms other state-of-the-art machine learning-based fault localization techniques such as MULTRIC [35], Savant [3], TraPT [35], and FLUCCS [61]. Therefore, CombineFL is a valid representative of the capabilities of pre-deep learning machine learning FL techniques.

Python vs. Java SBFL comparison. To our knowledge, Widyasari et al.'s recent empirical study of spectrum-based fault localization [68] is the only currently available large-scale study targeting real-world Python projects. Like our work, they use the bugs in the BugsInPy curated collection as experimental subjects [67]; and they compare their results to those obtained by others for Java [49]. Besides these high-level similarities, the scopes of our study and Widyasari et al.'s are fundamentally different: *i*) We are especially interested in comparing fault localization techniques in different *families*; they consider exclusively five *spectrum*-based techniques, and drill down into the relative performance of these techniques. *ii*) Accordingly, we consider orthogonal categorization of bugs: we classify bugs (see Section 4.3) according to characteristics that match the capabilities of different fault-localization families (e.g., stack-trace fault localization works for bugs that result in a crash); they classify bugs according to syntactic characteristics (e.g., multi-line vs. single-line patch). *iii*) Most important, even though both our paper and Widyasari et al.'s compare Python to Java, the framing of our comparisons is quite different: in Section 5.6, we compare our findings about fault localization in Python to Zou et al. [78]'s findings about fault localization in Java; for example, we confirm that SBFL techniques are generally more effective than MBFL techniques in Python, as they were found to be in Java. In contrast, Widyasari et al. directly compare various SBFL effectiveness metrics they collected on Python programs against the same metrics Pearson et al. [49] collected on Java programs; for example, Widyasari et al. report that the percentage of bugs in BugsInPy that their implementation of the Ochiai SBFL technique correctly localized within the top-5 positions is considerably lower than the percentage of bugs in Defects4J that Pearson et al.'s implementation of the Ochiai SBFL technique correctly localized within the top-5.

It is also important to note that there are several technical differences between ours and Widyasari et al.'s methodology. First, we handle ties between suspiciousness scores by computing the E_{inspect} rank (described in Section 4.5); whereas they use average rank (as well as other effectiveness metrics). Even though we also take our subjects from BugsInPy, we carefully selected a subset of bugs that are fully analyzable on our infrastructure with all fault localization techniques we consider (Section 4.1, Section 4.7); whereas they use all BugsInPy available bugs. The selection of subjects is likely to impact the value of *some* metrics more than others (see Section 4.5); for example, the exam score is undefined for bugs that a fault localization technique cannot localize, whereas the top-k counts are lower the more faults cannot be localized. These and numerous other differences make our results and Widyasari et al.'s incomparable and mostly complementary. A replication of their comparison following our methodology is an interesting direction for future work, but clearly outside the present paper's scope. In Section 6.1 we present some additional data, and outline a few directions for future work that are directly inspired by Widyasari et al.'s study [68].

4 Experimental Design

Our experiments assess and compare the effectiveness and efficiency of the seven FL techniques described in Section 2, as well as of their combinations, on real-world Python programs and faults. To this end, we target the following research questions:

RQ1. How *effective* are the fault localization techniques?
RQ1 compares fault localization techniques according to how accurately they identify program entities that are responsible for a fault.

RQ2. How *efficient* are the fault localization techniques?
RQ2 compares fault localization techniques according to their running time.

RQ3. Do fault localization techniques behave differently on *different* faults?
RQ3 investigates whether the fault localization techniques' effectiveness and efficiency depend on which kinds of faults and programs it analyzes.

RQ4. Does *combining* fault localization techniques improve their effectiveness?
RQ4 studies whether combining the information of different fault localization techniques for the same faults improves the effectiveness compared to applying each technique in isolation.

[9]The replication package of DEEPRL4FL [37] is not available at the time of writing.

RQ5. How does program entity *granularity* impact fault localization effectiveness?
RQ5 analyzes the relation between effectiveness and granularity: does the relative effectiveness of fault localization techniques change as they target coarser-grained program entities?
RQ6. Are fault localization techniques as effective on Python programs as they are on *Java* programs?
RQ6 compares our overall results to Zou et al. [78]'s, exploring similarities and differences between Java and Python programs.

4.1 Subjects

To have a representative collection of realistic Python bugs,[10] we used BugsInPy [67], a curated dataset of real bugs collected from real-world Python projects, with all the information needed to reproduce the bugs in controlled experiments. Table 1 overviews BugsInPy's 501 bugs from 17 projects.

Project category. Columns CATEGORY in Table 1 and Table 2 partition all BugsInPy projects into four non-overlapping categories:

Command line (CL) projects consist of tools mainly used through their command line interface.
Development (DEV) projects offer libraries and utilities useful to software developers.
Data science (DS) projects consist of machine learning and numerical computation frameworks.
Web (WEB) projects offer libraries and utilities useful for web development.

We classified the projects according to their description in their respective repositories, as well as how they are presented in BugsInPy. Like any classification, the boundaries between categories may be somewhat fuzzy, but the main focus of most projects is quite obvious (such as DS for keras and pandas, or CL for youtube-dl).

Unique bugs. Each bug $b = \langle p_b^-, p_b^+, F_b, P_b \rangle$ in BugsInPy consists of: *i*) a *faulty* version p_b^- of the project, such that tests in F_b all fail on it (all due to the same root cause); *ii*) a *fixed* version p_b^+ of the project, such that all tests in $F_b \cup P_b$ pass on it; *iii*) a collection of *failing* F_b and *passing* P_b tests, such that tests in P_b pass on both the faulty p_b^- and fixed p_b^+ versions of the project, whereas tests in F_b fail on the faulty p_b^- version and pass on the fixed p_b^+ version of the project.

Bug selection. Despite BugsInPy's careful curation, several of its bugs cannot be reproduced because their dependencies are missing or no longer available; this is a well-known problem that plagues reproducibility of experiments involving Python programs [43]. In order to identify which BugsInPy bugs were reproducible at the time of our experiments on our infrastructure, we took the following steps for each bug b:

i) Using BugsInPy's scripts, we generated and executed the faulty p_b^- version and checked that tests in F_b fail whereas tests in P_b pass on it; and we generated and executed the fixed p_b^+ version and checked that all tests in $F_b \cup P_b$ pass on it. Out of all of BugsInPy's bugs, 120 failed this step; we did not include them in our experiments.
ii) Python projects often have two sets of dependencies (*requirements*): one for users and one for developers; both are needed to run fault localization experiments, which require to instrument the project code. Another 39 bugs in BugsInPy miss some development dependencies; we did not include them in our experiments.
iii) Two bugs resulted in an empty ground truth (Section 4.2): essentially, there is no way of localizing the fault in p_b^-; we did not include these bugs in our experiments.

This resulted in $501 - 120 - 39 - 2 = 340$ bugs in 13 projects (all but ansible, matplotlib, PySnooper, and scrapy) that we could reproduce in our experiments.

However, this is still an impractically large number: just *reproducing* each of these bugs in BugsInPy takes nearly a full week of running time, and each FL experiment may require to rerun the same tests several times (hundreds of times in the case of MBFL). Thus, we first discarded 27 bugs that each take more than 48 hours to reproduce. We estimate that including these 27 bugs in the experiments would have taken over 14 CPU-months just for the MBFL experiments—not counting other FL techniques, nor the time for setup and dealing with unexpected failures.

Running all the fault localization experiments for each of the remaining $313 = 340 - 27$ bugs takes approximately eleven CPU-hours, for a total of nearly five CPU-months. We selected 135 bugs out of the 313 using stratified random sampling with the four project categories as the "strata", picking: 43 bugs in category CL, 30 bugs in category DEV, 42 bugs in category DS, and 20 bugs in category WEB. This gives us a still sizable, balanced, and representative[11] sample of all bugs in BugsInPy, which we could exhaustively analyze in around two CPU-months

[10]Henceforth, we use the terms "bug" and "fault" as synonyms.

[11]For example, this sample size is sufficient to estimate a ratio with up to 5.5% error and 90% probability with the most conservative (i.e., 50%) a priori assumption [9].

PROJECT	KLOC	|F|	|M|	BUGS	SUBJECTS	TESTS	TEST KLOC	CATEGORY	DESCRIPTION
ansible	82.6	3 713	493	18	0	1 830	103.1	DEV	IT automation platform
black	93.5	421	27	23	13	153	6.8	DEV	Code formatter
cookiecutter	1.6	62	18	4	4	218	4.1	DEV	Developer tool
fastapi	4.7	160	40	16	13	595	16.8	WEB	Web framework for building APIs
httpie	3.5	197	34	5	4	217	2.4	CL	Command-line HTTP client
keras	6.7	150	119	45	18	616	13.6	DS	Deep learning API
luigi	22.0	2 004	120	33	13	1 508	21.2	DEV	Pipelines of batch jobs management tool
matplotlib	99.6	5 526	147	30	0	2 484	34.9	DS	Plotting library
pandas	128.0	5 466	234	169	18	12 226	200.9	DS	Data analysis toolkit
PySnooper	0.7	60	7	3	0	49	3.9	DEV	Debugging tool
sanic	7.3	462	61	5	3	466	8.3	WEB	Web server and web framework
scrapy	15.7	1 509	179	40	0	1 572	24.5	WEB	Web crawling and web scraping framework
spaCy	97.2	852	415	10	6	986	13.4	DS	Natural language processing library
thefuck	4.7	604	203	32	16	614	7.3	CL	Console command tool
tornado	17.9	1 124	35	16	4	926	13.1	WEB	Web server
tqdm	3.3	200	28	9	7	120	2.7	CL	Progress bar for Python and CLI
youtube-dl	125.0	3 078	818	43	16	237	5.1	CL	Video downloader
total	714.0	25 588	2 978	501	135	24 817	482.1		

Table 1: Overview of projects in BUGSINPY. For each PROJECT, the table reports the project's overall size in KLOC (thousands of non-empty non-comment lines of code, excluding tests), the number |F| of functions (excluding test functions), the number |M| of modules (excluding test modules), the number of BUGS included in BUGSINPY, how many we selected as SUBJECTS for our experiments, the corresponding number of TESTS (i.e., test functions), their size in kLOC (TEST KLOC, thousands of non-empty non-comment lines of test code), the CATEGORY the project belongs to (CL: command line; DEV: development tools; DS: data science; WEB: web tools), and a brief DESCRIPTION of the project. Consistently with what done by the authors of BUGSINPY [67], the project statistics reported here refer to the *latest* version of the projects on 2020-06-19.

worth of experiments. In all, we used this selection of 135 bugs as our empirical study's subjects. Table 2 gives some details about the selected projects and their bugs.

As a side comment, note that our experiments with BUGSINPY were generally more time consuming than Zou et al.'s experiments with Defects4J. For example, the average per-bug running time of MBFL in our experiments (15 774 seconds in Table 6) was 3.3 times larger than in Zou et al.'s (4800 seconds in [78, Table 9]). Even more strikingly, running all fault localization experiments on the 357 Defects4J bugs took less than one CPU-month;[12] in contrast, running MBFL on just 27 "time consuming" bugs in BUGSINPY takes over 14 CPU-months. This difference may be partly due to the different characteristics of projects in Defects4J vs. BUGSINPY, and partly to the dynamic nature of Python (which is run by an interpreter).

4.2 Faulty Locations: Ground Truth

A fault localization technique's effectiveness measures how accurately the technique's list of suspicious entities matches the actual fault locations in a program—fault localization's *ground truth*. It is customary to use programmer-written patches as ground truth [78, 49]: the program locations modified by the patches that fix a certain bug correspond to the bug's actual fault locations.

Concretely, here is how to determine the ground truth of a bug $b = \langle p_b^-, p_b^+, F_b, P_b \rangle$ in BUGSINPY. The programmer-written fix p_b^+ consists of a series of *edits* to the faulty program p_b^-. Each edit can be of three kinds: *i*) *add*, which inserts into p_b^+ a new program location; *ii*) *remove*, which deletes a program location in p_b^-; *iii*) *modify*, which takes a program location in p_b^- and changes parts of it, without changing its location, in p_b^+. Take, for instance, the program in Figure 3b, which modifies the program in Figure 3a; the edited program includes two adds (lines 22, 31), one remove (line 35), and one modify (line 28).

Bug b's *ground truth* $\mathcal{F}(b)$ is a set of locations in p_b^- that are affected by the edits, determined as follows. First of all, ignore any blank or comment lines, since these do not affect a program's behavior and hence cannot be responsible for a fault. Then, finding the ground truth locations corresponding to removes and modifies is straightforward: a location ℓ that is removed or modified in p_b^+ exists by definition also in p_b^-, and hence it is part of the ground truth. In Figure 3, line 10 is modified and line 17 is removed by the edit that transforms Figure 3a into Figure 3b; thus 10 and 17 are part of the example's ground truth.

[12]The sum of column AVERAGE in [78, Table 9] multiplied by 357 gives 2.04 million seconds or 0.79 months.

CATEGORY	PROJECT	BUGS (SUBJECTS)		TESTS		GROUND TRUTH	
		C	*P*	*C*	*P*	*C*	*P*
CL	httpie	43	4	1188	217	139	12
	thefuck		16		614		55
	tqdm		7		120		22
	youtube-dl		16		237		50
DEV	black	30	13	1879	153	300	208
	cookiecutter		4		218		19
	luigi		13		1508		73
DS	keras	42	18	13828	616	186	111
	pandas		18		12226		64
	spaCy		6		986		11
WEB	fastapi	20	13	1987	595	174	156
	sanic		3		466		6
	tornado		4		926		12
	total	135	135	18882	18882	799	799

Table 2: Selected BUGSINPY bugs used in the paper's experiments. The PROJECTs are grouped by CATEGORY; the table reports—for each project individually (column *P*), as well as for all projects in the category (column *C*)—the number of BUGS selected as SUBJECTS for our experiments, the corresponding number of TESTS (i.e., test functions), and the total number of program locations that make up the GROUND TRUTH (described in Section 4.2).

Finding the ground truth locations corresponding to *adds* is more involved [57], because a location ℓ that is added to p_b^+ does not exist in p_b^-: b is a fault of omission [49].[13] A common solution [78, 49] is to take as ground truth the location in p_b^- that immediately *follows* ℓ. In Figure 3, line 6 corresponds to the first non-blank line that follows the assignment statement that is added at line 22 in Figure 3b; thus 6 is part of the example's ground truth. However, an add at ℓ is actually a modification between two other locations; therefore, the location that immediately *precedes* ℓ should also be part of the ground truth, since it identifies the same insertion location. In Figure 3, line 1 precedes the assignment statement that is added at line 22 in Figure 3a; thus 1 is also part of the example's ground truth.

A location's *scope* poses a final complication to determine the ground truth of adds. Consider line 31, added in Figure 3b at the very end of function `foo`'s body. The (non-blank, non-comment) location that follows it in Figure 3a is line 16; however, line 16 marks the beginning of another function `bar`'s definition. Function `bar` cannot be the location of a fault in `foo`, since the two functions are independent—in fact, the fact that `bar`'s declaration follows `foo`'s is immaterial. Therefore, we only include a location in the ground truth if it is within the same *scope* as the location ℓ that has been added. If ℓ is part of a function body (including methods), its scope is the function declaration; if ℓ is part of a class outside any function (e.g., an attribute), its scope is the class declaration; and otherwise ℓ's scope is the module it belongs to. In Figure 3, both lines 1 and 6 are within the same module as the added statement at line 22 in Figure 3a. In contrast, line 16 is within a different scope than the added statement at line 31 in Figure 3a. Therefore, lines 1, 6, and 12 are part of the ground truth, but not line 16.

Our definition of ground truth refines that used in related work [78, 49] by including the location that precedes an add, and by considering only locations within scope. We found that this definition better captures the programmer's intent and their corrective impact on a program's behavior.

How to best characterize bugs of omissions (fixed by an add) in fault localization remains an open issue [57]. Pearson et al.'s study [49] proposed the first viable solution: including the location following an add. Zou et al. [78] followed the same approach, and hence we also include the location following an add in our ground truth computation. We also noticed that, by also including the location preceding an add, and by taking scope into account, our ground truth computation becomes more comprehensive; in particular, it also works for statements added at the very end of a file—a location that has no following lines. While our approach is usually more precise, it is not necessarily the preferable alternative in all cases. Consider again, for instance, the add at line 31 in Figure 3; if we ignored the scope (and the preceding statement), only line 16 would be included in its ground truth. If this fault localization information were consumed by a developer, it could still be useful and actionable even if it reports a line outside the scope of the actual add location: the developer would use the location as a starting point for their inspection of the nearby code; and they may prefer a smaller, if slightly imprecise, ground truth to a larger, redundant one. However, this paper's focus is strictly evaluating the effectiveness of FL techniques as rigorously as possible—for which our stricter ground truth computation is more appropriate.

[13]In BUGSINPY, 41% of all fixes include at least one add edit.

```
a = 3




c = 5

# Function foo
def foo(y):
  if y > 3:
    a = y
  y = y * 2


# Function bar
def bar(z):
   z = z + 2
   return z
```

(a) Faulty program version. Lines with colored background are the ground truth locations. Extra blank lines are added for readability.

```
a = 3

# Global variable b
b = None          # add

c = 5

# Function foo
def foo(y):
  if y > 100:     # modify
    a = y
  y = y * 2
  a = y           # add

# Function bar
def bar(z):
  ~~z = z + 2~~       # remove
  return z
```

(b) Fixed program version, which edits Figure 3a's program with two adds, one modify, and one ~~remove~~.

Figure 3: An example of program edit, and the corresponding ground truth faulty locations.

	PROGRAM ENTITY ℓ									
	ℓ_1	ℓ_2	ℓ_3	ℓ_4	ℓ_5	ℓ_6	ℓ_7	ℓ_8	ℓ_9	ℓ_{10}
suspiciousness score s of ℓ	10	7	4	4	4	3	3	2	2	2
$\ell \in \mathcal{F}(b)$?		✗		✗				✗	✗	
start(ℓ)	1	2	3	3	3	6	6	8	8	8
ties(ℓ)	1	1	3	3	3	2	2	3	3	3
faulty(ℓ)	0	1	1	1	1	0	0	2	2	2
$\mathcal{I}_b(\ell, \langle \ell_1, s_1 \rangle \dots \langle \ell_n, s_n \rangle)$	1.0	2.0	4.0	4.0	4.0	6.0	6.0	8.3	8.3	8.3

Table 3: An example of calculating the E_{inspect} metric $\mathcal{I}_b(\ell, \langle \ell_1, s_1 \rangle \dots \langle \ell_n, s_n \rangle)$ for a list of 10 suspicious locations $\ell_1, \dots, \ell_{10}$ ordered by their decreasing suspiciousness scores $s_1, \dots, s_{10}$. For each location ℓ, the table reports its suspiciousness score s, and whether ℓ is a faulty location $\ell \in \mathcal{F}(b)$; based on this ranking of locations, it also shows the lowest rank start(ℓ) of the first location whose score is equal to ℓ's, the number ties(ℓ) of locations whose score is equal to ℓ's, the number of faulty locations among these, and the corresponding E_{inspect} value $\mathcal{I}_b(\ell, L)$—computed according to (6).

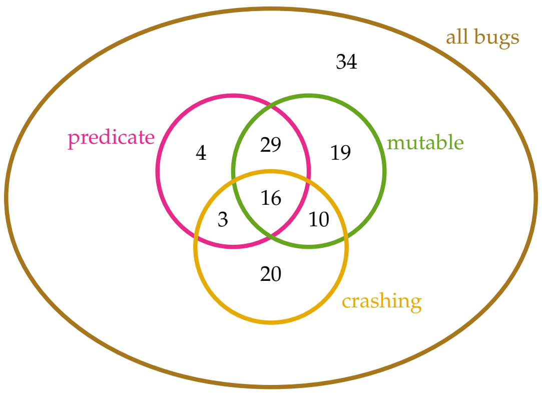


Figure 4: Classification of the 135 BugsInPy bugs used in our experiments into three categories.

4.3 Classification of Faults

Bug kind. The information used by each fault localization technique naturally captures the behavior of different *kinds* of faults. Stack trace fault localization analyzes the call stack after a program terminates with a crash; predicate switching targets branching conditions as program entities to perform fault localization; and MBFL crucially relies on the analysis of mutants to track suspicious locations.

Correspondingly, we classify a bug $b = \langle p_b^-, p_b^+, F_b, P_b \rangle$ as:

Crashing bug if any failing test in F_b terminates abruptly with an unexpected uncaught exception.

Predicate bug if any faulty entity in the ground truth $\mathcal{F}(b)$ includes a branching predicate (such as an **`if`** or **`while`** condition).

Mutable bug if any of the mutants generated by MBFL's mutation operators mutates any locations in the ground truth $\mathcal{F}(b)$. Precisely, a bug b's *mutability* is the percentage of all mutants of p_b^- that mutate locations in $\mathcal{F}(b)$; and b is mutable if its mutability is greater than zero.

The notion of crashing and predicate bugs is from Zou et al. [78].

We introduced the notion of mutable bug to try to capture scenarios where MBFL techniques have a fighting chance to correctly localize bugs. Since MBFL uses mutant analysis for fault localization, its capabilities depend on the mutation operators that are used to generate the mutants. Therefore, the notion of mutable bugs is somewhat dependent on the applied mutation operators.[14] Our implementation of FAUXPY uses the standard operators offered by the popular Python mutation testing framework Cosmic Ray [8]. As we discussed in Section 3, Cosmic Ray features a set of mutation operators that are largely similar to several other general-purpose mutation testing frameworks—all based on Offut et al.'s well known work [44]. These strong similarities between the mutation operators offered by most widely used mutation testing frameworks suggest that our definition of "mutable bug" is not strongly dependent on the specific mutation testing framework that is used. Correspondingly, bugs that we classify as "mutable" are likely to remain amenable to localization with MBFL provided one uses (at least) this standard set of core mutation operators. Conversely, we expect that devising new, specialized mutation operators may extend the number of bugs that we can classify as "mutable", and hence that are more likely to be amenable to localization with MBFL techniques.

Figure 4 shows the kind of the 135 BUGSINPY bugs we used in the experiments, consisting of 49 crashing bugs, 52 predicate bugs, 74 mutable bugs, and 34 bugs that do not belong to any of these categories.

Project category. Another, orthogonal classification of bugs is according to the project *category* they belong to. We classify a bug b as a CL, DEV, DS, or WEB bug according to the category of project (Table 2) b belongs to.

4.4 Ranking Program Entities

Running a fault localization technique L on a bug b returns a list of program entities $\ell_1, \ell_2, \ldots$, sorted by their decreasing suspiciousness scores $s_1 \geq s_2 \geq \ldots$. The programmer (or, more realistically, a tool [48, 16]) will go through the entities in this order until a faulty entity (that is an $\ell \in \mathcal{F}(b)$ that matches b's ground truth) is found. In this idealized process, the earlier a faulty entity appears in the list, the less time the programmer will spend going through the list, the more effective fault localization technique L is on bug b. Thus, a program entity's *rank* in the sorted list of suspicious entities is a key measure of fault localization effectiveness.

Computing a program entity ℓ's rank is trivial if there are no *ties* between scores. For example, consider Table 3's first two program entities ℓ_1 and ℓ_2, with suspiciousness scores $s_1 = 10$ and $s_2 = 7$. Obviously, ℓ_1's rank is 1 and ℓ_2's is 2; since ℓ_2 is faulty ($\ell_2 \in \mathcal{F}(b)$), its rank is also a measure of how many entities will need to be inspected in the aforementioned debugging process.

When several program entities tie the same suspiciousness score, their relative order in a ranking is immaterial [10]. Thus, it is a common practice to give all of them the same *average* rank [57, 62], capturing an average-case number of program entities inspected while going through the fault localization output list. For example, consider Table 3's first five program entities $\ell_1, \ldots, \ell_5$; ℓ_3, ℓ_4, and ℓ_5 all have the same suspiciousness score $s = 4$. Thus, they all have the same average rank $4 = (3 + 4 + 5)/3$, which is a proxy of how many entities will need to be inspected if ℓ_4 were faulty but ℓ_2 were not.

Capturing the "average number of inspected entities" is trickier still if more than one entity is faulty among a bunch of tied entities. Consider now all of Table 3's ten program entities; entities ℓ_8, ℓ_9, and ℓ_{10} all have the suspiciousness score $s = 2$; ℓ_8 and ℓ_9 are faulty, whereas ℓ_{10} is not. Their average rank $9 = (8 + 9 + 10)/3$ overestimates

[14]In this sense, "mutable" is a qualitatively different attribute than "crashing" and "predicate". Whether a bug b is "crashing" exclusively depends on the failing tests that trigger the bug; whether b is a "predicate" bug depends on the branching syntactic structure of b's program and how it relates to b. In contrast, whether b is a "mutable" bug depends on the mutation operators used to analyze b, and on whether they can change the program so as to effectively affect b's buggy behavior.

$$\mathcal{I}_b(L) = \min_{\ell \in L(b) \cap \mathcal{F}(b)} \mathcal{I}_b(\ell, L(b)) \qquad \widetilde{\mathcal{I}}_b(L) = \min_{\ell \in L^\infty(b) \cap \mathcal{F}(b)} \mathcal{I}_b(\ell, L^\infty(b)) \qquad \mathcal{E}_b(L) = \frac{\mathcal{I}_b(L)}{|p_b^-|} \tag{7}$$

$$L@_B n = \left|\{b \in B \mid \mathcal{I}_b(L) \leq n\}\right| \qquad \widetilde{\mathcal{I}}_B(L) = \frac{1}{|B|}\sum_{b \in B} \widetilde{\mathcal{I}}_b(L) \qquad \mathcal{E}_B(L) = \frac{1}{|B|}\sum_{b \in B} \mathcal{E}_b(L) \tag{8}$$

Figure 5: Definitions of common FL effectiveness metrics. The top row shows two variants $\mathcal{I}, \widetilde{\mathcal{I}}$ of the E_{inspect} metric, and the exam score $\mathcal{E}$, for a generic bug b and fault localization technique L. The bottom row shows cumulative metrics for a set B of bugs: the "at n" metric $L@_B n$, and the average $\widetilde{\mathcal{I}}$ and $\mathcal{E}$ metrics.

the number of entities to be inspected (assuming now that these are the only faulty entities in the output), since two entities out of three are faulty, and hence it is more likely that the faulty entity will appear before rank 9.

To properly account for such scenarios, Zou et al. [78] introduced the E_{inspect} metric, which ranks a program entity ℓ within a list $\langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle$ of program entities $\ell_1, \ldots, \ell_n$ with suspiciousness scores $s_1 \geq \ldots \geq s_n$ as:

$$\mathcal{I}_b(\ell, \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle) = \mathsf{start}(\ell) + \sum_{k=1}^{\mathsf{ties}(\ell) - \mathsf{faulty}(\ell)} k \, \frac{\binom{\mathsf{ties}(\ell) - k - 1}{\mathsf{faulty}(\ell) - 1}}{\binom{\mathsf{ties}(\ell)}{\mathsf{faulty}(\ell)}} \tag{6}$$

In (6), $\mathsf{start}(\ell)$ is the position k of the first entity among those with the same score as ℓ's; $\mathsf{ties}(\ell)$ is the number of entities (including ℓ itself) whose score is the same as ℓ's; and $\mathsf{faulty}(\ell)$ is the number of entities (including ℓ itself) that tie ℓ's score and are faulty (that is $\ell \in \mathcal{F}(b)$). Intuitively, the E_{inspect} rank $\mathcal{I}_b(\ell, \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle)$ is thus an average of all possible ranks where tied and faulty entities are shuffled randomly. When there are no ties, or only one entity among a group of ties is faulty, (6) coincides with the average rank.

Henceforth, we refer to a location's E_{inspect} rank $\mathcal{I}_b(\ell, \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle)$ as simply its *rank*.

Better vs. worse ranks. A clarification about terminology: a *high* rank is a rank that is close to the top-1 rank (the first rank), whereas a *low* rank is a rank that is further away from the top-1 rank. Correspondingly, a high rank corresponds to a small numerical ordinal value; and a low rank corresponds to a large numerical ordinal value. Consistently with this standard usage, the rest of the paper refers to "better" ranks to mean "higher" ranks (corresponding to smaller ordinals); and "worse" ranks to mean "lower" ranks (corresponding to larger ordinals).

4.5 Fault Localization Effectiveness Metrics

E_{inspect} effectiveness. Building on the notion of *rank*—defined in Section 4.4—we measure the *effectiveness* of a fault localization technique L on a bug b as the rank of the first faulty program entity in the list $L(b) = \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle$ of entities and suspiciousness scores returned by L running on b—defined as $\mathcal{I}_b(L)$ in (7). $\mathcal{I}_b(L)$ is L's E_{inspect} rank on bug b, which estimates the number of entities in L's one has to inspect to correctly localize b.

Generalized E_{inspect} effectiveness. What happens if a FL technique L cannot localize a bug b—that is, b's faulty entities $\mathcal{F}(b)$ do not appear at all in L's output? According to (6) and (7), $\mathcal{I}_b(L)$ is *undefined* in these cases. This is not ideal, as it fails to measure the effort wasted going through the location list when using L to localize b—the original intuition behind all rank metrics. Thus, we introduce a generalization L's E_{inspect} rank on bug b as follows. Given the list $L(b) = \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle$ of entities and suspiciousness scores returned by L running on b, let $L^\infty(b) = \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle \langle \ell_{n+1}, s_0 \rangle \langle \ell_{n+2}, s_0 \rangle \ldots$ be $L(b)$ followed by all *other entities* $\ell_{n+1}, \ell_{n+1}, \ldots$ in program p_b^- that are not returned by L, each given a suspiciousness $s_0 < s_n$ lower than any suspiciousness scores assigned by L.

With this definition, $\mathcal{I}_b(L) = \widetilde{\mathcal{I}}_b(L)$ whenever L can localize b—that is some entity from $\mathcal{F}(b)$ appears in L's output list. If some technique L_1 can localize b whereas another technique L_2 cannot, $\widetilde{\mathcal{I}}_b(L_2) > \widetilde{\mathcal{I}}_b(L_1)$, thus reflecting that L_2 is worse than L_1 on b. Finally, if neither L_1 nor L_2 can localize b, $\widetilde{\mathcal{I}}_b(L_2) > \widetilde{\mathcal{I}}_b(L_1)$ if L_2 returns a longer list than L_1: all else being equal, a technique that returns a shorter list is "better" than one that returns a longer list since it requires less of the user's time to inspect the output list. Accordingly, $\widetilde{\mathcal{I}}_b(L)$ denotes L's *generalized* E_{inspect} rank on bug b—defined as in (7).

Exam score effectiveness. Another commonly used effectiveness metric is the *exam score* $\mathcal{E}_b(L)$ [69], which is just a FL technique L's E_{inspect} rank on bug b over the number of program entities $|p_b^-|$ of the analyzed buggy program p_b^-—as in (7). Just like $\mathcal{I}_b(L)$, $\mathcal{E}_b(L)$ is undefined if L cannot localize b.

Effectiveness of a technique. To assess the overall effectiveness of a FL technique over a set B of bugs, we aggregate the previously introduced metrics in different ways—as in (8). The $L@_B n$ metric counts the number of bugs in B that L could localize within the top-n positions (according to their E_{inspect} rank); $n = 1, 3, 5, 10$ are common choices for n, reflecting a "feasible" number of entities to inspect. Then, the $L@_B n\% = 100 \cdot L@_B n / |B|$ metric is simply $L@_B n$ expressed as a percentage of the number $|B|$ of bugs in B. $\widetilde{\mathcal{I}}_B(L)$ is L's average generalized E_{inspect} rank of bugs in B. And $\mathcal{E}_B(L)$ is L's average exam score of bugs in B (thus ignoring bugs that L cannot localize).

Location list length. The $|L_b|$ metric is simply the number of suspicious locations output by FL technique L when run on bug b; and $|L_B|$ is the average of $|L_b|$ for all bugs in B. The location list length metric is not, strictly speaking, a measure of effectiveness; rather, it complements the information provided by other measures of effectiveness, as it gives an idea of how much output a technique produces to the user. All else being equal, a shorter location list length is preferable—provided it is not empty. In practice, we'll compare the location list length to other metrics of effectiveness, in order to better understand the trade-offs offered by each FL technique.

Different FL families use different kinds of information to compute suspiciousness scores; this is also reflected by the entities that may appear in their output location list. SBFL techniques include all locations executed by any tests T_b (passing or failing) even if their suspiciousness is zero; conversely, they omit all locations that are *not* executed by the tests. MBFL techniques include all locations executed by any *failing* tests F_b, since these locations are the targets of the mutation operators. PS includes all locations of *predicates* (branching conditions) that are executed by any failing tests F_b and that are *critical* (as defined in Section 2.3). ST includes all locations of all functions that appear in the stack trace of any crashing test in F_b.

Effectiveness metrics: limitations. Despite being commonly used in fault localization research, the effectiveness metrics presented in this section rely on assumptions that may not realistically capture the debugging work of developers. First, they assume that a developer can understand the characteristics of a bug and devise a suitable fix by examining just one buggy entity; in contrast, debugging often involves disparate activities, such as analyzing control and data dependencies and inspecting program states with different inputs [47]. Second, debugging is often not a *linear* sequence of activities [31] as simple as going through the ranked list of entities produced by fault localization techniques. Despite these limitations, we still rely on this section's effectiveness metrics: on the one hand, they are used in practically all related work on fault localization (in particular, Zou et al. [77]); thus, they make our results comparable to others. On the other hand, there are no viable, easy-to-measure alternative metrics that are also fully realistic; devising such metrics is outside this paper's scope and belongs to future work.

4.6 Comparison: Statistical Models

To quantitatively compare the capabilities of different fault localization techniques, we consider several standard statistics.

Pairwise comparisons. Let $M_b(L)$ be any metric M measuring the capabilities of fault-localization technique L on bug b; M can be any of Section 4.5's effectiveness metrics, or L's wall-clock running time $T_b(L)$ on bug b as performance metric. Similarly, for a fault-localization family F, $M_b(F)$ denotes the average value $\sum_{k \in F} M_b(k)/|F|$ of M_b for all techniques in family F. Given a set $B = \{b_1, \ldots, b_n\}$ of bugs, we compare the two vectors $M_B(F_1) = \langle M_{b_1}(F_1) \ldots M_{b_n}(F_1)\rangle$ and $M_B(F_2) = \langle M_{b_1}(F_2) \ldots M_{b_n}(F_2)\rangle$ using three statistics:

Correlation τ between $M_B(F_1)$ and $M_B(F_2)$ computed using Kendall's τ statistics. The absolute value $|\tau|$ of the correlation τ measures how closely changes in the value of metric M for F_1 over different bugs are *associated* to changes for F_2 over the same bugs: if $0 \le |\tau| \le 0.3$ the correlation is *negligible*; if $0.3 < |\tau| \le 0.5$ the correlation is *weak*; if $0.5 < |\tau| \le 0.7$ the correlation is *medium*; and if $0.7 < |\tau| \le 1$ the correlation is *strong*.

P-value p of a paired Wilcoxon signed-rank test—a nonparametric statistical test comparing $M_B(F_1)$ and $M_B(F_2)$. A small value of p is commonly taken as evidence against the "null-hypothesis" that the distributions underlying $M_B(F_1)$ and $M_B(F_2)$ have different medians:[15] usually, $p \le 0.05$, $p \le 0.01$, and $p \le 0.001$ are three conventional thresholds of increasing strength.

Cliff's δ effect size—a nonparametric measure of how often the values in $M_B(F_1)$ are larger than those in $M_B(F_2)$. The absolute value $|\delta|$ of the effect size δ measures how much the values of metric M differ, on the same bugs, between F_1 and F_2 [54]: if $0 \le |\delta| < 0.147$ the differences are *negligible*; if $0.145 \le |\delta| < 0.33$ the differences are *small*; if $0.33 \le |\delta| < 0.474$ the differences are *medium*; and if $0.474 \le |\delta| \le 1$ the differences are *large*.

[15] The practical usefulness of statistical hypothesis tests has been seriously questioned in recent years [66, 2, 14]; therefore, we mainly report this statistics for conformance with standard practices, but we refrain from giving it any serious weight as empirical evidence.

Regression models. To ferret out the individual impact of several different factors (fault localization family, project category, and bug kind) on the capabilities of fault localization, we introduce two varying effects regression models with normal likelihood and logarithmic link function.

$$\begin{bmatrix} E_b \\ T_b \end{bmatrix} \sim \mathsf{MVNormal}\left(\begin{bmatrix} e_b \\ t_b \end{bmatrix}, S\right) \qquad \log(e_b) = \alpha + \alpha_{\mathit{family}[b]} + \alpha_{\mathit{category}[b]} \qquad \log(t_b) = \beta + \beta_{\mathit{family}[b]} + \beta_{\mathit{category}[b]} \tag{9}$$

$$E_b \sim \mathsf{Normal}\,(e_b,\ \sigma) \qquad \log(e_b) = \begin{pmatrix} \alpha + \alpha_{\mathit{family}[b]} + \alpha_{\mathit{category}[b]} \\ \quad + c_{\mathit{family}[b]}\ \mathit{crashing}_b \\ \quad + p_{\mathit{family}[b]}\ \mathit{predicate}_b \\ \quad + m_{\mathit{family}[b]}\ \log(1 + \mathit{mutability}_b) \end{pmatrix} \tag{10}$$

Model (9) is multivariate, as it simultaneously captures effectiveness and runtime cost of fault localization. For each fault localization experiment on a bug b, (9) expresses the vector $[E_b, T_b]$ of standardized[16] E_{inspect} metric E_b and running time T_b as drawn from a multivariate normal distribution whose means e_b and t_b are log-linear functions of various *predictors*. Namely, $\log(e_b)$ is the sum of a base intercept α; a family-specific intercept $\alpha_{\mathit{family}[b]}$, for each fault-localization family SBFL, MBFL, PS, and ST; and a category-specific intercept $\alpha_{\mathit{category}[b]}$, for each project category CL, DEV, DS, and WEB. The other model component $\log(t_b)$ follows the same log-linear relation.

Model (10) is univariate, since it only captures the relation between bug kinds and effectiveness. For each fault localization experiment on a bug b, (10) expresses the standardized E_{inspect} metric E_b as drawn from a normal distribution whose mean e_b is a log-linear function of a base intercept α; a family-specific intercept $\alpha_{\mathit{family}[b]}$; and a category-specific intercept $\alpha_{\mathit{category}[b]}$; a varying intercept $c_{\mathit{family}[b]}\mathit{crashing}_b$, for the interactions between each family and crashing bugs; a varying intercept $p_{\mathit{family}[b]}\mathit{predicate}_b$, for the interactions between each family and predicate bugs; and a varying slope $m_{\mathit{family}[b]} \log(1 + \mathit{mutability}_b)$, for the interactions between each family and bugs with different mutability.[17] Variables *crashing* and *predicate* are indicator variables, which are equal to 1 respectively for crashing or predicate-related bugs, and 0 otherwise; variable *mutability* is instead the mutability percentage defined in Section 4.3.

After completing regression models (9) and (10) with suitable priors and fitting them on our experimental data[18] gives a (sampled) distribution of values for the coefficients α's, c, p, m, and β's, which we can analyze to infer the effects of the various predictors on the outcome. For example, if the 95% probability interval of α_F's distribution lies entirely below zero, it suggests that FL family F is consistently associated with below-average values of E_{inspect} metric $\mathcal{I}$; in other words, F tends to be more effective than techniques in other families. As another example, if the 95% probability interval of β_C's distribution includes zero, it suggests that bugs in projects of category C are not consistently associated with different-than-average running times; in other words, bugs in these projects do not seem either faster or slower to analyze than those in other projects.

4.7 Experimental Methodology

To answer Section 4's research questions, we ran FAUXPY using each of the 7 fault localization techniques described in Section 2 on all 135 selected bugs (described in Section 4.1) from BUGSINPY v. b4bfe91, for a total of $945 = 7 \times 135$ FL experiments. Henceforth, the term "*standalone* techniques" refers to the 7 classic FL techniques described in Section 2; whereas "*combined* techniques" refers to the four techniques introduced for RQ4.

Test selection. The test suites of projects such as keras (included in BUGSINPY) are very large and can take more than 24 hours to run even once. Without a suitable test selection strategy, large-scale FL experiments would be prohibitively time consuming (especially for MBFL techniques, which rerun the same test suite hundreds of times). Therefore, we applied a simple test selection strategy to only include tests that directly target the parts of a program that contribute to the failures.[19]

As we mentioned in Section 4.1, each bug b in BUGSINPY comes with a selection of failing tests F_b and passing tests P_b. The failing tests are usually just a few, and specifically trigger bug b. The passing tests, in contrast, are much more numerous, as they usually include all non-failing tests available in the project. In order to cull the number of passing tests to only include those that expressly target the failing code, we applied a simple dependency analysis: for each BUGSINPY bug b used in our experiments, we built the module-level call graph $G(b)$ for the whole of b's

[16]We standardize the data since this simplifies fitting the model; for the same reason, we also log-transform the running time in seconds.

[17]We log-transform *mutability* in this term, since this smooths out the big differences between mutability scores in different experiments (in particular, between zero and non-zero), which simplifies modeling the relation statistically. We add one to *mutability* before log-transforming it, so that the logarithm is always defined.

[18]The replication package includes all details about the regression models, as well as their validation [15].

[19]The Defects4J curated collection also includes a selection of so-called *relevant* tests [29].

project;[20] each node in $G(b)$ is a module of the project (including its tests), and each edge $x_m \to y_m$ means that module x_m directly uses some entities defined in module y_m. Consider any of b's project *test module* t_m; we run the tests in t_m in our experiments if and only if: *i)* t_m includes at least one of the *failing* tests in F_b; *ii) or*, $G(b)$ includes an edge $t_m \to f_m$, where f_m is a module that includes at least one of b's faulty locations $\mathcal{F}(b)$ (see Section 4.2). In other words: we include *all failing* tests for b, as well as the passing tests that directly exercise the parts of the project that are faulty. This simple heuristics substantially reduced the number of tests that we had to run for the largest projects, without meaningfully affecting the fault localization's scope.

Our test selection strategy does not include test modules that *indirectly* involve failing locations (unless they include any *failing* tests): if the tests in a module t_m only call directly an application module x_m, and then some parts of module x_m call another application module y_m (i.e., $t_m \to x_m \to y_m$ in the module-level call graph), x_m does not include any faulty locations, and y_m does include some faulty locations, then we do *not* include the tests in t_m in our test suite; instead, we will include *other* test modules u_m that directly call y_m (i.e., $u_m \to y_m$).

To demonstrate that our more aggressive test selection strategy does not exclude any relevant tests, and is unlikely to affect the quantitative fault localization results, we first computed, for each bug b used in our experiments: *i)* the set S_b^0 of tests selected using the strategy described above; and *ii)* the set $S_b^+ \supseteq S_b^0$ of tests selected by including also *indirect* dependencies (i.e., by taking the transitive closure of the module-level use relation). For 48% of the 135 bugs used in our experiments, $S_b^+ = S_b^0$, that is both test selection strategies select the same tests. However, there remain a long tail of bugs for which including indirect dependencies leads to many more tests being selected; for example, for 40 bugs in 7 projects, considering indirect dependencies leads to selecting more than 50 additional tests—which would significantly increase the experiments' running time. Thus, we randomly selected one bug for each project among those 40 bugs for which indirect dependencies would lead to including more than 50 additional tests. For each bug b in this sample, we performed an additional run of our fault localization experiments with SBFL and MBFL techniques[21] using all tests in S_b^+, for a total of 35 new experiments. We found that none of the key fault localization effectiveness metrics significantly changed compared to the same experiments using only tests in S_b^0.[22] This confirms that our test selection strategy does not alter the general effectiveness of fault localization, and hence we adopted it for the rest of the paper's experiments.

Table 4 shows statistics about the fraction of tests that we selected for our experiments according to the test selection strategy. Those data indicate that test selection has a disproportionate impact on projects that have very large test suites, such as those in the DS category. In these projects, it happens often that the vast majority of tests are irrelevant for the portion of the project where a failure occurred; therefore, excluding these tests from our experiments is instrumental in drastically bringing down execution times without sacrificing experimental accuracy.

Experimental setup. Each experiment ran on a node of USI's HPC cluster,[23] each equipped with 20-core Intel Xeon E5-2650 processor and 64 GB of DDR4 RAM, accessing a shared 15 TB RAID 10 SAS3 drive, and running CentOS 8.2.2004.x86_64. We provisioned three CPython Virtualenvs with Python v. 3.6, 3.7, and 3.8; our scripts chose a version according to the requirements of each BugsInPy subject. The experiments took more than two CPU-months to complete—not counting the additional time to setup the infrastructure, fix the execution scripts, and repeat any experiments that failed due to incorrect configuration.

This paper's detailed replication package includes all scripts used to ran these experiments, as well as all raw data that we collected by running them. The rest of this section details how we analyzed and summarized the data to answer the various research questions.[24]

4.7.1 RQ1. Effectiveness

To answer RQ1 (fault localization *effectiveness*), we report the $L@_B1\%$, $L@_B3\%$, $L@_B5\%$, and $L@_B10\%$ counts, the average generalized E_{inspect} rank $\widetilde{\mathcal{I}}_B(L)$, the average exam score $\mathcal{E}_B(L)$, and the average location list length $|L_B|$ for each technique L among Section 2's seven standalone fault localization *techniques*; as well as the same metrics averaged over each of the four fault localization *families*. These metrics measure the effectiveness of fault localization from different angles. We report these measures for *all* 135 BugsInPy bugs B selected for our experiments.

[20]To build the call graph we used Python static analysis framework Scalpel [34], which in turn relies on PyCG [56] for this task.

[21]Since PS and ST only use failing tests, their behavior does not change as S_b^0 always includes the same failing tests as S_b^+.

[22]Precisely, in 20 of these 35 experiments the E_{inspect} score did not change at all. As for the remaining experiments, the E_{inspect} score changed but only for bugs that were not effectively localized: the bugs localized in the top-1, top-3, top-5, and top-10 positions did not change, except for a single bug whose $\mathcal{I}_b(\text{Metallaxis})$ went from 13 to 9 when we added the extra tests.

[23]Managed by USI's Institute of Computational Science (`https://intranet.ics.usi.ch/HPC`).

[24]Research questions RQ1, RQ2, RQ3, RQ4, and RQ6 only consider *statement-level* granularity; in contrast, RQ5 considers all granularities (see Section 2.5).

CATEGORY	PROJECT	MIN				MEDIAN				MEAN				MAX			
		C		P		C		P		C		P		C		P	
		#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
CL	httpie	1	0.6	1	0.8	17.0	15.2	7.0	4.7	40.6	17.9	8.0	7.2	126	100.0	17	100.0
	thefuck			3	0.6			5.0	1.7			7.4	2.0			18	5.4
	tqdm			1	1.7			63.0	95.2			47.9	82.1			77	100.0
	youtube-dl			15	14.3			89.5	51.0			78.7	43.8			126	55.8
DEV	black	2	0.2	16	88.5	80.0	27.6	91.0	91.0	80.8	19.9	83.3	91.2	198	93.5	129	93.5
	cookiecutter			11	5.2			44.0	26.3			39.8	20.9			60	28.0
	luigi			2	0.2			91.0	11.3			90.8	11.6			198	33.2
DS	keras	1	0.0	18	3.0	67.5	3.2	58.0	10.5	112.8	2.1	76.6	13.9	1 036	51.6	288	51.6
	pandas			1	0.0			91.5	0.8			159.8	1.4			1 036	8.9
	spaCy			13	1.4			75.0	7.8			80.5	8.6			152	16.9
WEB	fastapi	1	0.3	1	0.3	32.0	4.5	5.0	1.5	139.6	25.7	37.6	8.8	787	76.7	282	49.9
	sanic			98	21.2			265.0	56.5			220.3	47.3			298	64.2
	tornado			32	3.4			411.5	40.5			410.5	41.9			787	76.7
	overall	1	0.0	1	0.0	53.0	8.9	53.0	8.9	86.6	4.6	86.6	4.6	1 036	100.0	1 036	100.0

Table 4: Tests used in the fault localization experiments with the bugs of Table 2. Following the procedure described in Section 4.7, we selected s_b tests out of the t_b BugsInPy tests for each bug b among the 135 bugs used in our experiments. For each PROJECT, the table reports the MINimum, MEDIAN, MEAN, and MAXimum percentage $100 \cdot s_b/t_b$ % of selected tests among bugs b in the project (columns P); similarly, columns # report the same statistics the MINimum, MEDIAN, MEAN, and MAXimum number of selected tests s_b among all bug b in the project. Finally, columns C report the same statistics among all bugs in projects of the same CATEGORY; and the bottom row reports the **overall** statistics among all 135 bugs.

To qualitatively summarize the effectiveness comparison between two FL techniques A and B, we consider their counts $A@1\% \leq A@3\% \leq A@5\% \leq A@10\%$ and $B@1\% \leq B@3\% \leq B@5\% \leq B@10\%$ and compare them pairwise: $A@k\%$ vs. $B@k\%$, for the each k among 1, 3, 5, 10. We say that:

$A \gg B$: *"A is much more effective than B"*, if $A@k\% > B@k\%$ for all ks, and $A@k\% - B@k\% \geq 10$ for at least three ks out of four;

$A > B$: *"A is more effective than B"*, if $A@k\% > B@k\%$ for all ks, and $A@k\% - B@k\% \geq 5$ for at least one k out of four;

$A \geq B$: *"A tends to be more effective than B"*, if $A@k\% \geq B@k\%$ for all ks, and $A@k\% > B@k\%$ for at least three ks out of four;

$A \simeq B$: *"A is about as effective as B"*, if none of $A \gg B$, $A > B$, $A \geq B$, $B \gg A$, $B > A$, and $B \geq A$ holds.

To *visually compare* the effectiveness of different FL families, we use *scatterplots*—one for each pair F_1, F_2 of families. The scatterplot comparing F_1 to F_2 displays one point at coordinates (x, y) for each bug b analyzed in our experiments. Coordinate $x = \widetilde{\mathcal{I}}_b(F_1)$, that is the average generalized E_{inspect} rank that techniques in family F_1 achieved on b; similarly, $y = \widetilde{\mathcal{I}}_b(F_2)$, that is the average generalized E_{inspect} rank that techniques in family F_2 achieved on b. Thus, points lying below the diagonal line $x = y$ (such that $x > y$) correspond to bugs for which family F_2 performed *better* (remember that a lower E_{inspect} score means more effective fault localization) than family F_1; the opposite holds for points lying above the diagonal line. The location of points in the scatterplot relative to the diagonal gives a clear idea of which family performed better in most cases.

To *analytically compare* the effectiveness of different FL families, we report the estimates and the 95% probability intervals of the coefficients α_F in the fitted regression model (9), for each FL family F. If the interval of values lies entirely below zero, it means that family F's effectiveness tends to be *better* than the other families on average; if it lies entirely above zero, it means that family F's effectiveness tends to be *worse* than the other families; and if it includes zero, it means that there is no consistent association (with above- or below-average effectiveness).

4.7.2 RQ2. Efficiency

To answer RQ2 (fault localization *efficiency*), we report the average wall-clock running time $T_B(L)$, for each technique L among Section 2's seven standalone fault localization *techniques*, on bugs in B; as well as the same metric averaged over each of the four fault localization *families*. This basic metric measures how long the various FL techniques take to perform their analysis. We report these measures for *all* 135 BugsInPy bugs B selected for our experiments.

To qualitatively summarize the efficiency comparison between two FL techniques A and B, we compare pairwise their average running times $T(A)$ and $T(B)$, and say that:

$A \gg B$: *"A is much more efficient than B"*, if $T(A) > 10 \cdot T(B)$;
$A > B$: *"A is more efficient than B"*, if $T(A) > 1.1 \cdot T(B)$;
$A \simeq B$: *"A is about as efficient as B"*, if none of $A \gg B$, $A > B$, $B \gg A$, and $B > A$ holds.

To *visually compare* the efficiency of different FL families, we use *scatterplots*—one for each pair F_1, F_2 of families. The scatterplot comparing F_1 to F_2 displays one point at coordinates (x, y) for each bug b analyzed in our experiments. Coordinate $x = T_b(F_1)$, that is the average running time of techniques in family F_1 on b; similarly, $y = T_b(F_2)$, that is the average running time of techniques in family F_2 on b. The interpretation of these scatterplots is as those considered for RQ1.

To *analytically compare* the efficiency of different FL families, we report the estimates and the 95% probability intervals of the coefficients β_F in the fitted regression model (9), for each FL family F. The interpretation of the regression coefficients' intervals is similar to those considered for RQ1: β_F's lies entirely above zero when F tends to be *slower* (less efficient) than other families; it lies entirely below zero when F tends to be *faster*; and it includes zero when there is no consistent association with above- or below-average efficiency.

4.7.3 RQ3. Kinds of Faults and Projects

To answer RQ3 (fault localization behavior for different *kinds of faults* and *projects*), we report the same effectiveness metrics considered in RQ1 ($F@_X1\%$, $F@_X3\%$, $F@_X5\%$, and $F@_X10\%$ percentages, average generalized E_{inspect} ranks $\widetilde{\mathcal{I}}_X(F)$, average exam scores $\mathcal{E}_X(F)$, and average location list length $|F_X|$), as well as the same efficiency metrics considered in RQ2 (average wall-clock running time $T_X(F)$) for each standalone fault localization family F and separately for *i*) bugs X of different *kinds*: crashing bugs, predicate bugs, and mutable bugs (see Figure 4); *ii*) bugs X from projects of different *category*: CL, DEV, DS, and WEB (see Section 4.3).

To *visually compare* the effectiveness and efficiency of fault localization families on bugs from projects of different *category*, we color the points in the scatterplots used to answer RQ1 and RQ2 according to the bug's project category.

To *analytically compare* the effectiveness of different FL families on bugs of different *kinds*, we report the estimates and the 95% probability intervals of the coefficients c_F, p_F, and m_F in the fitted regression model (10), for each FL family F. The interpretation of the regression coefficients' intervals is similar to those considered for RQ1 and RQ2: c_F, p_F, and m_F characterize the effectiveness of family F respectively on crashing, predicate, and mutable bugs, *relative* to the average effectiveness of the *same family* F on other kinds of bugs.

Finally, to understand whether bugs from projects of certain categories are intrinsically harder or easier to localize, we report the estimates and the 95% probability intervals of the coefficients α_C and β_C in the fitted regression model (9), for each project category C. The interpretation of these regression coefficients' intervals is like those considered for RQ1 and RQ2; for example if α_C's interval is entirely below zero, it means that bugs of projects in category C are easier to localize (higher effectiveness) than the average of bugs in any project. This sets a baseline useful to interpret the other data that answer RQ3.

4.7.4 RQ4. Combining Techniques

To answer RQ4 (the effectiveness of *combining* FL techniques), we consider two additional fault localization techniques: CombineFL and AvgFL—both combining the information collected by some of Section 2's standalone techniques from different families.

CombineFL was introduced by Zou et al. [78]; it uses a learning-to-rank model to learn how to combine lists of ranked locations given by different FL techniques. After fitting the model on labeled training data,[25] one can use it like any other fault localization technique as follows: *i*) Run any combination of techniques $L_1, \ldots, L_n$ on a bug b; *ii*) Feed the ranked location lists output by each technique into the fitted learning-to-rank model; *iii*) The model's output is a list $\ell_1, \ell_2, \ldots$ of locations, which is taken as the FL output of technique CombineFL. We used Zou et al. [78]'s replication package to run CombineFL on the Python bugs that we analyzed using FauxPy.

To see whether a simpler combination algorithm can still be effective, we introduced the combined FL technique AvgFL, which works as follows: *i*) Each basic technique L_k returns a list $\langle \ell_1^k, s_1^k \rangle \ldots \langle \ell_{n_k}^k, s_{n_k}^k \rangle$ of locations with *normalized*[26] suspiciousness scores $0 \leq s_j^k \leq 1$; *ii*) AvgFL assigns to location ℓ_x the weighted average $\sum_k w_k s_x^k$, where k ranges over all of FL techniques supported by FauxPy but Tarantula, and w_k is an integer weight that depends on the FL family of k: 3 for SBFL, 2 for MBFL, and 1 for PS and ST;[27] *iii*) The list of locations ranked by their weighted average suspiciousness is taken as the FL output of technique AvgFL.

[25]Since the training time is negligible, we ignore it in all measures of running time—consistently with Zou et al. [78].
[26]We used min-max normalization, also known as feature scaling [24].
[27]These weights roughly reflect the relative effectiveness and applicability of FL techniques suggested by our experimental results.

Finally, we answer RQ4 by reporting the same effectiveness metrics considered in RQ1 (the $L@_B1\%$, $L@_B3\%$, $L@_B5\%$, and $L@_B10\%$ counts, the average generalized E_{inspect} rank $\widetilde{\mathcal{I}}_B(L)$, the average exam score $\mathcal{E}_B(L)$, and the average location list length $|L_B|$) for techniques CombineFL and AvgFL. Precisely, we consider two variants A and S of CombineFL and of AvgFL, giving a total of four *combined* fault localization techniques: variants A (CombineFL$_A$ and AvgFL$_A$) use the output of all FL techniques supported by FauxPy but Tarantula—which was not considered in [78]; variants S (CombineFL$_S$ and AvgFL$_S$) only use the Ochiai, DStar, and ST FL techniques (excluding the more time-consuming MBFL and PS families).

4.7.5 RQ5. Granularity

To answer RQ5 (how fault localization effectiveness changes with granularity), we report the same effectiveness metrics considered in RQ1 (the $L@_B1$, $L@_B3$, $L@_B5$, and $L@_B10$ counts, the average generalized E_{inspect} rank $\widetilde{\mathcal{I}}_B(L)$, the average exam score $\mathcal{E}_B(L)$, and the average location list length $|L_B|$) for all seven standalone techniques, and for all four combined techniques, but targeting *functions* and *modules* as suspicious entities. Similar to Zou et al. [78], for function-level and module-level granularities, we define the suspiciousness score of an entity as the maximum suspiciousness score computed for the statements in them.

4.7.6 RQ6. Comparison to Java

To answer RQ6 (comparison between Python and Java), we quantitatively and qualitatively compare the main findings of Zou et al. [78]—whose empirical study of fault localization in Java was the basis for our Python replication—against our findings for Python.

For the *quantitative* comparison of *effectiveness*, we consider the metrics that are available in both studies: the percentage of all bugs each technique localized within the top-1, top-3, top-5, and top-10 positions of its output ($L@1\%$, $L@3\%$, $L@5\%$, and $L@10\%$); and the average exam score. For Python, we consider all 135 BugsInPy bugs we selected for our experiments; the data for Java is about Zou et al.'s experiments on 357 bugs in Defects4J [28]. We consider all standalone techniques that feature in both studies: Ochiai and DStar (SBFL), Metallaxis and Muse (MBFL), predicate switching (PS), and stack-trace fault localization (ST).

We also consider the combined techniques CombineFL$_A$ and CombineFL$_S$. The original idea of the CombineFL technique was introduced by Zou et al.; however, the variants used in their experiments combine all eleven FL techniques they consider, some of which we did not include in our replication (see Section 3 for details). Therefore, we modified [78]'s replication package to extract from their Java experimental data the rankings obtained by CombineFL$_A$ and CombineFL$_S$ combining the same techniques as in Python (see Section 4.7.4). This way, the quantitative comparison between Python and Java involves exactly the same techniques and combinations thereof.

Since we did not re-run Zou et al.'s experiments on the same machines used for our experiments, we cannot compare efficiency quantitatively. Anyway, a comparison of this kind between Java and Python would be outside the scope of our studies, since any difference would likely merely reflect the different performance of Java and Python—largely independent of fault localization efficiency.

For the *qualitative* comparison between Java and Python, we consider the union of all findings presented in this paper or in Zou et al. [78]; we discard all findings from one paper that are outside the scope of the other paper (for example, Java findings about history-based fault localization, a standalone technique that we did not implement for Python; or Python findings about AvgFL, a combined technique that Zou et al. did not implement for Java); for each within-scope finding, we determine whether it is confirmed ✔ (there is evidence corroborating it) or refuted ✖ (there is evidence against it) for Python and for Java.

5 Experimental Results

This section summarizes the experimental results that answer the research questions detailed in Section 4.7. All results except for Section 5.5's refer to experiments with statement-level granularity; results in Sections Section 5.1–5.3 only consider standalone techniques. To keep the discussion focused, we mostly comment on the @n% metrics of effectiveness, whereas we only touch upon the exam score, E_{inspect}, and location list length when they complement other results.

5.1 RQ1. Effectiveness

Family effectiveness. Among standalone techniques, the SBFL fault localization family achieves the best effectiveness according to several metrics. Table 5 shows that all SBFL techniques have better average E_{inspect} rank $\widetilde{\mathcal{I}}$; and

FAMILY	TECHNIQUE L	$\widetilde{\mathcal{I}}_B(L)$		$L@_B1\%$		$L@_B3\%$		$L@_B5\%$		$L@_B10\%$		$\mathcal{E}_B(L)$		$\lvert L_B \rvert$	
		F	T	F	T	F	T	F	T	F	T	F	T	F	T
MBFL	Metallaxis	6 710	6 706	8	10	22	25	27	30	34	37	0.0029	0.0035	113.9	113.9
	Muse		6 714		6		19		25		32		0.0023		113.9
PS		11 945	11 945	3	3	5	5	7	7	7	7	0.0001	0.0001	1.0	1.0
SBFL	DStar		1 583		11		30		42		54		0.0042		2 521.3
	Ochiai	1 584	1 583	12	12	30	30	43	43	54	54	0.0042	0.0042	2 521.3	2 521.3
	Tarantula		1 586		12		30		43		54		0.0042		2 521.3
ST		9 810	9 810	0	0	4	4	6	6	13	13	0.0024	0.0024	42.9	42.9

Table 5: Effectiveness of standalone fault localization techniques at the statement-level granularity on all 135 selected bugs B. Each row reports a TECHNIQUE L's average generalized E_{inspect} rank $\widetilde{\mathcal{I}}_B(L)$; the percentage of all bugs it localized within the top-1, top-3, top-5, and top-10 positions of its output ($L@_B1\%$, $L@_B3\%$, $L@_B5\%$, and $L@_B10\%$); its average exam score $\mathcal{E}_B(L)$; and its average suspicious locations length $|L_B|$. Columns F report the same metrics averaged for all techniques that belong to the same FAMILY. Highlighted numbers denote the best technique according to each metric.

higher percentages of faulty locations in the top-1, top-3, top-5, and top-10. The advantage over MBFL—the second most-effective family—is consistent and conspicuous. According to the same metrics, the MBFL fault localization family achieves clearly better effectiveness than PS and ST. Then, PS tends to do better than ST, but only according to some metrics: PS has better @1%, @3%, and @5%, and location list length, whereas ST has better E_{inspect} and @10%.

Finding 1.1: SBFL is the most effective standalone fault localization family.

Finding 1.2: Standalone fault localization families ordered by effectiveness: SBFL $>$ MBFL $\gg$ PS $\simeq$ ST, where $>$ means better, $\gg$ much better, and $\simeq$ about as good.

Contrary to these general trends, PS achieves the best (lowest) exam score and location list length of all families; and ST is second-best according to these metrics. As Section 5.3 will discuss in more detail, PS and ST are techniques with a narrower scope than SBFL and MBFL: they can perform very well on a subset of bugs, but they fail spectacularly on several others. They also tend to return shorter lists of suspicious locations, which is also conducive to achieving a better exam score: since the exam score is undefined when a technique fails to localize a bug at all (as explained in Section 4.5), the average exam score of ST and, especially, PS is computed over the small set of bugs on which they work fairly well.

Finding 1.3: PS and ST are specialized fault localization techniques, which may work well only on a small subset of bugs, and thus often return short lists of suspicious locations.

Figure 6's scatterplots confirm SBFL's general advantage: in each scatterplot involving SBFL, all points are on a straight line corresponding to low ranks for SBFL but increasingly high ranks for the other family. The plots also indicate that MBFL is often better than PS and ST, although there are a few hard bugs for which the latter are just as effective (points on the diagonal line). The PS-vs-ST scatterplot suggests that these two techniques are largely complementary: on several bugs, PS and ST are as effective (points on the diagonal); on several others, PS is more effective (points above the diagonal); and on others still, ST is more effective (points below the diagonal).

Figure 7a confirms these results based on the statistical model (9): the intervals of coefficients α_{SBFL} and α_{MBFL} are clearly below zero, indicating that SBFL and MBFL have better-than-average effectiveness; conversely, those of coefficients α_{PS} and α_{ST} are clearly above zero, indicating that PS and ST have worse-than-average effectiveness.

Figure 7a's estimate of α_{SBFL} is below that of α_{MBFL}, confirming that SBFL is the most effective family overall. The bottom-left plot in Figure 6 confirms that SBFL's advantage can be conspicuous but is observed only on a minority of bugs—whereas SBFL and MBFL achieve similar effectiveness on the majority of bugs. In fact, the effect size comparing SBFL and MBFL is -0.18—weakly in favor of SBFL.

Finding 1.4: SBFL and MBFL often achieve similar effectiveness; however, SBFL is strictly better than MBFL on a minority of bugs.

Technique effectiveness. FL techniques of the same family achieve very similar effectiveness. Table 5 shows nearly identical results for the 3 SBFL techniques Tarantula, Ochiai, and DStar. The plots and statistics in Figure 8 confirm this: points lie along the diagonal lines in the scatterplots, and E_{inspect} ranks for the same bugs are strongly correlated and differ by a vanishing effect size.

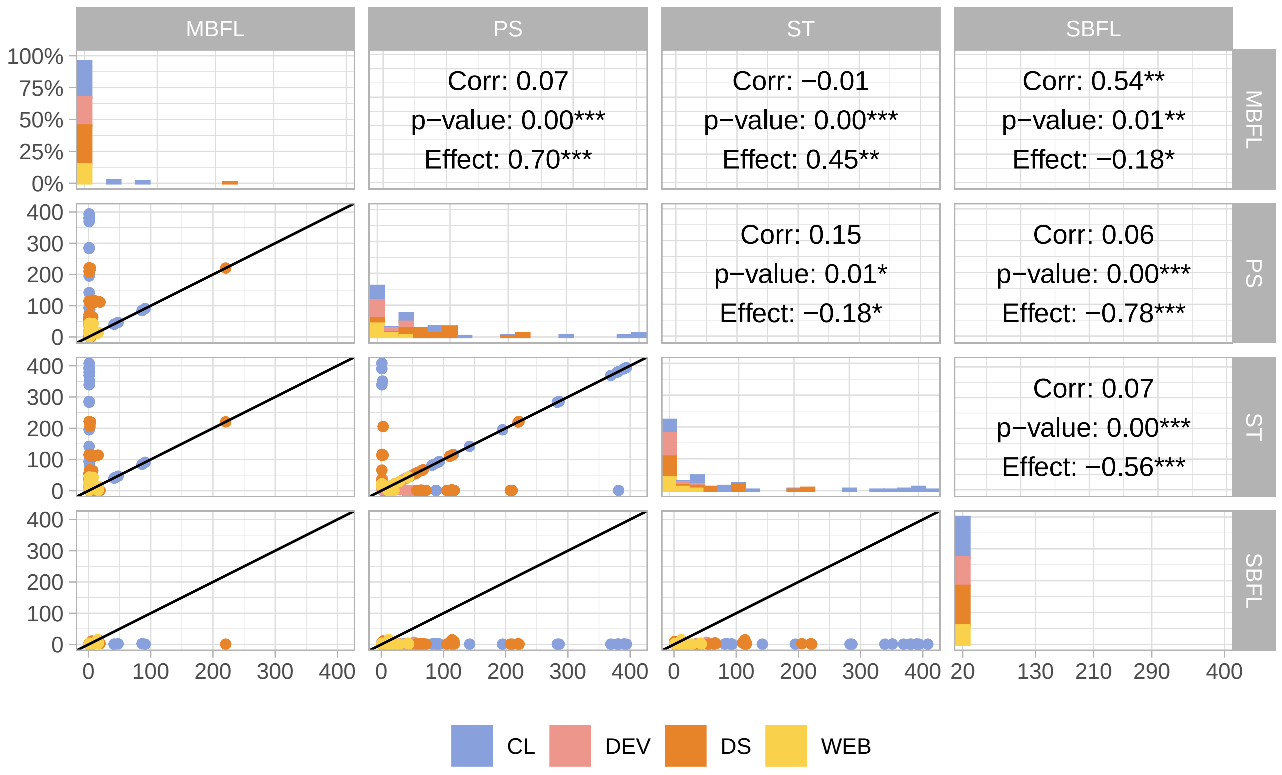


Figure 6: Pairwise visual comparison of four FL families for effectiveness. Each point in the scatterplot at row labeled R and column labeled C has coordinates (x, y), where x is the generalized E_{inspect} rank $\widetilde{\mathcal{I}}_b(C)$ of FL techniques in family C and y is the rank $\widetilde{\mathcal{I}}_b(R)$ of FL techniques in family R on the same bug b. Thus, points below (resp. above) the diagonal line denote bugs on which R had better (resp. worse) E_{inspect} ranks. Points are colored according to the bug's project category. The opposite box at row labeled C and column labeled R displays three statistics (correlation, p-value, and effect size, see Section 4.6) quantitatively comparing the same average generalized E_{inspect} ranks of C and R; negative values of effect size mean that R tends to be better, and positive values mean that C tends to be better. Each bar plot on the diagonal at row F, column F is a histogram of the distribution of $\widetilde{\mathcal{I}}_b(F)$ for all bugs. Horizontal axes of all diagonal plots have the same E_{inspect} scale as the bottom-right plot's (SBFL); their vertical axes have the same 0–100% scale as the top-left plot (MBFL).

Finding 1.5: All techniques in the SBFL family achieve very similar effectiveness.

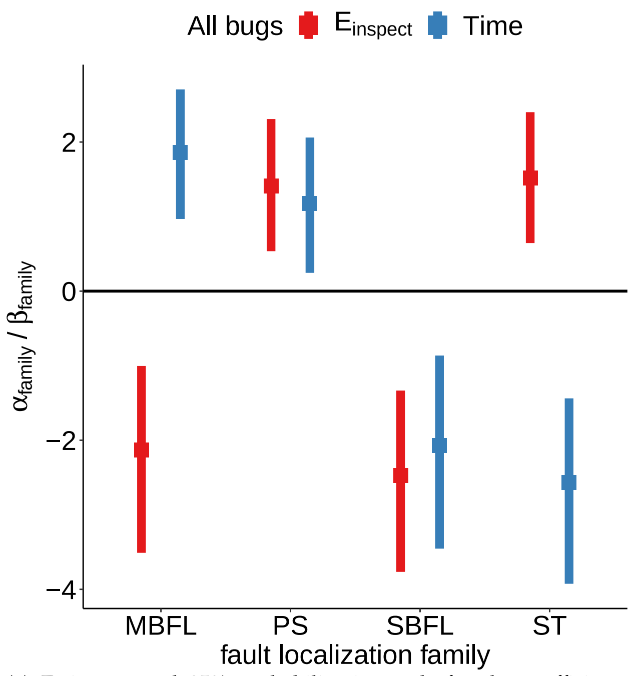
The 2 MBFL techniques also behave similarly, but not quite as closely as the SBFL ones. Metallaxis has a not huge but consistent advantage over Muse according to Table 5. Figure 9 corroborates this observation: the cloud of points in the scatterplot is centered slightly above the diagonal line; the correlation between Muse's and Metallaxis's data is medium (not strong); and the effect size suggests that Metallaxis is more effective on around 11% of subjects.

Muse's lower effectiveness can be traced back to its stricter definition of "mutant killing", which requires that a failing test becomes passing when run on a mutant (see Section 2.2). As observed elsewhere [49], this requirement may be too demanding for fault localization of real-world bugs, where it is essentially tantamount to generating a mutant that is similar to a patch.

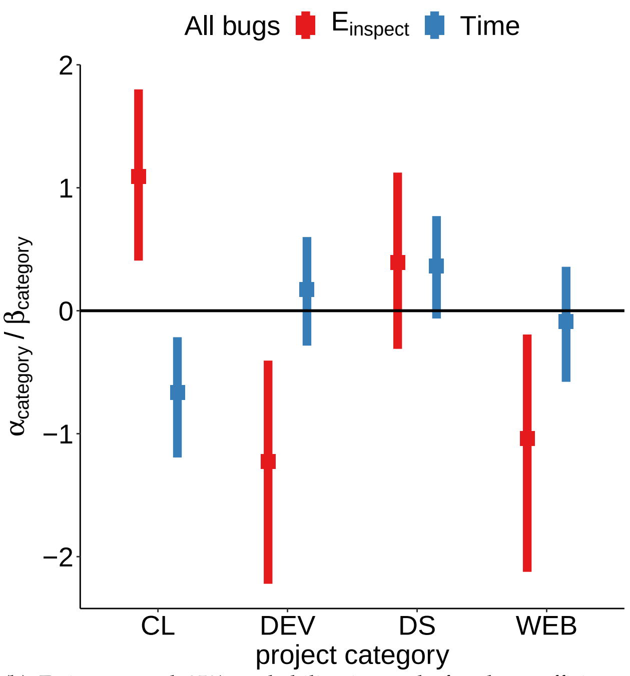
Finding 1.6: The techniques in the MBFL family achieve generally similar effectiveness, but Metallaxis tends to be better than Muse.

5.2 RQ2. Efficiency

As demonstrated in Table 6, the four FL families differ greatly in their efficiency—measured as their wall-clock running time. ST is by far the fastest, taking a mere 2 seconds per bug on average; SBFL is second-fastest, taking around 10 *minutes* on average; PS is one order of magnitude slower, taking approximately 2.7 *hours* on average; and MBFL is slower still, taking over 4 hours per bug on average.



(a) Estimates and 95% probability intervals for the coefficients α_{family} and β_{family} in model (9) fitted on all bugs, for each FL family MBFL, PS, SBFL, and ST.



(b) Estimates and 95% probability intervals for the coefficients $\alpha_{category}$ and $\beta_{category}$ in model (9) fitted on all bugs, for each project category CL, DEV, DS, and WEB.

Figure 7: Point estimates (boxes) and 95% probability intervals (lines) for the regression coefficients of model (9). The scale of the vertical axes is over standard deviation log-units.

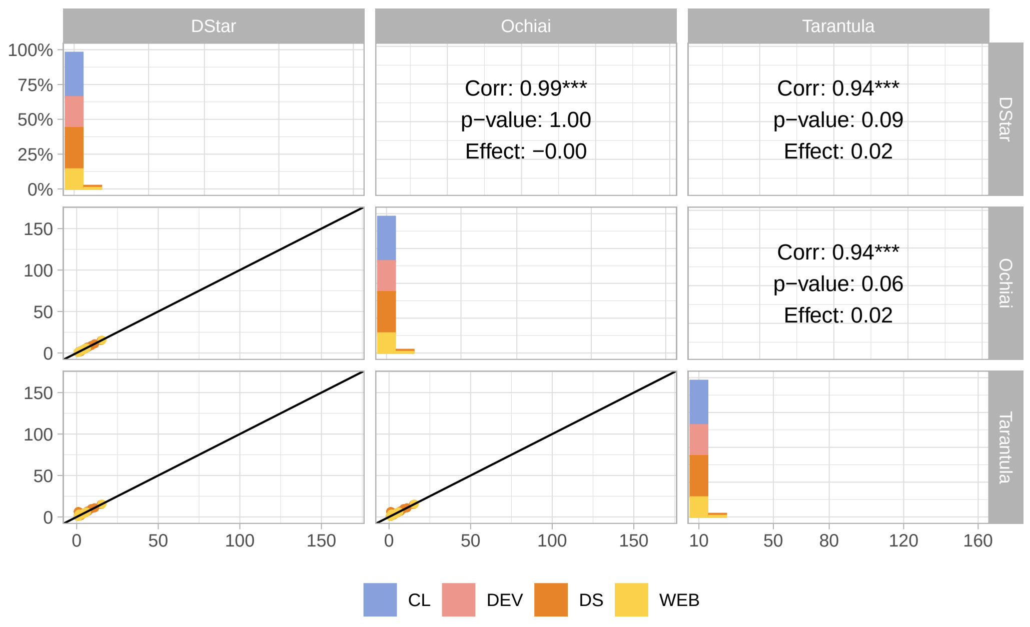


Figure 8: Pairwise visual comparison of 3 SBFL techniques for effectiveness. The interpretation of the plots is the same as in Figure 6.

Finding 2.1: Standalone fault localization families ordered by efficiency: ST $\gg$ SBFL $\gg$ PS $>$ MBFL, where $>$ means faster, and $\gg$ much faster.[a]

[a] As we discuss at the end of Section 5.2, these results are largely expected given how the different fault localization techniques work algorithmically.

Figure 10's scatterplots confirm that ST outperforms all other techniques, and that SBFL is generally second-fastest. It also shows that MBFL and PS have similar overall performance but can be slower or faster on different bugs: a narrow majority of points lies below the diagonal line in the scatterplot (meaning PS is faster than MBFL),

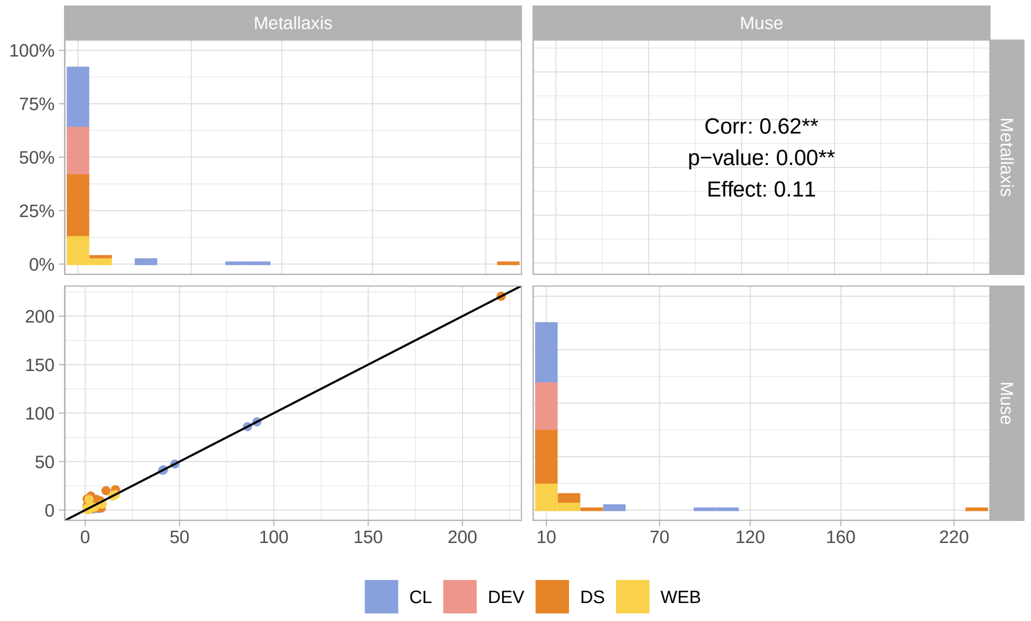


Figure 9: Pairwise visual comparison of 2 MBFL techniques for effectiveness. The interpretation of the plots is the same as in Figure 6.

FAMILY	TECHNIQUE L	ALL	CRASHING	PREDICATE	MUTABLE	CL	DEV	DS	WEB
MBFL	Metallaxis Muse	15 774	18 278	19 671	17 744	3 770	18 694	29 799	7 753
PS		9 751	11 419	17 287	12 932	528	20 210	15 972	828
SBFL	DStar Ochiai Tarantula	589	890	1 284	521	30	38	1 726	231
ST		2	2	2	2	2	1	2	1

Table 6: Efficiency of fault localization techniques at the statement-level granularity. Each row reports a TECHNIQUE L's per-bug average wall-clock running time $T_X(L)$ in seconds on: ALL 135 bugs selected for the experiments ($X = B$); CRASHING, PREDICATE-related, and MUTABLE bugs; bugs in projects of category CL, DEV, DS, and WEB (see Section 4.3). The running time is the same for all techniques of the same FAMILY. Highlighted numbers denote the fastest technique for bugs in each group.

but there are also several points that are on the opposite side of the diagonal—and their effect size (0.34) is medium, lower than all other pairwise effect sizes in the comparison of efficiency.

Finding 2.2: PS is more efficient than MBFL on average; however, the two families tend to be faster or slower on different bugs.

Based on the statistical model (9), Figure 7a clearly confirms the differences of efficiency: the intervals of coefficients β_{ST} and β_{SBFL} are well below zero, indicating that ST and SBFL are faster than average (with ST the fastest, as its estimated β_{ST} is lower); conversely, the intervals of coefficients β_{MBFL} and β_{PS} are entirely above zero, indicating that MBFL and PS stand out as slower than average compared to the other families.

These major differences in efficiency are unsurprising if one remembers that the various FL families differ in what kind of information they collect for localization. ST only needs the stack-trace information, which only requires to run once the failing tests; SBFL compares the traces of passing and failing runs, which involves running *all* tests once. PS dynamically tries out a large number of different branch changes in a program, each of which runs the failing tests; in our experiments, PS tried 4 588 different "switches" on average for each bug—up to a whopping 101 454 switches for project black's bug #6. MBFL generates hundreds of different mutations of the program under analysis, each of which has to be run against *all* tests; in our experiments, MBFL generated 461 mutants on average for each bug—up to 2 718 mutants for project black's bug #6. After collecting this information, the additional running time to compute suspiciousness scores (using the formulas presented in Section 2) is negligible for all techniques—which explains why the running times of techniques of the same family are practically indistinguishable.

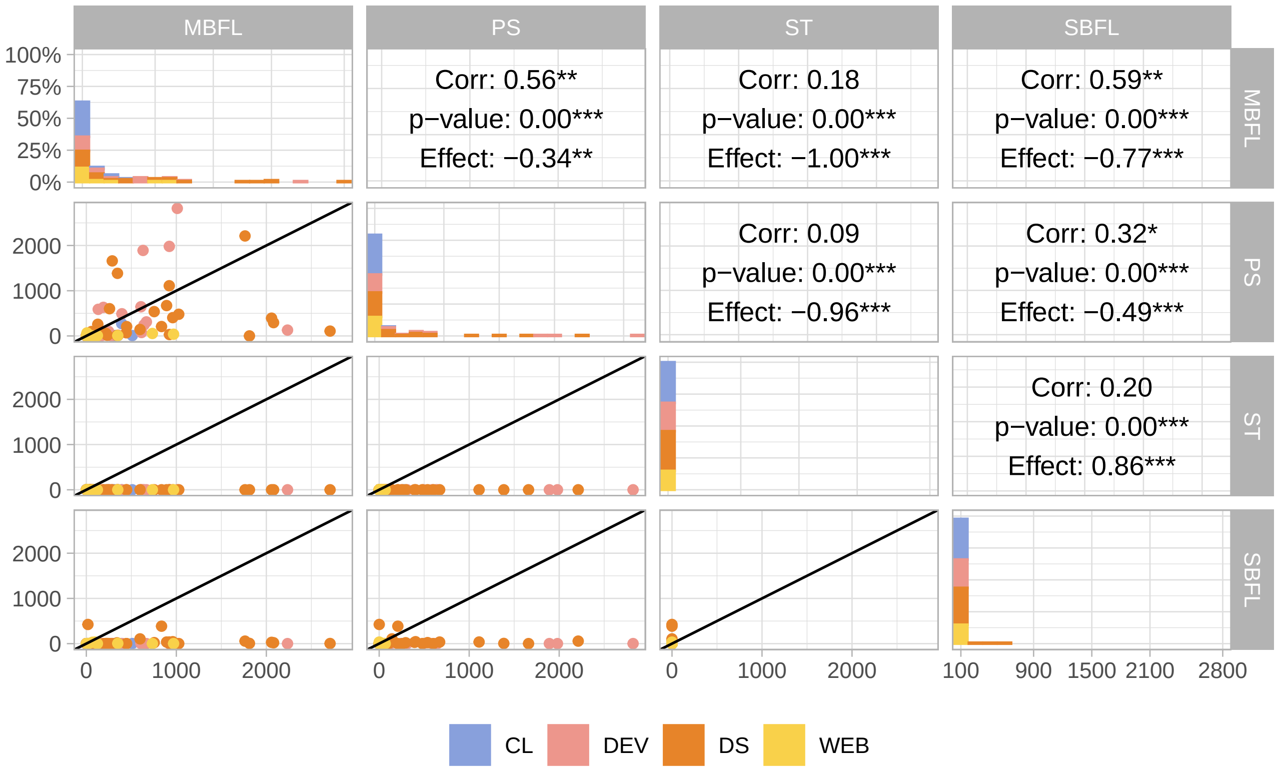


Figure 10: Pairwise visual comparison of four FL families for efficiency. Each point in the scatterplot at row labeled R and column labeled C has coordinates (x, y), where x is the average per-bug wall-clock running time of FL techniques in family C and y average per-bug wall-clock running time of FL techniques in family R. Points are colored according to the bug's project category. The opposite box at row labeled C and column labeled R displays three statistics (correlation, p-value, and effect size, see Section 4.6) quantitatively comparing the same per-bug average running times of C and R; negative values of effect size mean that R tends to be better, and positive values that C tends to be better.

5.3 RQ3. Kinds of Faults and Projects

Project category: effectiveness. Figure 7's intervals of coefficients $\alpha_{category}$ in model (9) indicate that fault localization tends to be more accurate on projects in categories DEV and WEB, and less accurate on projects in categories CL and DS.

This finding is consistent with the observations that data science programs, their bugs, and their fixes are often different compared to traditional programs [22, 23]. For instance, bug #38 in project keras is an example of what Islam et al. call "structural data flow" bugs [22]: its root cause is passing an incorrect input shape setting to a neural network layer. These characteristics also determine long spectra (i.e., execution traces) that span several functions—which are required to construct the various layer objects; as a result, SBFL techniques struggle to effectively localize this bug. Bugs #68 and #137 in project pandas are instead examples of API bugs, whose root causes are incorrect import statements. While such bugs may occur in any kind of project, they are common in data science programs [22] due to their complex dependencies. In Python, import statements are usually top-level declarations; therefore, FL techniques that can only target locations inside functions end up being ineffective at localizing these API bugs. As yet another example, the overall mutability of bugs in DS projects is 0.7%, whereas it is 1.3% for bugs in other categories of projects. This indicates that the standard mutation operators, used by MBFL, are a poor fit for the kinds of bugs that are most commonly found in data science projects.

Finding 3.1: Bugs in data science projects challenge fault localization's effectiveness (that is, they are harder to localize correctly) more than bugs in other categories of projects.

The data in Table 7's bottom section confirm that SBFL remains the most effective FL family, largely independent of the category of projects it analyzes. MBFL ranks second for effectiveness in every project category; it is not that far from SBFL for projects in categories DEV and CL (for example, MBFL and SBFL both localize 9% of CL bugs in the first position; and both localize over 40% of DEV bugs in the top-10 positions). In contrast, SBFL's advantage over MBFL is more conspicuous for projects in categories DS and WEB. Given that bugs in categories CL are generally

BUGS X	FAMILY F	$\widetilde{\mathcal{I}}_X(F)$	$F@_X1\%$	$F@_X3\%$	$F@_X5\%$	$F@_X10\%$	$\mathcal{E}_X(F)$	$\lvert F_X \rvert$
ALL	MBFL	6 710	8	22	27	34	0.0029	113.9
	PS	11 945	3	5	7	7	0.0001	1.0
	SBFL	1 584	12	30	43	54	0.0042	2 521.3
	ST	9 810	0	4	6	13	0.0024	42.9
CRASHING	MBFL	7 806	7	21	27	34	0.0018	104.4
	PS	15 607	0	0	0	0	–	0.3
	SBFL	897	14	31	43	53	0.0025	3 147.5
	ST	5 273	0	10	16	37	0.0024	118.1
PREDICATE	MBFL	1 891	11	33	40	52	0.0031	146.5
	PS	8 425	8	13	17	17	0.0001	1.3
	SBFL	374	12	23	38	50	0.0065	3 041.5
	ST	9 194	0	2	6	17	0.0007	47.2
MUTABLE	MBFL	489	14	41	50	63	0.0029	138.7
	PS	10 081	5	9	12	12	0.0001	1.1
	SBFL	524	12	35	50	57	0.0042	2 396.2
	ST	9 304	0	4	5	19	0.0007	35.3
CL	MBFL	2 910	9	33	38	45	0.0032	34.3
	PS	8 667	2	5	5	5	0.0002	0.3
	SBFL	2 356	9	42	60	74	0.0056	687.1
	ST	9 124	0	9	9	14	0.0084	19.9
DEV	MBFL	4 720	12	25	28	40	0.0045	160.6
	PS	7 768	3	7	10	10	0.0001	2.1
	SBFL	2 081	20	33	37	47	0.0053	1 431.5
	ST	8 279	0	0	10	13	0.0028	12.4
DS	MBFL	14 519	4	12	19	24	0.0006	169.3
	PS	22 847	2	5	7	7	0.0000	1.0
	SBFL	827	6	23	30	43	0.0018	5 775.4
	ST	15 174	0	0	0	12	0.0003	97.7
WEB	MBFL	1 465	8	18	20	25	0.0042	98.7
	PS	2 362	5	5	5	5	0.0002	1.1
	SBFL	770	15	15	40	45	0.0049	1 266.4
	ST	2 319	0	5	5	15	0.0014	22.7

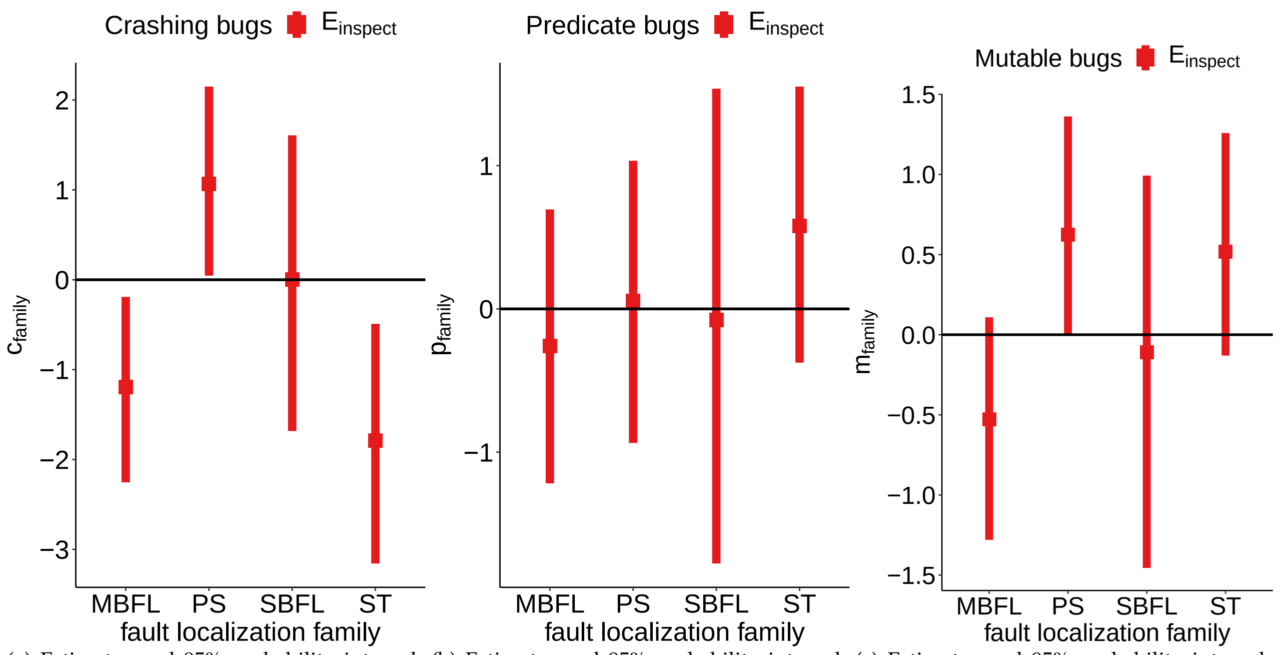
Table 7: Effectiveness of fault localization families at the statement-level granularity on different *kinds* of bugs and *categories* of projects. Each row reports a FAMILY F's average generalized E_{inspect} rank $\widetilde{\mathcal{I}}_X(F)$; the percentage of all bugs it localized within the top-1, top-3, top-5, and top-10 positions of its output ($F@_X1\%$, $F@_X3\%$, $F@_X5\%$, and $F@_X10\%$); its average exam score $\mathcal{E}_X(F)$ and the length $\lvert F_X \rvert$ of the output list of locations on different groups X of bugs: ALL bugs selected for the experiments (same results as in Table 5); bugs of different *kinds* (CRASHING, PREDICATE-related, and MUTABLE bugs); and bugs from projects of different *categories* (CL, DEV, DS, and WEB). Highlighted numbers denote the best family on each group of bugs according to each metric.

harder to localize, this suggests that the characteristics of bugs in these projects seem to be a good fit for MBFL. As we have seen in Section 5.2, MBFL is the slowest FL family by far; since it reruns the available tests hundreds, or even thousands, of times, projects with a large number of tests are near impossible to analyze efficiently with MBFL. As we'll discuss below, MBFL is considerably faster on projects in category CL than on projects in other categories; this is probably the main reason why MBFL is also more effective on these projects: it simply generates a more manageable number of mutants, which sharpen the dynamic analysis.

Finding 3.2: SBFL remains the most effective standalone fault localization family on all categories of projects.

Figure 6's plots confirm some of these trends. In most plots, we see that the points positioned far apart from the diagonal line correspond to projects in the CL and DS categories, confirming that these "harder" bugs exacerbate the different effectiveness of the various FL families.

Project category: efficiency. Figure 7's intervals of coefficients $\beta_{category}$ in model (9) indicate that fault localization tends to be more efficient (i.e., faster) on projects in category CL, and less efficient (i.e., slower) on projects in category DS (β_{DS} barely touches zero). In contrast, projects in categories DEV and WEB do not have a consistent association with faster or slower fault localization. Table 2 shows that projects in category DS have the largest number of tests by far (mostly because of outlier project pandas); furthermore, some of their tests involve training and testing different machine learning models, or other kinds of time-consuming tasks. Since FL invariably requires to run tests, this explains why bugs in DS projects tend to take longer to localize.



(a) Estimates and 95% probability intervals for the coefficients c_{family} in model (10), for each FL family MBFL, PS, SBFL, and ST.

(b) Estimates and 95% probability intervals for the coefficients p_{family} in model (10), for each FL family MBFL, PS, SBFL, and ST.

(c) Estimates and 95% probability intervals for the coefficients m_{family} in model (10), for each FL family MBFL, PS, SBFL, and ST.

Figure 11: Point estimates (boxes) and 95% probability intervals (lines) for the regression coefficients of model (10). The scale of the vertical axes is over standard deviation log-units.

Finding 3.3: Bugs in data science projects challenge fault localization's efficiency (that is, they take longer to localize) more than bugs in other categories of projects.

The data in Table 6's right-hand side generally confirm the same rankings of efficiency among FL families, largely regardless of what category of projects we consider: ST is by far the most efficient, followed by SBFL, and then—at a distance—PS and MBFL. The difference of performance between SBFL and ST is largest for projects in category DS (three orders of magnitude), large for projects in category WEB (two orders of magniture), and more moderate for projects in categories CL and DEV (one order of magnitude). PS is slower than MBFL only for projects in category DEV, although their absolute difference of running times is not very big (around 7.5%); in contrast, it is one order of magnitude faster for projects in categories CL and WEB.

Finding 3.4: The difference in efficiency between MBFL and SBFL is largest for data science projects.

In most of Figure 10's plots, we see that the points most frequently positioned far apart from the diagonal line correspond to projects in category DS, confirming that these bugs take longer to analyze and aggravate performance differences among techniques. In the scatterplot comparing MBFL to PS, points corresponding to projects in categories WEB and CL are mostly below the diagonal line, which corroborates the advantage of PS over MBFL for bugs of projects in these two categories.

Crashing bugs: effectiveness. According to Figure 11a, both FL families ST and MBFL are more effective on *crashing* bugs than on other kinds of bugs. Still, their *absolute* effectiveness on crashing bugs remains limited compared to SBFL's, as shown by the results in Table 7's middle part; for example, $@_{\text{CRASHING}}10\%$ is 37% for ST, 34% for MBFL, and 53% for SBFL, whereas ST localizes zero (crashing) bugs in the top rank. Remember that ST assigns that same suspiciousness to all statements within the same function (see Section 2.4); thus, it cannot be as accurate as SBFL even on the minority of crashing bugs.

Finding 3.5: ST and MBFL are more effective on crashing bugs than on other kinds of bugs
(but they remain overall less effective than SBFL even on crashing bugs).

On the other hand, PS is *less* effective on crashing bugs than on other kinds of bugs; in fact, it localizes zero bugs among the top-10 ranks. PS has a chance to work only if it can find a so-called *critical predicate* (see Section 2.3); only three of the crashing bugs included critical predicates, and hence PS was a bust.

Finding 3.6: PS is the least effective on crashing bugs.

Predicate-related bugs: effectiveness. Figure 11b says that no FL family achieves consistently better or worse effectiveness on predicate-related bugs. Table 7 complements this observation; the ranking of families by effectiveness is different for predicate-related bugs than it is for all bugs: MBFL is about as effective as SBFL, whereas PS is clearly more effective than ST.

Finding 3.7: On predicate-related bugs, MBFL is about as effective as SBFL, and PS is more effective than ST.

This outcome is somewhat unexpected for PS: predicate-related bugs are bugs whose ground truth includes at least a branching predicate (see Section 4.3), and yet PS is still clearly less effective than SBFL or MBFL. Indeed, the presence of a faulty predicate is not sufficient for PS to work: the predicate must also be *critical*, which means that flipping its value turns a failing test into a passing one. When a program has no critical predicates, PS simply returns an empty list of locations. In contrast, when a program has a critical predicate, PS is highly effective: PS@$_\chi$1% $=$ 14%, PS@$_\chi$3% $=$ 24%, and PS@$_\chi$5% $=$ 31% for PS on the 29 bugs χ with a critical predicate—even better than SBFL's results for the same bugs (SBFL@$_\chi$1% $=$ 13%, SBFL@$_\chi$3% $=$ 16%, and SBFL@$_\chi$5% $=$ 20%). In all, PS is a highly specialized FL technique, which works quite well for a narrow category of bugs, but is inapplicable in many other cases.

Finding 3.8: On the few bugs that it can analyze successfully, PS is the most effective standalone fault localization technique.

Mutable bugs: effectiveness. According to Figure 11c, FL family MBFL tends to be more effective on *mutable* bugs than on other kinds of bugs: m_{MBFL} 95% probability interval is mostly below zero (and the 87% probability interval would be entirely below zero). Furthermore, Table 7 shows that MBFL is the most effective technique on mutable bugs, where it tends to outperform even SBFL. Intuitively, a bug is mutable if the syntactic mutation operators used for MBFL "match" the fault in a way that it affects program behavior. Thus, the capabilities of MBFL ultimately depend on the nature of faults it analyzes and on the selection of mutation operators it employs.

Finding 3.9: MBFL is more effective on mutable bugs than on other kinds of bugs; in fact, it is the most effective standalone fault localization family on these bugs.

Figure 11c also suggests that PS and ST are less effective on mutable bugs than on other kinds of bugs. Possibly, this is because mutable bugs tend to be more complex, "semantic" bugs, whereas ST works well only for "simple" crashing bugs, and PS is highly specialized to work on a narrow group of bugs.

Finding 3.10: PS and ST are less effective on mutable bugs than on other kinds of bugs.

Bug kind: efficiency. Table 6 does not suggest any consistent changes in the efficiency of FL families when they work on crashing, predicate-related, or mutable bugs—as opposed to all bugs. In other words, for every kind of bugs: ST is orders of magnitude faster than SBFL, which is one order of magnitude faster than PS, which is 14–37% faster than MBFL. As discussed above, the kind of information that a FL technique collects is the main determinant of its overall efficiency; in contrast, different kinds of bugs do not seem to have any significant impact.

Finding 3.11: The relative efficiency of each fault localization family does not depend on the kinds of bugs that are analyzed.

5.4 RQ4. Combining Techniques

Effectiveness. Table 8 clearly indicates that the combined FL techniques AvgFL and CombineFL achieve high effectiveness—especially according to the fundamental @n% metrics. CombineFL$_A$ and AvgFL$_A$, combining the information from all other FL techniques, beat every other technique. For example, AvgFL$_A$ localizes in the top position 18% of all bugs, CombineFL$_A$ localizes 20% of all bugs, whereas the next-best technique is SBFL, which localizes 12% of all bugs (Table 5). CombineFL$_S$ and AvgFL$_S$, combining the information from only SBFL and ST techniques, do at least as well as every other standalone technique.

Finding 4.1: Combined fault localization techniques AvgFL$_A$ and CombineFL$_A$, which combine all baseline techniques, achieve better effectiveness than any other techniques.

While CombineFL$_A$ is strictly more effective than AvgFL$_A$, their difference is usually modest (at most three percentage points). Similarly, the difference between CombineFL$_S$, AvgFL$_S$, and SBFL is generally limited; however,

TECHNIQUE L		$\widetilde{\mathcal{I}}_B(L)$	$L@_B1\%$	$L@_B3\%$	$L@_B5\%$	$L@_B10\%$	$\mathcal{E}_B(L)$	$\lvert L_B\rvert$	$T_B(L)$
AvgFL	AvgFL$_A$	1 575	18	36	47	59	0.0033	2 548.4	26 116
	AvgFL$_S$	1 585	12	33	44	56	0.0040	2 548.4	591
CombineFL	CombineFL$_A$	1 580	20	39	49	60	0.0033	2 548.4	26 116
	CombineFL$_S$	1 584	12	32	41	56	0.0039	2 548.4	591

Table 8: Effectiveness and efficiency of fault localization techniques AvgFL and CombineFL at the statement-level granularity on all 135 selected bugs B. Each row reports a TECHNIQUE L's average generalized E_{inspect} rank $\widetilde{\mathcal{I}}_B(L)$; the percentage of all bugs it localized within the top-1, top-3, top-5, and top-10 positions of its output ($L@_B1\%$, $L@_B3\%$, $L@_B5\%$, and $L@_B10\%$); its average exam score $\mathcal{E}_B(L)$; its average suspicious locations length $\lvert L_B\rvert$; and its average per-bug wall-clock running time $T_B(L)$ in seconds. The four rows correspond to two variants AvgFL$_A$ and CombineFL$_A$ that combine the information of all FL techniques but Tarantula, and two variants AvgFL$_S$ and CombineFL$_S$ that combine the information of SBFL and ST techniques but Tarantula. Highlighted numbers denote the best technique according to each metric.

SBFL tends to be less effective than AvgFL$_S$, whereas CombineFL$_S$ is never strictly more effective than AvgFL$_S$. In all, AvgFL is a simpler approach to combining techniques than CombineFL, but both are quite successful at boosting FL effectiveness.

Finding 4.2: Fault localization families ordered by effectiveness:
CombineFL$_A$ $\geq$ AvgFL$_A$ $>$ CombineFL$_S$ $\simeq$ AvgFL$_S$ $>$ SBFL $>$ MBFL $\gg$ PS $\simeq$ ST,
where $>$ means better, $\geq$ better or as good, $\gg$ much better, and $\simeq$ about as good.

The suspicious location length is the very same for AvgFL and CombineFL, and higher than for every other technique. This is simply because all variants of AvgFL and CombineFL consider a location as suspicious if and only if any of the techniques they combine considers it so. Therefore, they end up with long location lists—at least as long as any combined technique's.

Efficiency. The running time of AvgFL and CombineFL is essentially just the sum of running times of the FL families they combine, because merging the output list of locations and training CombineFL's machine learning model take negligible time. This makes AvgFL$_A$ and CombineFL$_A$ the least efficient FL techniques in our experiments; and AvgFL$_S$ and CombineFL$_S$ barely slower than SBFL.

Finding 4.3: Combined fault localization techniques AvgFL$_A$ and CombineFL$_A$, which combine all baseline techniques, achieve worse efficiency than any other techniques.

Combining these results with those about effectiveness, we conclude that AvgFL$_A$ and CombineFL$_A$ exclusively favor effectiveness; whereas AvgFL$_S$ and CombineFL$_S$ promise a modest improvement in effectiveness in exchange for a modest performance loss.

Finding 4.4: Fault localization families ordered by efficiency:
ST $\gg$ SBFL $\geq$ AvgFL$_S$ $\simeq$ CombineFL$_S$ $\gg$ PS $>$ MBFL $>$ AvgFL$_A$ $\simeq$ CombineFL$_A$,
where $>$ means faster, $\geq$ faster or as fast, $\gg$ much faster, and $\simeq$ about as fast.

5.5 RQ5. Granularity

Function-level granularity. Table 9's data about function-level effectiveness of the various FL techniques and families lead to very similar high-level conclusions as for statement-level effectiveness: combination techniques CombineFL$_A$ and AvgFL$_A$ achieves the best effectiveness, followed by CombineFL$_S$ and AvgFL$_S$, then SBFL, and finally MBFL; differences among techniques in the same family are modest (often negligible).

ST is the only technique whose relative effectiveness changes considerably from statement-level to function-level: ST is the least effective at the level of statements, but becomes considerably better than PS at the level of functions. This change is no surprise, as ST is precisely geared towards localizing *functions* responsible for crashes—and cannot distinguish among statements belonging to the same function. ST's overall effectiveness remains limited, since the technique is simple and can only work on crashing bugs.

Module-level granularity. Table 10 leads to the same conclusions for module-level granularity: the relative effectiveness of the various techniques is very similar as for statement-level granularity, except that ST gains effectiveness simply because it is designed for coarser granularities.

FAMILY	TECHNIQUE L	$\widetilde{\mathcal{I}}_B(L)$		$L@_B1\%$		$L@_B3\%$		$L@_B5\%$		$L@_B10\%$		$\mathcal{E}_B(L)$		$\lvert L_B \rvert$	
		F	T	F	T	F	T	F	T	F	T	F	T	F	T
	AvgFL$_A$	66	66	53	53	71	71	77	76	84	84	0.0129	0.0130	296.3	296.3
	CombineFL$_A$		66		53		70		77		84		0.0128		296.3
	AvgFL$_S$	67	66	44	44	64	64	73	73	79	79	0.0153	0.0153	296.3	296.3
	CombineFL$_S$		67		44		64		73		79		0.0154		296.3
MBFL	Metallaxis	95	93	31	34	51	56	61	64	67	70	0.0150	0.0135	30.7	30.7
	Muse		97		27		46		57		64		0.0166		30.7
PS		618	618	8	8	13	13	13	13	15	15	0.0025	0.0025	0.6	0.6
	DStar	67	67	37	37	61	61	72	72	79	79	0.0156	0.0156	296.3	296.3
SBFL	Ochiai		67		38		61		72		79		0.0156		296.3
	Tarantula		67		36		61		71		78		0.0156		296.3
ST		451	451	21	21	27	27	27	27	29	29	0.0045	0.0045	1.0	1.0

Table 9: Effectiveness of fault localization techniques at the *function*-level granularity on all 135 selected bugs B. The table reports the same metrics as Table 5 and Table 8 but targeting functions as suspicious entities. Highlighted numbers denote the best technique according to each metric.

FAMILY	TECHNIQUE L	$\widetilde{\mathcal{I}}_B(L)$		$L@_B1\%$		$L@_B3\%$		$L@_B5\%$		$L@_B10\%$		$\mathcal{E}_B(L)$		$\lvert L_B \rvert$	
		F	T	F	T	F	T	F	T	F	T	F	T	F	T
	AvgFL$_A$	2	2	70	70	89	89	93	93	99	99	0.0339	0.0338	20.9	20.9
	CombineFL$_A$		2		70		89		93		99		0.0340		20.9
	AvgFL$_S$	2	2	64	64	87	87	93	93	98	98	0.0362	0.0363	20.9	20.9
	CombineFL$_S$		2		64		87		93		98		0.0362		20.9
MBFL	Metallaxis	6	6	52	57	80	82	86	87	90	92	0.0406	0.0366	5.6	5.6
	Muse		7		47		77		85		87		0.0446		5.6
PS		67	67	13	13	17	17	21	21	28	28	0.0234	0.0234	0.4	0.4
	DStar	2	2	60	61	86	87	92	93	98	98	0.0369	0.0365	20.9	20.9
SBFL	Ochiai		2		61		87		93		98		0.0365		20.9
	Tarantula		2		59		84		91		98		0.0375		20.9
ST		61	61	29	29	33	33	36	36	41	41	0.0284	0.0284	0.6	0.6

Table 10: Effectiveness of fault localization techniques at the *module*-level granularity on all 135 selected bugs B. The table reports the same metrics as Table 5 and Table 8 but targeting modules (files in Python) as suspicious entities. Highlighted numbers denote the best technique according to each metric.

Finding 5.1: ST is more effective than PS both at the function-level and module-level granularity; however, it remains considerably less effective than other fault localization techniques even at these coarser granularities.

Comparisons between granularities. It is apparent that fault localization's absolute effectiveness strictly *increases* as we target coarser granularities—from statements, to functions, to modules. This happens simply because the number of entities at a coarser granularity is considerably less than the number of entities at a finer granularity: each function consists of several statements, and each module consists of several functions. Therefore, it does not make sense to directly compare the same effectiveness metric measured at two different granularity levels, since each granularity level refers to different entities—and inspecting different entities involves incomparable effort.

We do not discuss efficiency (i.e., running time) in relation to granularity: the running time of our fault localization techniques does not depend on the chosen level of granularity, which only affects how the collected information is combined (see Section 2).

5.6 RQ6. Comparison to Java

Table 11 collects the main quantitative results for Python fault localization effectiveness that we presented in detail in previous parts of the paper, and displays them next to the corresponding results for Java. The results are selected so that they can be directly compared: they exclude any technique (e.g., Tarantula) or family (e.g., history-based

FAMILY	TECHNIQUE L	L@1%		L@3%		L@5%		L@10%		$\mathcal{E}(L)$	
		Python	*Java*	*Python*	*Java*	*Python*	*Java*	*Python*	*Java*	*Python*	*Java*
CombineFL	CombineFL$_A$	20	19	39	33	49	42	60	52	0.0033	0.0186
	CombineFL$_S$	12	10	32	23	41	30	56	40	0.0039	0.0265
MBFL	Metallaxis	10	6	25	22	30	29	37	36	0.0035	0.1180
	Muse	6	7	19	12	25	16	32	19	0.0023	0.3040
PS		3	1	5	4	7	6	7	6	0.0001	0.3310
SBFL	DStar	11	5	30	24	42	31	54	43	0.0042	0.0330
	Ochiai	12	4	30	23	43	31	54	44	0.0042	0.0330
ST		0	6	4	9	6	11	13	11	0.0024	0.3110

Table 11: Effectiveness of fault localization techniques in Python and Java. Each row reports a TECHNIQUE L's percentage of all bugs it localized within the top-1, top-3, top-5, and top-10 positions of its output (L@1%, L@3%, L@5%, and L@10%); and its average exam score $\mathcal{E}(L)$. Python's data corresponds to the experiments discussed in the rest of the paper on the 135 bugs from BUGSINPY; Java's data is taken from Zou et al.'s empirical study [78] or computed from its replication package. Highlighted numbers denote each language's best technique according to each metric.

fault localization) that was not experimented within both our paper and Zou et al. [78]; and the rows about CombineFL were computed using [78]'s replication package so that they combine exactly the same techniques (DStar, Ochiai, Metallaxis, Muse, PS, and ST for CombineFL$_A$; and DStar, Ochiai, and ST for CombineFL$_S$).

Then, Table 12 lists all claims about fault localization made in our paper or in [78] that are within the scope of both papers, and shows which were confirmed or refuted for Python and for Java. Most of the findings (25/28) were confirmed consistently for both Python and Java. Thus, the big picture about the effectiveness and efficiency of fault localization is the same for Python programs and bugs as it is for Java programs and bugs.

There are, however, a few interesting discrepancies; let's discuss possible explanations for them. The most marked difference is about the effectiveness of ST, which was mediocre on Python programs but competitive on Java programs (row 3 in Table 12). We think the main reason for these differences is that there were more Java experimental subjects that were an ideal target for ST: 20 out of the 357 Defects4J bugs used in [78]'s experiments consisted of short failing methods whose programmer-written fixes entirely replaced or removed the method body.[28] In these cases, the ground truth consists of all locations within the method; thus, ST easily ranks the fault location at the top by simply reporting all lines of the crashing method with the same suspiciousness. As a result, Table 11 shows that ST was consistently more effective than PS in the Java experiments—whereas there was no consistent difference between ST and PS in our Python experiments. For the same reason, the difference between Java and Python is even more evident on crashing bugs: ST outperformed all other techniques on such bugs in Java but not in Python (row 19 in Table 12). We still confirmed that ST works better on crashing bugs than on other kinds of bugs in Python as well, but the nature of our experimental subjects did not allow ST to reach an overall competitive effectiveness on crashing bugs.

Other findings about MBFL were different in Python compared to Java, but the differences were more nuanced in this case. In particular, Zou et al. found that the correlation between the effectiveness of SBFL and MBFL techniques is negligible, whereas we found a medium correlation ($\tau = 0.54$). It is plausible that the discrepancy (reflected in Table 12's row 23) is simply a result of several details of how this correlation was measured: we use Kendall's τ, they use the coefficient of determination r^2; we use a generalized E_{inspect} measure $\widetilde{\mathcal{I}}$ that applies to all bugs, they exclude experiments where a technique completely fails to localize the bug ($\mathcal{I}$); we compare the average effectiveness of SBFL vs. MBFL techniques, they pairwise compare individual SBFL and MBFL techniques. Even if the correlation patterns were actually different between Python and Java, this would still have limited practical consequences: MBFL and SBFL techniques still have clearly different characteristics, and hence they remain largely complementary. The same analysis applies to the other correlation discrepancy (reflected in Table 12's row 25): in Python, we found a medium correlation between the effectiveness of the Metallaxis and Muse MBFL techniques ($\tau = 0.62$); in Java, Zou et al. found negligible correlation.

Finally, a clarification about the finding that "*On predicate-related bugs, MBFL is about as effective as SBFL*", which Table 12 reports as confirmed for both Python and Java. This claim hinges on the definition of "about as effective", which we rigorously introduced in Section 4.7.1. To clarify the comparison, Table 13 displays the Python and Java data about the effectiveness of MBFL and SBFL on predicate bugs. On Python predicate-related bugs (left part of Table 13), MBFL achieves better @3%, @5%, and @10% than SBFL but a worse @1% (by only one percentage point); similarly, on Java predicate-related bugs (right part of Table 13), MBFL achieves better @1%, @3%, and @5% than

[28] For example, project Chart's bug #17 in Defects4J v1.0.1.

	FINDING	PYTHON		JAVA	
1	SBFL is the most effective standalone fault localization family.	✓	f 1.1	✓	[78, f 1.1]
2	Standalone fault localization families ordered by effectiveness: SBFL $>$ MBFL $\gg$ PS, ST	✓	f 1.2	✓	[78, T 3]
3	Regarding effectiveness, PS $\simeq$ ST.	✓	f 1.2	✗	T 11
4	All techniques in the SBFL family achieve very similar effectiveness.	✓	f 1.5	✓	[78, T 3]
5	The techniques in the MBFL family achieve generally similar effectiveness.	✓	f 1.6	✓	[78, T 3]
6	Metallaxis tends to be better than Muse.	✓	f 1.6	✓	[78, T 3]
7	Standalone fault localization families ordered by efficiency: ST $\gg$ SBFL $>$ PS $>$ MBFL	✓	f 2.1	✓	[78, f 4.2]
8	PS is more efficient than MBFL on average.	✓	f 2.2	✓	[78, T 9]
9	ST is more effective on crashing bugs than on other kinds of bugs.	✓	f 3.5	✓	[78, f 1.3]
10	MBFL is more effective on crashing bugs than on other kinds of bugs.	✓	f 3.5	✓	[78, T 3] , [78, T 4]
11	PS is the least effective on crashing bugs.	✓	f 3.6	✓	[78, T 4]
12	On predicate-related bugs, MBFL is about as effective as SBFL.	✓	T 13 , f 3.7	✓	T 13 , [78, T 5]
13	On predicate-related bugs, PS tends to be more effective than ST.	✓	f 3.7	✓	[78, T 5]
14	Combined fault localization technique $\mathsf{CombineFL}_A$, which combines all baseline techniques, achieves better effectiveness than any other techniques.	✓	f 4.1	✓	T 11
15	Fault localization families ordered by effectiveness: $\mathsf{CombineFL}_A$ $>$ $\mathsf{CombineFL}_S$ $>$ SBFL $>$ MBFL $\gg$ PS, ST	✓	f 4.2	✓	T 11
16	Combined fault localization technique $\mathsf{CombineFL}_A$, which combines all baseline techniques, achieves worse efficiency than any other technique.	✓	f 4.3	✓	[78, T 10]
17	Fault localization families ordered by efficiency: ST $\gg$ SBFL $\geq$ $\mathsf{CombineFL}_S$ $>$ PS $>$ MBFL $>$ $\mathsf{CombineFL}_A$	✓	f 4.4	✓	[78, T 10]
18	ST is more effective than PS at the function-level granularity; however, it remains considerably less effective than other fault localization techniques even at this coarser granularity.	✓	f 5.1	✓	[78, T 11]
19	ST is the most effective technique for crashing bugs.	✗	T 7	✓	[78, f 1.3]
20	PS is not the most effective technique for predicate-related faults.	✓	T 7	✓	[78, f 1.4]
21	Different correlation patterns exist between the effectiveness of different pairs of techniques.	✓	F 6 , F 8	✓	[78, f 2.1]
22	The effectiveness of most techniques from different families is weakly correlated.	✓	F 6	✓	[78, f 2.2]
23	The SBFL family's effectiveness has medium correlation with the MBFL family's.	✓	F 6	✗	[78, T 6]
24	The effectiveness of SBFL techniques is strongly correlated.	✓	F 8	✓	[78, T 6]
25	The effectiveness of MBFL techniques is weakly correlated.	✗	F 9	✓	[78, T 6]
26	Techniques with strongly correlated effectiveness only exist in the same family.	✓	F 6 , F 8 , F 9	✓	[78, f 2.3]
27	Not all techniques in the same family have strongly correlated effectiveness.	✓	F 8 , F 9	✓	[78, f 2.3]
28	The main findings about the relative effectiveness of fault localization families at statement-level granularity still hold at function-level granularity.	✓	T 9	✓	[78, f 5.1]

Table 12: A comparison of findings about fault localization in Python vs. Java. Each row lists a FINDING discussed in the present paper or in Zou et al. [78], whether the finding was confirmed ✓ or refuted ✗ for PYTHON and for JAVA, and the reported evidence that confirms or refutes it (a reference to a numbered finding, Figure, or Table in our paper or in [78]).

SBFL but a worse @10% (by three percentage points). In both cases, MBFL is not strictly better than SBFL, but one could argue that a clear tendency exists. Regardless of the definition of "more effective" (which can be arbitrary), the conclusion we can draw remain very similar in Python as in Java.

Finding 6.1: Our experiments confirmed for Python programs most of Zou et al. [78]'s findings about fault localization techniques on Java programs.

	PYTHON				JAVA			
FAMILY *F*	*F*@1%	*F*@3%	*F*@5%	*F*@10%	*F*@1%	*F*@3%	*F*@5%	*F*@10%
MBFL	11	33	40	52	9	21	29	34
SBFL	12	23	38	50	4	18	26	37

Table 13: A comparison of MBFL's and SBFL's effectiveness on Python and Java *predicate-related* bugs. The left part of the table reports a portion of the same data as Table 7: each column @*k*% reports the average percentage of the 52 predicate bugs in BugsInPy Python projects used in our experiments that techniques in the MBFL or SBFL family ranked within the top-*k*. The right part of the table averages some of the data in [78, Table 5] by family: each column @*k*% reports the average percentage of the 115 predicate bugs in Defects4J Java projects used in Zou et al.'s experiments that techniques in the MBFL or SBFL family ranked within the top-*k*. Highlighted numbers denote each language's best family according to each metric.

5.7 Threats to Validity

Construct validity refers to whether the experimental metrics adequately operationalize the quantities of interest. Since we generally used widely adopted and well-understood metrics of effectiveness and efficiency, threats of this kind are limited.

The metrics of effectiveness are all based on the assumption that users of a fault localization technique process its output list of program entities in the order in which the technique ranked them. This model has been criticized as unrealistic [48]; nevertheless, the metrics of effectiveness remain the standard for fault localization studies, and hence are at least adequate to compare the capabilities of different techniques and on different programs.

Using BugsInPy's curated collection of Python bugs helps reduce the risks involved with our selection of subjects; as we detail in Section 4.1, we did not blindly reuse BugsInPy's bugs but we first verified which bugs we could reliably reproduce on our machines.

Internal validity can be threatened by factors such as implementation bugs or inadequate statistics, which may jeopardize the reliability of our findings. We implemented the tool FauxPy to enable large-scale experimenting with Python fault localization; we applied the usual best practices of software development (testing, incremental development, refactoring to improve performance and design, and so on) to reduce the chance that it contains fundamental bugs that affect our overall experimental results. To make it a robust and scalable tool, FauxPy's implementation uses external libraries for tasks, such as coverage collection and mutant generation, for which high-quality open-source implementations are available.

The scripts that we used to process and summarize the experimental results may also include mistakes; we checked the scripts several times, and validated the consistency between different data representations.

We did our best to validate the test-selection process (described in Section 4.7), which was necessary to make feasible the experiments with the largest projects; in particular, we ran fault localization experiments on about 30 bugs without test selection, and checked that the results did not change after we applied test selection.

Our statistical analysis (Section 4.6) follows best practices [15], including validations and comparisons of the chosen statistical models (detailed in the replication package). To further help future replications and internal validity, we make available all our experimental artifacts and data in a detailed replication package.

External validity is about generalizability of our findings. Using bugs from real-world open-source projects substantially mitigates the threat that our findings do not apply to realistic scenarios. Precisely, we analyzed 135 bugs in 13 projects from the curated BugsInPy collection, which ensures a variety of bugs and project types.

As usual, we cannot make strong claims that our findings generalize to different application scenarios, or to different programming languages. Nevertheless, our study successfully confirmed a number of findings about fault localization in Java [78] (see Section 5.6), which further mitigates any major threats to external validity.

Zou et al.'s study used the Defects4J [28] curated collection of real-world Java faults as their experimental subjects; we used the BugsInPy [67] curated collection of real-world Python faults. This invariably limits the generalizability of our findings to *all* Python programs, and the generalizability of our comparison to all Python vs. Java programs: the two curated collections of bugs may not represent all programs and faults in Python or Java. While there is always a risk that any selection of experimental subjects is not fully representative of the whole population, choosing standard well-known benchmarks such as Defects4J and BugsInPy helps mitigate this threat. First, BugsInPy was explicitly inspired by Defects4J, and was built following a very similar approach but applied to real-world open-source Python programs. Second, BugsInPy projects were "selected as they represent the diverse domains [. . .] that Python is used for" [67, Sec. 1], which bodes well for generalizability. Third, BugsInPy and Defects4J are extensible frameworks, which have been and will be extended with new projects and bugs; thus, using them as the basis of FL studies helps to make future research in this area comparable to previous results. While BugsInPy and Defects4J are only imperfect proxies for a fully general comparison of FL in Java and Python, they are a sensible basis given the current state of the art.

6 Conclusions

This paper described an extensive empirical study of fault localization in Python, based on a differentiated conceptual replication of Zou et al.'s recent Java empirical study [78]. Besides replicating for Python several of their results for Java, we shed light on some nuances, and released detailed experimental data that can support further replications and analyses.

As a concluding discussion, let's highlight a few points relevant for possible follow-up work. Section 6.1 discusses a different angle for a comparison with other studies, suggested by Widyasari et al.'s recent work [68]. Section 6.2 describes broader ideas to improve the capabilities of fault localization in Python.

6.1 Other Fault Localization Studies

As we discussed in Section 3, Widyasari et al.'s recent work [68] is the only other large-scale study targeting fault localization in real-world Python projects. We also explained how our study's goals and methodology is quite different from theirs; as a result, we cannot directly compare most of their findings to ours. Now that we have presented our results in detail, we are in a better position to discuss how Widyasari et al.'s methodology suggests future work that complements our own.

Widyasari et al. directly compare FL effectiveness metrics (such as exam score) between their experiments on Python subjects from BugsInPy and Pearson et al.'s experiments on Java subjects from Defects4J [49]. Table 14a displays the key results of their comparison, alongside a roughly similar comparison between our experiments on Python subjects from BugsInPy and Zou et al.'s experiments on Java subjects from Defects4J [78]. The picture that emerges from these comparisons is somewhat inconclusive: in our comparison, there is a significant difference, with large effect size, between Python and Java with respect to exam scores, but not with respect to the E_{inspect} metric; conversely, in their comparison, there is a significant difference, with large/medium effect size, between Python and Java with respect to the top-k ranks in the best-case debugging scenarios (roughly analogous to the E_{inspect} ranking metric), whereas the differences with respect to exam scores are significant but with small effect sizes. Furthermore, the *sign* of the effect sizes is opposite: in our comparison, fault localization is more effective on Python programs (negative effect sizes); in their comparison, it is more effective on Java programs (positive effect sizes). It is plausible to surmise that these inconsistencies reflect differences between the effectiveness metrics, how they are measured in each study, and—most important—differences between the experimental subjects; the exam score metric, in particular, also depends on the size of the programs under analysis. As we discussed in Section 5.7, even though both benchmarks BugsInPy and Defects4J are carefully curated and of significant size, there is the risk that they do not necessarily represent *all* Python and Java real-world projects and their faults. This suggests that follow-up studies targeting different projects in Python and Java (or different selections of projects from BugsInPy and Defects4J) could help validate the generalizability of any results. Conversely, applying stricter project and bug selection criteria could also be useful not to generalize findings, but to strengthen their validity in more specific settings (for example, with projects of certain characteristics). Without provisioning stricter experimental controls, directly comparing, fault localization effectiveness metrics on sundry programs in two different programming languages, as we did in Table 14a for the sake of illustration, is unlikely to lead to clear-cut, robust findings.

Even though Widyasari et al.'s study found some statistically significant differences of effectiveness between SBFL techniques, those differences tend to be modest or insignificant. As shown in Table 14b, this is largely consistent with our findings: even though we found some weakly statistically significant differences between SBFL techniques (between DStar and Tarantula for $p < 0.1$, and between Ochiai and Tarantula for $p < 0.06$) these have little practical consequence as the effect sizes of the differences are vanishing small.

Our study did not consider two dimensions of analysis that play an important role in Widyasari et al.'s study: different debugging scenarios, and a classification of faults according to their syntactic characteristics. Debugging scenarios determine how we classify a fault as localized when it affects multiple lines. In our paper, we only considered the "best-case" scenario: as long as *any* of the ground-truth locations is localized, we consider the fault localized. Widyasari et al. also consider other scenarios such as the worst-case scenario (*all* ground-truth locations must be localized). While they did not find any significant differences in the various findings under different debugging scenarios, investigating the robustness of our empirical findings in different scenarios remains a viable direction for future work.

METRIC	TECHNIQUE L	THIS PAPER p	THIS PAPER EFFECT		[68] p	[68] EFFECT		REFERENCE
$\mathcal{E}(L)$	DStar	0.0000	-0.64	L	0.000000	0.32	S	[68, Tab. 5]
	Ochiai	0.0000	-0.64	L	0.000093	0.15	S	
$\mathcal{I}(L)$	DStar	0.0000	-0.27	S	0.000000	0.54	L	[68, Tab. 3]
	Ochiai	0.0000	-0.28	S	0.000000	0.41	M	

(a) Comparison of SBFL techniques on Python vs. Java programs. Each row compares the same SBFL TECHNIQUE L applied to PYTHON and to JAVA programs, reporting the p-value of a Wilcoxon rank-sum test, and Cliff's delta EFFECT size; a letter gives a qualitative assessment of the effect size: N for negligible, S for small, M for medium, and L for large. The data for THIS PAPER is each technique L's exam score $\mathcal{E}(L)$ and E_{inspect} rank $\mathcal{I}(L)$ for each bug among all 135 Python bugs used in the rest of the paper's experiments, and for each Java bug in Zou et al.'s replication package data [78]; to reflect the behavior on all bugs in these statistics, bugs that were not localized are assigned an $\mathcal{I}$ rank and an exam score of -1 (unlike the rest of the paper where this value is undefined). The statistics of [68] (in the four rightmost columns) are taken from its Table 5 (exam score, which they compute based on their top-k ranks) and Table 3 (best-case debugging scenario top-k ranks).

TECHNIQUE L_1	TECHNIQUE L_2	THIS PAPER p	THIS PAPER EFFECT		[68, Tab. 14] EFFECT	
DStar	Ochiai	0.584	0.00	N	0.14	N
DStar	Tarantula	0.093	-0.01	N	0.19	S
Ochiai	Tarantula	0.056	-0.01	N	0.04	N

(b) Pairwise comparison of SBFL techniques according to exam score. Each row compares the exam scores of two TECHNIQUEs L_1 and L_2 for significant differences, reporting the p-value of a Wilcoxon signed-rank test, and Cliff's delta EFFECT size; a letter gives a qualitative assessment of the effect size: N for negligible, S for small, M for medium, and L for large. The data for THIS PAPER is each technique L's exam score $\mathcal{E}(L)$ for each bug among all 135 Python bugs used in the rest of the paper's experiments; to reflect the behavior on all bugs in these statistics, bugs that were not localized are assigned an exam score of -1 (unlike the rest of the paper where this value is undefined). The statistics of [68] (in the two rightmost columns) are taken from its Table 14.

Table 14: A summary of some data presented in Widyasari et al.'s fault localization study [68] vis-à-vis analogous data presented in this paper.

6.2 Future Work

One of the dimensions of analysis that we included in our empirical study was the classification of projects (and their bugs) in categories, which led to the finding that faults in data science projects tend to be harder and take longer to localize. This is not a surprising finding if we consider the sheer size of some of these projects (and of their test suites). However, it also highlights an important category of projects that are much more popular in Python as opposed to more "traditional" languages like Java. In fact, a lot of the exploding popularity of Python in the last decade has been connected to its many usages for statistics, data analysis, and machine learning. Furthermore, there is growing evidence that these applications have distinctive characteristics—especially when it comes to faults [22, 19, 53]. Thus, investigating how fault localization can be made more effective for certain categories of projects is an interesting direction for related work (which we briefly discussed in Section 3).

It is remarkable that SBFL techniques, proposed nearly two decades ago [26], still remain formidable in terms of both effectiveness and efficiency. As we discussed in Section 3, MBFL was introduced expressly to overcome some limitations of SBFL. In our experiments (similarly to Java projects [78]) MBFL performed generally well but not always on par with SBFL; furthermore, MBFL is much more expensive to run than SBFL, which may put its practical applicability into question. Our empirical analysis of "mutable" bugs (Section 5.3) indicated that MBFL loses to SBFL usually when its mutation operators are not applicable to the faulty statements (which happened for nearly half of the bugs we used in our experiments); in these cases, the mutation analysis will not bring relevant information about the faulty parts of the program. These observations raise the question of whether it is possible to predict the effectiveness of MBFL based on preliminary information about a failure; and whether one can develop new mutation operators that extend the practical capabilities of MBFL to new kinds of bugs. More generally, one could try to relate the various kinds of source-code edits (add, remove, modify) [60] introduced to fix a fault to the effectiveness of different fault localization algorithms. We leave answering these questions to future research in this area.

Acknowledgements

Work partially supported by SNF grant 200021-182060 (Hi-Fi).

Data Availability

A replication package with data, analysis scripts, and other artifacts related to the research described in this paper are available: `https://doi.org/10.6084/m9.figshare.23254688`.

Declaration of Competing Interest

The authors declare that they have no competing interests that are related to the work described in this paper.

References

1. Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.
2. Valentin Amrehin, Sander Greenland, and Blake McShane. Scientists rise up against statistical significance. *Nature*, 567:305–307, 2019.
3. Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177—-188, 2016. doi: 10.1145/2931037.2931049.
4. Ned Batchelder. Coverage.py. `https://coverage.readthedocs.io/`, 2023. [Online; accessed 6-April-2023].
5. Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 308–318, 2008. doi: 10.1145/1453101.1453146.
6. Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. Dynamic slicing of Python programs. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 219–228, 2014. doi: 10.1109/COMPSAC.2014.30.
7. Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005. doi: 10.1145/1062455.1062522.
8. Cosmic Ray. Cosmic Ray: mutation testing for Python. `https://cosmic-ray.readthedocs.io/`, 2019. [Online; accessed 6-April-2023].
9. Wayne W. Daniel. *Biostatistics: A Foundation for Analysis in the Health Sciences*. Wiley, 7 edition, 1999.
10. Vidroha Debroy, W. Eric Wong, Xiaofeng Xu, and Byoungju Choi. A grouping-based strategy to improve the effectiveness of fault localization techniques. In *2010 10th International Conference on Quality Software*, pages 13–22, 2010. doi: 10.1109/QSIC.2010.80.
11. Anna Derezińska and Konrad Hałas. Analysis of mutation operators for the Python language. In Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak, and Janusz Kacprzyk, editors, *Proceedings of the 9th International Conference on Dependability and Complex Systems*, pages 155–164, 2014.
12. Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, 2005. doi: 10.1007/s10664-005-3861-2.
13. Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. DeepFault: Fault localization for deep neural networks. In Reiner Hähnle and Wil van der Aalst, editors, *Fundamental Approaches to Software Engineering*, pages 171–191. Springer International Publishing, 2019.
14. Carlo A. Furia, Robert Feldt, and Richard Torkar. Bayesian data analysis in empirical software engineering research. *IEEE Transactions on Software Engineering*, 47(9):1786–1810, September 2021.
15. Carlo A. Furia, Richard Torkar, and Robert Feldt. Applying Bayesian analysis guidelines to empirical software engineering data: The case of programming languages and code quality. *ACM Transactions on Software Engineering and Methodology*, 31(3):40:1–40:38, July 2022.
16. Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45:34–67, 2019. doi: 10.1109/TSE.2017.2755013.
17. Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. Audee: Automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 486–498, 2020. doi: 10.1145/3324884.3416571.

18. Clemens Hammacher. Design and implementation of an efficient dynamic slicer for Java. Bachelor's Thesis, November 2008.
19. Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1110–1121. ACM, 2020. doi: 10.1145/3377811.3380395. URL `https://doi.org/10.1145/3377811.3380395`.
20. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*, pages 191–200, 1994. doi: 10.1109/ICSE.1994.296778.
21. Qusay Idrees Sarhan, Attila Szatmári, Rajmond Tóth, and Árpád Beszédes. CharmFL: A fault localization tool for Python. In *IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 114–119, 2021. doi: 10.1109/SCAM52516.2021.00022.
22. Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 510–520, 2019. doi: 10.1145/3338906.3338955.
23. Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1135–1146, 2020. doi: 10.1145/3377811.3380378.
24. Md. Johirul Islam, Shamim Ahmad, Fahmida Haque, Mamun Bin Ibne Reaz, Mohammad Arif Sobhan Bhuiyan, and Md. Rezaul Islam. Application of min-max normalization on subject-invariant emg pattern recognition. *IEEE Transactions on Instrumentation and Measurement*, 71:1–12, 2022. doi: 10.1109/TIM.2022.3220286.
25. Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62.
26. James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005. doi: 10.1145/1101908.1101949.
27. René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 433–436, 2014. doi: 10.1145/2610384.2628053.
28. René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 437–440, 2014. doi: 10.1145/2610384.2628055.
29. René Just, Darioush Jalali, and other Defects4J contributors. Defects4J repository. `https://github.com/rjust/defects4j#export-version-specific-properties`, 2023.
30. Natalia Juristo Juzgado and Omar S. Gómez. Replication of software engineering experiments. In *Empirical Software Engineering and Verification – International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, volume 7007 of *Lecture Notes in Computer Science*, pages 60–88. Springer, 2010. doi: 10.1007/978-3-642-25231-0_2. URL `https://doi.org/10.1007/978-3-642-25231-0_2`.
31. Amy J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 301–310. ACM, 2008. doi: 10.1145/1368088.1368130. URL `https://doi.org/10.1145/1368088.1368130`.
32. Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, October 2011. URL `https://www.bodden.de/pubs/lblh11soot.pdf`.
33. Tien-Duy B. Le, Ferdian Thung, and David Lo. Theory and practice, do they match? a case with spectrum-based fault localization. In *IEEE International Conference on Software Maintenance*, pages 380–383, 2013. doi: 10.1109/ICSM.2013.52.
34. Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The Python static analysis framework. *arXiv preprint arXiv:2202.11840*, 2022.
35. Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.*, 1(OOPSLA):1–30, 2017. doi: 10.1145/3133916. Replication package: `https://github.com/deeprl4fl2021icse/deeprl4fl-2021-icse`.
36. Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180, 2019. doi: 10.1145/3293882.3330574.
37. Yi Li, Shaohua Wang, and Tien N. Nguyen. Fault localization with code coverage representation learning. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 661–673, 2021. doi: 10.1109/ICSE43902.2021.00067.
38. Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 664–676, 2021. doi: 10.1145/3468264.3468580.

39. Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005. doi: https://doi.org/10.1002/stvr.308.
40. David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891–1893, 2019. doi: 10.21105/joss.01891.
41. Steve McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004.
42. Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162, 2014. doi: 10.1109/ICST.2014.28.
43. Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 439–451, 2021. doi: 10.1145/3460319.3464797.
44. A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, apr 1996. ISSN 1049-331X. doi: 10.1145/227607.227610. URL `https://doi.org/10.1145/227607.227610`.
45. Mike Papadakis and Yves Le Traon. Metallaxis-fl: Mutation-based fault localization. *Softw. Test. Verif. Reliab.*, 25(5–7):605–628, 2015. doi: 10.1002/stvr.1509.
46. Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–378. 2019. doi: https://doi.org/10.1016/bs.adcom.2018.03.015.
47. Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209, 2011. doi: 10.1145/2001420.2001445.
48. Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In Matthew B. Dwyer and Frank Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 199–209. ACM, 2011. doi: 10.1145/2001420.2001445. URL `https://doi.org/10.1145/2001420.2001445`.
49. Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620, 2017. doi: 10.1109/ICSE.2017.62.
50. Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 322–331, 2011. doi: 10.1145/2025113.2025157.
51. M. Renieres and S.P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003. doi: 10.1109/ASE.2003.1240292.
52. Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 432–449, 1997. doi: 10.1145/267895.267925.
53. Mohammad Rezaalipour and Carlo A. Furia. An annotation-based approach for finding bugs in neural network programs. *Journal of Systems and Software*, 201:111669, July 2023.
54. J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In *Annual meeting of the Florida Association of Institutional Research*, 2006.
55. Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013. doi: 10.1109/ASE.2013.6693093.
56. Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. PyCG: Practical call graph generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 1646–1657, 2021. doi: 10.1109/ICSE43902.2021.00146.
57. Qusay Idrees Sarhan and Árpád Beszédes. A survey of challenges in spectrum-based software fault localization. *IEEE Access*, 10:10618–10639, 2022. doi: 10.1109/ACCESS.2022.3144079.
58. Eldon Schoop, Forrest Huang, and Bjoern Hartmann. Umlaut: Debugging deep learning programs using program structure and model behavior. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021. doi: 10.1145/3411764.3445538.
59. Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121, 2010. doi: 10.1109/MSR.2010.5463280.

60. Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 130–140. IEEE Computer Society, 2018. doi: 10.1109/SANER.2018.8330203. URL `https://doi.org/10.1109/SANER.2018.8330203`.
61. Jeongju Sohn and Shin Yoo. FLUCCS: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283, 2017. doi: 10.1145/3092703.3092717.
62. Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 314–324, 2013. doi: 10.1145/2483760.2483767.
63. Attila Szatmári, Qusay Idrees Sarhan, and Árpád Beszédes. Interactive fault localization for Python with CharmFL. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, pages 33–36, 2022. doi: 10.1145/3548659.3561312.
64. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot – a Java bytecode optimization framework. In Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999. URL `https://dl.acm.org/citation.cfm?id=782008`.
65. Mohammad Wardat, Wei Le, and Hridesh Rajan. Deeplocalize: Fault localization for deep neural networks. In *ICSE'21: The 43nd International Conference on Software Engineering*, 2021.
66. Ronald L. Wasserstein and Nicole A. Lazar. The ASA statement on *p*-values: Context, process, and purpose. *The American Statistician*, 70(2):129–133, 2016. `https://www.amstat.org/asa/files/pdfs/P-ValueStatement.pdf`.
67. Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1556–1560, 2020. doi: 10.1145/3368089.3417943.
68. Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus Agus Haryono, Shaowei Wang, and David Lo. Real world projects, real faults: Evaluating spectrum based fault localization techniques on Python projects. *Empirical Softw. Engg.*, 27(6), 2022. doi: 10.1007/s10664-022-10189-4.
69. Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *1st International Conference on Software Testing, Verification, and Validation*, pages 42–51, 2008. doi: 10.1109/ICST.2008.65.
70. W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014. doi: 10.1109/TR.2013.2285319.
71. W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016. doi: 10.1109/TSE.2016.2521368.
72. Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 191–200, 2014. doi: 10.1109/ICSME.2014.41.
73. Andreas Zeller. *Why Programs Fail – A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009. ISBN 978-0-12-374515-6. URL `http://store.elsevier.com/product.jsp?isbn=9780123745156&pagename=search`.
74. X. Zhang, R. Gupta, and N. Gupta. Locating faults through automated predicate switching. In *Software Engineering, International Conference on*, pages 272–281, 2006. doi: 10.1145/1134285.1134324.
75. Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. Autotrainer: An automatic dnn training problem detection and repair system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 359–371, 2021. doi: 10.1109/ICSE43902.2021.00043.
76. Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 826–837, 2020. doi: 10.1145/3368089.3409720.
77. Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012. doi: 10.1109/ICSE.2012.6227210.
78. Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2021. doi: 10.1109/TSE.2019.2892102.