Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

11-2022

Real world projects, real faults: Evaluating spectrum based fault localization techniques on Python projects

RATNADIRA WIDYASARI
Singapore Management University, ratnadiraw.2020@phdcs.smu.edu.sg

Gede Artha Azriadi PRANA
Singapore Management University, arthaprana.2016@phdis.smu.edu.sg

Stefanus AGUS HARYONO
Singapore Management University, stefanusah@smu.edu.sg

Shaowei WANG

David LO
Singapore Management University, davidlo@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Software Engineering Commons

Citation

RATNADIRA WIDYASARI; PRANA, Gede Artha Azriadi; AGUS HARYONO, Stefanus; WANG, Shaowei; and LO, David. Real world projects, real faults: Evaluating spectrum based fault localization techniques on Python projects. (2022). *Empirical Software Engineering*. 27, (6), 1-50.
Available at: https://ink.library.smu.edu.sg/sis_research/7321

This Journal Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Published in Empirical Software Engineering (2022) 27(6), 147
DOI: 10.1007/s10664-022-10189-4

Real world projects, real faults: evaluating spectrum based fault localization techniques on Python projects

Ratnadira Widyasari[1] · Gede Artha Azriadi Prana[1] · Stefanus Agus Haryono[1] · Shaowei Wang[2] · David Lo[1]

Abstract

Spectrum Based Fault Localization (SBFL) is a statistical approach to identify faulty code within a program given a *program spectra* (i.e., records of program elements executed by passing and failing test cases). Several SBFL techniques have been proposed over the years, but most evaluations of those techniques were done only on Java and C programs, and frequently involve artificial faults. Considering the current popularity of Python, indicated by the results of the Stack Overflow survey among developers in 2020, it becomes increasingly important to understand how SBFL techniques perform on Python projects. However, this remains an understudied topic. In this work, our objective is to analyze the effectiveness of popular SBFL techniques in real-world Python projects. We also aim to compare our observed performance on Python to previously-reported performance on Java. Using the recently-built bug benchmark BugsInPy as our fault dataset, we apply five popular SBFL techniques (Tarantula, Ochiai, O^P, Barinel, and DStar) and analyze their performances. We subsequently compare our results with results from Java and C projects reported in earlier related works. We find that 1) the real faults in BugsInPy are harder to identify using SBFL techniques compared to the real faults in Defects4J, indicated by the lower performance of the evaluated SBFL techniques on BugsInPy; 2) older techniques such as Tarantula, Barinel, and Ochiai consistently outperform newer techniques (i.e., O^P and DStar) in a variety of metrics and debugging scenarios; 3) claims in preceding studies done on artificial faults in C and Java (such as "O^P outperforms Tarantula") do not hold on Python real faults; 4) lower-performing techniques can outperform higher-performing techniques in some cases, emphasizing the potential benefit of combining SBFL techniques. Our results yield insight into how popular SBFL techniques perform in real Python faults and emphasize the importance of conducting SBFL evaluations on real faults.

Keywords Spectrum-based fault localization · Python · testing and debugging · empirical study

1 Introduction

Software debugging is an important but expensive part of software evolution (Planning 2002; Wright and Zia 2011). It becomes both more challenging and indispensable as modern software becomes increasingly complex and ubiquitous, with faults potentially incurring large economic cost[1] or even loss of human life (Wong et al. 2010). In the debugging process, developers often spend significant time and effort to discover the parts of source code responsible for a fault (Vessey 1985), i.e. *fault localization*. This has motivated the creation of various automated techniques and tools to aid fault localization over the past few decades. Automated fault localization techniques use a program's passing and failing test cases to narrow down the set of most suspicious locations in the program, which the developers can then manually inspect. By enabling developers to focus their efforts on a small portion of the program code, such techniques can save developers' time and increase the efficiency of the overall software development process (Xia et al. 2016). The automated fault localization technique especially the spectrum-based fault localization also usually use by automated program repair tools to identify the potential fault location. For example, Ochiai (Abreu et al. 2006) SBFL technique have been used in several automated program repair tool such as SimFix(Jiang et al. 2018), PraPR(Ghanbari et al. 2019), and CapGen(Wen et al. 2018).

Several types of approaches have been proposed for automated fault localization, such as spectrum-based (Abreu et al. 2006; Jones et al. 2001; Koca et al. 2013; Renieres and Reiss 2003; Wong et al. 2013), model-based (Abreu and Van Gemund 2009; Baah et al. 2010; Chaki et al. 2004; Könighofer and Bloem 2011), machine learning-based (Briand et al. 2007; Wong et al. 2011), and many others (Bouillon et al. 2007; Gouveia et al. 2013; Hao et al. 2009). In this work, we focus on Spectrum-Based Fault Localization (SBFL), which uses statistical formulas to measure the suspiciousness of program units based on the program's execution traces. Execution traces, also called program spectra, contain details on failing and passing test cases, along with some information regarding parts of the program executed by these test cases such as executed statements. The suspiciousness scores computed from program spectra are then used to generate a ranked list of program elements that are most likely to be responsible for the fault.

Spectrum-Based Fault Localization techniques have been gaining research attention in the past decade (Wong et al. 2016), and a number of studies have also evaluated the effectiveness of various techniques in this category (Lo et al. 2010; Renieres and Reiss 2003; Xie et al. 2013; Le et al. 2013). However, the issue regarding many existing evaluations stems from the common usage of artificial faults. Many existing studies use a dataset that comprises entirely or mostly of artificial faults. In view of this, (Pearson et al. 2017a) conducted a study that uses two sets of data related to 6 real-world projects (one set comprising artificial faults and another comprising real faults) to evaluate claims made by a number of earlier works. They found that a fault localization technique's performance on artificial faults is not a useful predictor of its performance on real faults.

Beyond common usage of artificial faults in existing studies, there is also a knowledge gap stemming from the evolving landscape of software development. As of the time of writing, evaluations on fault localization techniques' effectiveness are typically done on Java or C projects. However, other languages have gained popularity in recent years. As an example, the StackOverflow 2020 survey[2] ranks Python above Java and C in popularity, and

[1] http://www.abeacha.com/NIST_press_release_bugs_cost.htm
[2] https://insights.stackoverflow.com/survey/2020

Python is also used in a wide range of software including scientific computing libraries, web applications, and software engineering tools. However, to the best of our knowledge, there has been no study on the effectiveness of various fault localization techniques in Python programs.

The inconsistency between SBFL evaluation results from studies done on artificial faults versus that done on real-world Java faults, and the lack of evaluation of SBFL on currently popular languages beyond Java and C, poses an issue that needs to be addressed to provide researchers and practitioners alike with a more accurate view of the techniques' effectiveness. One way to address this is through replication studies, which can help increase accuracy and confidence in the original experiment's findings (Santos et al. 2020; Chen et al. 2019). Lindsay and Ehrenberg argue that in addition to validating the findings, replications are also needed to identify the range under which the findings hold as well as the exceptions (Lindsay and Ehrenberg 1993). In particular, an independent replication which only shares prior experiments' research objectives but varies one or more major aspects (Shull et al. 2008) will help in providing researchers and practitioners a more realistic assessment of the findings since such replications can demonstrate if "an effect is robust to changes with subjects, settings, and materials" (Kitchenham 2008).

In view of the above, we believe it is important to conduct an independent replication of the evaluation of SBFL techniques' effectiveness using real faults in real-world Python projects. We perform an evaluation on 5 popular SBFL techniques (Tarantula, Ochiai, O^P, Barinel, and DStar) (Wong et al. 2016; Pearson et al. 2017a) on a recent Python benchmark BugsInPy dataset by (Widyasari et al. 2020), containing 17 real-world Python projects that comprise of 493 real faults. We aim to answer the following research questions:

- **RQ-1: Does the effectiveness of SBFL techniques on Python projects (in BugsInPy) differ from that on Java projects (in Defects4J)?**

  We compare the performance of five popular SBFL techniques (Pearson et al. 2017a; Wong et al. 2016) on Defects4J (Just et al. 2014a) that were evaluated by (Pearson et al. 2017a) with the results on BugsInPy (Widyasari et al. 2020). We use *EXAM* score (Wong et al. 2008) and Top-k (Pearson et al. 2017a) to compare the performance of the techniques, and find that faults in BugsInPy are harder to identify using SBFL than faults in Defects4J, indicated by the lower performance result.
- **RQ-2**: **How effective are popular SBFL techniques on real faults of Python projects (in BugsInPy)?**

  To answer RQ-2, we run five popular SBFL techniques on BugsInPy dataset and evaluate the techniques' performance on several metrics including *EXAM* score (Wong et al. 2008), FLT rank (Pearson et al. 2017a), and improvement (Horváth et al. 2020). We subsequently conduct a statistical analysis on the result. For the *EXAM* score and FLT ranking, we find Tarantula to be the best-performing technique, but with a statistically insignificant difference from Barinel and Ochiai. We also find that the results of Top-K and improvement metrics are inline with the results on metrics *EXAM* score and FLT rank. Compared to the result of (Pearson et al. 2017a) where DStar, Ochiai, Barinel, and Tarantula do not have significant differences, our results show that DStar is significantly different with Ochiai, Barinel, and Tarantula.
- **RQ-3: Are the findings of previous studies applicable to Python projects (in BugsInPy)?**

  We examine the relative performance of different pairs of SBFL techniques and compare our result with claims of the preceding studies (such as "O^P outperforms Tarantula" and "Barinel outperforms Ochiai"). We use 7 finding statements from previous studies

that are checked by (Pearson et al. 2017a). We want to know whether these statements hold on Python real faults. Our findings are in line with findings of prior study by (Pearson et al. 2017a), where the performance differences between pairs of examined SBFL techniques on real faults do not match the findings of preceding studies done using artificial faults (Le et al. 2013; Abreu et al. 2009b; Le et al. 2015b; Naish et al. 2011a; Wong et al. 2016; Xuan and Monperrus 2014b; Moon et al. 2014; Ju et al. 2014).

In summary, our contributions are as follows:

1. **We investigate the effectiveness of 5 popular SBFL techniques on real faults in Python projects (in BugsInPy).** Our results yield insight into these techniques' effectiveness on real Python faults, which can aid both researchers and practitioners considering Python's popularity.
2. **We examine the generalizability of findings of previous studies on Python real-world fault dataset.** Our analysis of the findings, many of which are based on experiments on artificial faults, emphasizes the importance of conducting SBFL evaluations on real faults. Further, it can also motivate further research into the difference between the characteristics of artificial faults and that of real faults, which will be useful to aid the generation of better artificial faults.
3. **We examine how different the performance of SBFL techniques on Python real faults (in BugsInPy) compares to the Java real faults (in Defects4J).** Our results from this aspect of our work highlight the importance of more research into differences between characteristics of common faults in popular languages.
4. **We take into account real-world problems such as types of debugging scenarios and types of faults in the evaluation.** Debugging scenario types refer to situations where there can be more than one statement associated with a fault (i.e. multi-statement faults). We have three different debugging scenarios, which are worst-case, average-case, and best-case scenarios. Our results demonstrate that while rankings of the techniques are generally consistent across scenarios, even techniques with lower overall performance can outperform "better" techniques in some cases, indicating the need for additional research into the interaction between fault characteristics and performance of different SBFL techniques.

The rest of this paper is organized as follows: Section 2 summarizes previous works related to this study. Section 3 provides more details on the dataset we use as well as our empirical study methodology. Section 4 reports the result of our analyses. Section 5 provides discussion as well as implications of our results for practitioners and researchers. In Section 5 we also discusses the threats to our study's validity. Finally, Section 6 concludes this paper and presents future work.

2 Related Work

2.1 Spectrum-Based Fault Localization

Fault localization has been a hot research topic for several decades. Among the categories of approaches, Spectrum-Based Fault Localization (SBFL) has been one of the most popular and actively researched (Wong et al. 2016). Over the past decades, numerous variants of SBFL techniques have been proposed (Debroy et al. 2010; Renieres and Reiss 2003; Jones et al. 2001; Abreu et al. 2006; Naish et al. 2011b; Abreu et al. 2009b; Wong et al. 2013).

There has also been attempts to combine SBFL with other approaches, such as by (Ju et al. 2014).

Beyond proposed techniques, multiple studies have been conducted to investigate and compare the effectiveness of various SBFL techniques under different settings. For example, (Jones and Harrold 2005) evaluated the performance of Tarantula by comparing it against four other techniques. (Kim and Lee 2014) evaluated 32 SBFL techniques using Siemens Test Suite and a fault localization tool they developed (SKKU-FL). Another example is a study by (Le et al. 2013) that attempts to evaluate SBFL methods using a dataset comprising real world 199 faulty versions of Java and C projects, including 164 versions with artificial faults and 35 with real faults. A limitation of Le et al.'s work, as well as many other evaluations of SBFL techniques, is their usage of datasets that comprise mostly or entirely of artificial faults. This common limitation means evaluation of fault localization techniques' ability to find real faults is not sufficiently studied. In view of this, (Pearson et al. 2017a) conducted an evaluation of a range of fault localization techniques, including SBFL, using 310 real faults from Defects4J (Just et al. 2014b) in addition to 2,995 artificial faults generated using Major mutation framework tool (Just 2014). By comparing 7 SBFL technique pairs (e.g., Ochiai better than Tarantula, Barinel better than Ochiai, etc.) from previous studies, they found that, unlike results of prior evaluations that were done using artificial faults, the results from an evaluation using real faults differ.

Pearson et al. focus on real world projects written in Java and C. As the Python language becomes increasingly popular, there is a need to evaluate the effectiveness of SBFL on real faults of Python projects. To fill this gap, we select 5 popular SBFL techniques that have also been examined by (Pearson et al. 2017a) on real faults in Java projects and investigate their effectiveness on real faults in Python projects. We then examine whether the techniques' effectiveness of SBFL on Python projects differs from that on C/Java projects (see details in Section 3). Using the same Python dataset, we also investigate the validity of 7 comparisons SBFL techniques that were investigated by (Pearson et al. 2017a) on Java.

2.2 Faults Benchmark

There have been a number of attempts to build faults benchmark datasets. One of the earliest is the Siemens test suite (Hutchins et al. 1994), which contains 130 faulty versions of 7 C programs generated by manually seeding them with bugs. Other efforts include BugBench (Lu et al. 2005) that contains faults from 17 open-source C and C++ projects, BegBunch (Cifuentes et al. 2009) that comprises "Accuracy" and "Scalability" suite of faults obtained from several open-source C and C++ projects, ManyBugs (Le Goues et al. 2015) that contains 185 defects from 9 large open-source C projects, and Bugs.jar (Saha et al. 2018) that contains 1,158 faults from 8 popular open-source Java projects. Currently, one of the most popular faults benchmark is Defects4J (Just et al. 2014b); its initial version contains 357 real faults from 5 real-world Java projects. There are some appealing aspects of Defects4J that make it popular, such as 1) it is constructed from real-world projects; 2) its faults are reproducible, and each is accompanied with a failing test case that passes once the fault is fixed; 3) the faults are isolated, and the code changes that fix the faults do not contain irrelevant changes; 4) it includes a set of scripts that help the developer to get each fault from a project. This dataset has been popularly used for controlled testing and debugging studies (Sobreira et al. 2018; Pearson et al. 2017a). There has also been Bugswarm by (Tomassi et al. 2019) that contains pairs of failing and passing builds of Java and Python projects encapsulated in Docker images. However, an evaluation by

(Durieux and Abreu 2019) concluded that only a small percentage of its content is suitable for evaluating automated program repair and fault localization techniques due to issues such as the fault not being isolated. They also cited the need to download Docker containers for individual faults as another downside since it incurs high execution and storage cost on consumer-grade hardware. More recently, there is a faults benchmark on Python called BugsInPy (Widyasari et al. 2020), which is inspired by Defects4J. BugsInPy dataset is curated by hand to ensure that the faults are reproducible and isolated. BugsInPy comprises 493 faults from 17 real-world GitHub projects that have at least 10,000 stars, and like Defects4J, require relatively low overhead to retrieve each fault. Therefore, for this work, we choose BugsInPy to evaluate the SBFL techniques.

3 Dataset and Methodology

3.1 Fault Dataset

For this study, we use the BugsInPy dataset from (Widyasari et al. 2020). BugsInPy is a dataset comprising 493 real faults from 17 real-world Python projects, with each faulty program version comes together with the fixed version. Compared to Defects4J which has been used as the baseline dataset in many studies, BugsInPy is collected from a higher number of projects (i.e., 17 projects in BugsInPy compared to 6 projects in Defects4J). Moreover, the projects that are used in BugsInPy are popular GitHub projects with a high number of stars (i.e., more than 10,000 stars). Note that a project with a high number of starts usually corresponds to a high quality project (Ren et al. 2020). The projects in BugsInPy also span multiple domains including web framework, developer tool, and machine learning tool, which we believe will improve the generalizability of our results. As BugsInPy contains real faults, the erroneous portion of the program code may span multiple statements. Table 1 shows the statistics of the dataset.

3.2 Experiments Design

3.2.1 Research Questions

In this study, we seek to answer the following research questions:

– **RQ-1: Does the effectiveness of SBFL techniques on Python projects (in BugsInPy) differ from that on Java projects (in Defects4J)?**
  Previous study by (Pearson et al. 2017a) has evaluated several popular SBFL techniques on a dataset of Java projects. We want to investigate whether the performance of SBFL techniques on Python projects aligns with their findings on Java. This is necessary since no previous study has evaluated SBFL techniques on Python fault dataset even though Python is currently one of the most popular languages[3]. Identifying the potential differences in SBFL techniques' performance on BugsInPy versus Defects4J can shed more light into this unexplored research problem.
– **RQ-2: How effective are popular SBFL techniques on real faults of Python projects (in BugsInPy)?**

[3]https://insights.stackoverflow.com/survey/2020##technology-most-loved-dreaded-and-wanted-languages-wanted

Table 1 Projects and number of real faults available in a version of BugsInPy as of 19 June 2021

Project	Faults	LoC	Test LoC	# Tests	# Stars
ansible/ansible	18	207.3K	128.8K	20,434	43.6K
cookiecutter/cookiecutter	4	4.7K	3.4K	300	12.2K
cool-RR/PySnooper	3	4.3K	3.6K	73	13.5K
explosion/spaCy	10	102K	13K	1,732	16.6K
huge-success/sanic	5	14.1K	8.1K	643	13.9K
jakubroztocil/httpie	5	5.6K	2.2K	309	47K
keras-team/keras	45	48.2K	17.9K	841	48.6K
matplotlib/matplotlib	30	213.2K	23.2K	7,498	11.6K
nvbn/thefuck	32	11.4K	6.9K	1,741	53.9K
pandas-dev/pandas	169	292.2K	196.7K	70,333	25.4K
psf/black	15	96K	5.8K	142	16.4K
scrapy/scrapy	40	30.7K	18.6K	2,381	37.4K
spotify/luigi	33	41.5K	20.7K	1,718	13.4K
tiangolo/fastapi	16	25.3K	16.7K	842	15.3K
tornadoweb/tornado	16	27.7K	12.9K	1,160	19.2K
tqdm/tqdm	9	4.8K	2.3K	88	14.9K
ytdl-org/youtube-dl	43	124.5K	5.2K	2,367	67.3K
Total	493	1253.5K	486K	112,602	470.2K

Currently, the effectiveness of SBFL techniques in Python is still unknown as there has been no attempt to evaluate the performance of SBFL techniques in Python. We analyze how popular SBFL techniques perform in the BugsInPy dataset, and whether there is a "best" technique for use on real Python faults. Answering this question will help researchers and practitioners to characterize SBFL techniques' effectiveness in Python and identify potential areas of improvement.

– **RQ-3: Are the findings of previous studies applicable to Python projects (in BugsInPy)?**

The 7 finding statements from prior works that were checked by (Pearson et al. 2017a) (for example, "O^P outperforms Tarantula" and "Barinel outperforms Ochiai") are mostly made through experiments that use artificial faults to evaluate the SBFL techniques' performance. These finding statements are also made using Java or C projects. In this RQ-3, we use Python's real faults to evaluate these statements. Validating these statements is important as the previous study by (Pearson et al. 2017a) found that the results of artificial faults on Java do not portray the results of the real faults. Whether the same applies to Python projects is still unknown. As the Python language becomes more and more popular, there is an increasing need to investigate the validity of these statements on Python projects.

3.2.2 SBFL techniques

SBFL techniques exploit a program's test case results as well as their corresponding code coverage information to identify program units (e.g., statements, functions, etc.) that are most likely to be responsible for a failure. Many variants of SBFL techniques have been

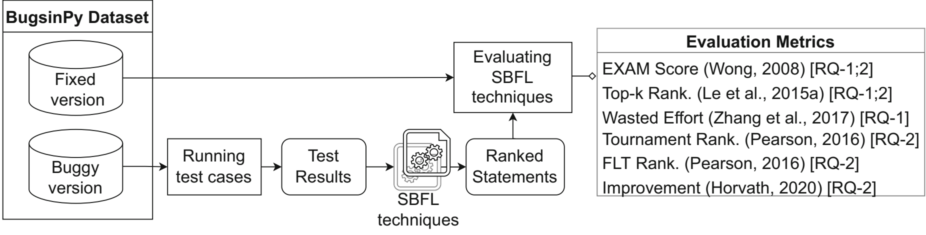


Fig. 1 Workflow of RQ-1 and RQ-2

proposed, and 5 of the most well-studied techniques (Wong et al. 2016) are Tarantula (Jones et al. 2001), Ochiai (Abreu et al. 2006), $\mathrm{O^P}$ (Naish et al. 2011b), BARINEL (Abreu et al. 2009b), and DStar (Wong et al. 2013). In this study, we investigate whether the findings found by (Pearson et al. 2017a) for Java can be replicated for other programming languages. For this purpose, we choose to evaluate these 5 SBFL techniques as well on Python projects (BugsInPy). These techniques are popular and also used in the previous study by (Pearson et al. 2017a) to evaluate SBFL on real faults of Java projects.

The formulas for the techniques are explained below, using the following notations: n_f denotes the number of total failing test cases, $n_f(s)$ denotes the number of failing test cases that execute statement s, n_p denotes the number of total passing test cases, and $n_p(s)$ denotes the number of passing test cases that execute statement s.

$$Tarantula(s) = \frac{\frac{n_f(s)}{n_f}}{\frac{n_f(s)}{n_f} + \frac{n_p(s)}{n_p}}$$

$$Ochiai(s) = \frac{n_f(s)}{\sqrt{n_f.(n_p(s) + n_f(s))}}$$

$$O^P(s) = n_f(s) - \frac{n_p(s)}{n_p + 1}$$

$$BARINEL(s) = 1 - \frac{n_p(s)}{n_p(s) + n_f(s)}$$

$$DStar(s) = \frac{n_f(s)^2}{n_p(s) + (n_f - n_f(s))}$$

3.2.3 Methodology

RQ-1. The overview of our methodology to answer RQ-1 is shown in Figure 1. We first obtain the statement coverage from the buggy version in the BugsInPy dataset using coverage.py[4], which is one of the most popular third-party coverage tools according to the official Python Developer's Guide[5]. From the coverage and test results, we obtain information required to apply SBFL, such as numbers of failing and passing test cases that execute a given statement, and obtain suspiciousness score for the statement.

In the scenario when some statements share the same suspiciousness score, we assign the average rank (Steimann et al. 2013; Pearson et al. 2017a) to these statements which are

[4]https://coverage.readthedocs.io/en/coverage-5.1/
[5]https://devguide.python.org/coverage/

calculated by $\left(\frac{n}{2}\right) + (k - 1)$ where n is the number of statements that have same suspiciousness score and k is the best rank of the statement (e.g., if the statement a, b, and c have the suspiciousness score 1, statements a, b, c will have rank $\left(\frac{3}{2}\right) + (1 - 1) = 1.5$). Since in our dataset it is possible to have a multi-statement fault, we evaluate the fault localization techniques using three debugging scenarios following the settings considered by (Pearson et al. 2017a):

- **Best-case**: Find any one of the faulty statements.
- **Worst-case**: Find all faulty statements.
- **Average-case**: Find 50% (half) of the faulty statements. In case of where the number of faulty statements is odd, we round it down, i.e. if the number of faulty statements is 7, this scenario requires 3 of them to be found.

We ran all test cases for every project in the dataset, with the exception of *pandas*. This is because *pandas* has a clear division of unit tests by component, which means localization can be performed by running a set of tests related to a specific component, instead of running the entire *pandas* test suite. For example, if failure occurs in a test for component *io*, we run tests only for component *io*.

Metrics that we used to get performance of SBFL techniques for this RQ-1 are mean *EXAM* score (Wong et al. 2008), Top-k (Le et al. 2015a), and wasted effort (Xuan and Monperrus 2014b). These metrics are used in all three debugging scenarios we consider, i.e., best-case, worst-case, and average-case. *EXAM* score (Wong et al. 2008) is one of the most popular metrics used to evaluate the effectiveness of fault localization techniques. The *EXAM* score shows the percentage of executable statements that need to be inspected until it reaches the first faulty statement. The formula of *EXAM* score is defined as follows:

$$EXAMScore = \frac{Rankofthefaultystatement}{Totalnumberofstatement}$$

EXAM score ranges from 0 to 1 (inclusive), with smaller score indicating the better performance of a SBFL technique. As an illustration of the *EXAM* score computation, consider the following suspiciousness score from five code statements (s_1, s_2, s_3, s_4, and s_5), which are 0.6, 0.7, 1.0, 0.5, and 0.8. Assume that s_2 is the faulty statement, the *EXAM* score will be $\frac{2}{5} = 40\%$, since the developer needs to inspect three statements (40% of the code base) to reach the faulty one.

As the *EXAM* score relies on the number of total statements in the program under study, a good *EXAM* score can still be achieved even if the faulty statement is not listed among the top results for cases where the total number statements are high. To mitigate this limitation of the *EXAM* score metric, we also use the top-k metric to compare the result from the SBFL techniques in Defects4J with the result of the best SBFL techniques in BugsInPy. This metric choice is also motivated by findings of (Parnin and Orso 2011) where the absolute rank is more important than the percentage ranking. Top-k measures how often the faulty statement is included in the highest-ranking k results, and for this part of the analysis, we use k values of 5, 10, and 200. We choose to include k=5 and k=10, since a survey with 386 practitioners done by (Kochhar et al. 2016) found that 73.58% and 98% of the practitioners only consider a fault localization instance to be successful if the faulty statement appears in the top 5 and 10 positions respectively. This is also supported by findings of (Parnin and Orso 2011) that highlight most programmers will move to traditional debugging when the faulty statements

are not found in the first few statements. We also utilized k=200 following the previous studies (Pearson et al. 2017b; Just et al. 2018) that also reported it for completeness-sake. Another study (Long and Rinard 2016) also found that automatic program repair methods work best when the program only includes the top-200 suspicious statements.

We also use wasted effort (Zhang et al. 2017) as one of the metrics. The wasted effort represents the number of statements that need to be checked before getting to the faulty statement. The smaller number of wasted effort indicates a better performance of an SBFL technique. We use wasted effort as it indicates how much effort that has been wasted as a consequence of the inaccurate fault localization (He et al. 2020). Several previous studies (Xuan and Monperrus 2014a; He et al. 2020; Zhang et al. 2017) also used wasted effort as their evaluation metrics.

After we obtain the results for both Defects4J and BugsInPy, we compare their performance using Wilcoxon rank-sum test (Wilcoxon 1992) to identify statistically significant differences following previous work by (Le et al. 2015a). We use this statistical test for Top-k, wasted effort, and *EXAM* score metrics for each considered SBFL technique. Following the work by (Le et al. 2015a), we also compute the effect size using Cliff's d effect size (Cliff 1993), with the following interpretation: negligible if $d<0.147$; small if $d>=0.147$, medium if $d>=0.33$, and large if $d>=0.474$ (Romano et al. 2006).

RQ-2. In RQ-2, we measure the effectiveness of popular SBFL techniques on real faults in Python projects. While on RQ-1 we focus on the comparison between datasets, our focus in RQ-2 is the comparison between SBFL techniques. Following the work of (Pearson et al. 2017a), which uses code changes in the fixed program version and suspiciousness ranking of the different statements in the buggy version, we use the following metrics to rank the SBFL techniques:

1. **Mean *EXAM* Score**
2. **Mean FLT (Fault Localization Technique) Ranking** (Pearson et al. 2017a): As we are examining 5 SBFL techniques, for every fault we rank the SBFL techniques from 1 to 5 where 1 indicates the best technique while 5 indicates the worst technique. The FLT rank value is based on the rank of fault for each technique. As an example, if fault X is ranked as number 1 in Tarantula, number 30 in Ochiai, number 2 in O^P, number 20 in DStar, and number 5 in Barinel, then, for fault X, the FLT rank for Tarantula, O^P, Barinel, DStar, and Ochiai is 1, 2, 3, 4, and 5 respectively. Afterward, we calculate the average rank for each technique.
3. **Tournament Ranking** (Pearson et al. 2017a): This ranking is computed by comparing pairs of SBFL techniques to determine whether one of the SBFL techniques gives a better result than the other. This is run for both *EXAM* score and FLT ranking, using the following approach: For each pair of SBFL techniques whose difference is statistically significant, as examined using Wilcoxon rank-sum test (Wilcoxon 1992), we assign 1 point to the winner. We subsequently rank the techniques based on the number of points.

Beyond this, we further analyze the result of SBFL techniques using two additional metrics: top-k and improvement. For the top-k metric, we use the same k values as in RQ-1, i.e. 5, 10, and 200. The improvement metric is designed based on the study by (Horváth et al. 2020). We consider 6 cases of improvement:

- $[201,\infty] \rightarrow [11,200]$: SBFL technique A ranks a faulty statement at a position larger than 200 and SBFL technique B ranks a faulty statement in a position between 11 to 200.

- [201,∞]→ [6,10]: SBFL technique A ranks a faulty statement at a position larger than 200 and SBFL technique B ranks a faulty statement in a position between 6 to 10.
- [201,∞]→ [1,5]: SBFL technique A ranks a faulty statement at a position larger than 200 and SBFL technique B ranks a faulty statement in top-5.
- [11,200]→ [6,10]: SBFL technique A ranks a faulty statement at a position between 11 to 200 and SBFL technique B ranks the faulty statement in a position between 6 to 10.
- [11,200]→ [1,5]: SBFL technique A ranks a faulty statement at a position between 11 to 200 and SBFL technique B ranks a faulty statement in top-5.
- [6,10]→ [1,5]: SBFL technique A ranks a faulty statement at a position between 6 to 10 and SBFL technique B ranks a faulty statement in top-5.

For this improvement metric, we only consider the faulty statement in the last position. For example, if in the average-case scenario there are 2 faulty statements in positions 5 and 20 of the result, we use the fault in position 20 to compute the improvement metric. From these 6 cases of improvement, we formulate a new metric we call "total improvement count", which is the total count of the 6 cases above. We present our results using this total improvement count. For a detailed breakdown of improvements based on the 6 cases, please refer to the Appendix. All metrics we use in RQ-2 are measured in the same three debugging scenarios we use in RQ-1, i.e., best-case, worst-case, and average-case.

Following previous work by (Le et al. 2015a), we use Wilcoxon rank-sum test (Wilcoxon 1992) to identify statistically significant differences between each pair of SBFL techniques. We apply this test to both the *EXAM* score and FLT ranking. This test is chosen as the data is not normally distributed, with d'Agostino-Pearson normality test (D'Agostino 1971; D'Agostino and Pearson 1973) rejecting the hypothesis of normality with a p-value less than 0.05. As the Wilcoxon rank-sum test is also able to handle the ranking data, which is ordinal (i.e. categorical data with set order), we also use it for the top-k and improvement metrics.

RQ-3. There have been many studies that evaluated and compared SBFL techniques. Summary of the previous studies on SBFL techniques is shown in Table 2. We choose the same set of previous studies whose claims were examined by (Pearson et al. 2017a). These claims that were examined are claims regarding the effectiveness of five different SBFL techniques. We summarize claims regarding the effectiveness between the pair of SBFL techniques that were evaluated by the set of studies. As an example, from the studies of artificial and real faults by (Naish et al. 2011b), artificial and real faults by (Moon et al. 2014), and artificial faults by (Pearson et al. 2017a), we retrieve the evaluations between O^P and Ochiai technique pair, where all three studies highlight that O^P performs better than Ochiai. Using real Python faults from BugsInPy, we subsequently examine the relative performance of each pair of SBFL techniques in best-case, worst-case, and average-case debugging scenarios. In other words, we compare the performance of 1) Ochiai versus Tarantula, 2) Barinel versus Ochiai, 3) Barinel versus Tarantula, 4) O^P versus Ochiai, 5) O^P versus Tarantula, 6) DStar versus Ochiai, and 7) DStar versus Tarantula. Then, we determine whether the distribution of *EXAM* scores between the two techniques is statistically significant. We use Wilcoxon rank-sum test (Wilcoxon 1992) with a significance threshold of 0.05 for statistical comparison and Cliff's d to compute effect size (Cliff 1993) following Le et al.'s study (2015a) since the results of normal distribution test show that the data comes from a non-normal distribution. Afterward, we compare the results on the real Python faults against claims made by the previous studies.

Table 2 Summary of the previous studies on SBFL techniques

Reference	Lang.	SBFL Rank (from best to worst)	Projects	Artif. Faults	Real Faults
Jones 2005	C	Tarantula	Siemens	122	-
(Abreu et al. 2007)	C	Ochiai, Tarantula	Siemens	120	-
(Abreu et al. 2009a)	C	Ochiai, Tarantula	Siemens, space	128	34
(Abreu et al. 2009b)	C	Barinel, Ochiai, Tarantula	Siemens, space, gzip, sed	141	38
(Ali et al. 2009)	C	Tarantula	Concordance	200	13
(Naish et al. 2011b)	C	O^P, Ochiai, Tarantula	Siemens, space	132	32
(Le et al. 2013)	C	Ochiai, Tarantula	Siemens, space, NanoXML, XML-Security	164	35
(Wong et al. 2013)	C	DStar, Ochiai, Tarantula	Siemens, space, ant, flex, grep, gzip, make, sed, Unix	436	34
(Moon et al. 2014)	C	O^P, Ochiai	space, flex, grep, gzip, sed	11	3
Xuan (2014b)	Java	Ochiai, Tarantula	JExel, JParsec, Jaxen, Commons Codec, Commons Lang, Joda-Time	1800	-
(Ju et al. 2014)	C, Java	DStar, Tarantula	printtokens, printtokens2, schedule, schedule2, tot-info, Jtcas, Sorting, NanoXML, XML-Security	104	-
(Le et al. 2015b)	C	DStar, Ochiai, Tarantula	Siemens, space, NanoXML, XML-Security	165	35
(Pearson et al. 2017a)	Java	O^P, DStar, Ochiai, Barinel, Tarantula	JFreeChart, Closure, Commons Lang, Commons Math, Joda-Time	3242	
		{DStar≈Ochiai≈ Barinel≈Tarantula}, O^P			323

4 Results

4.1 RQ-1

Top-k: To answer RQ-1, we compare the performance of the SBFL techniques on Defects4J (Just et al. 2014a) with their performance on BugsInPy (Widyasari et al. 2020). The top-k results for all the scenarios (i.e., best-case, average-case, worst-case scenarios) are shown in Table 3. For all the debugging scenarios, results on BugsInPy are lower for the same SBFL techniques (i.e., there is a smaller percentage of faults included in top-k). The percentage difference in top-k metric between BugsInPy and Defects4J results ranges from 11% to 44%, with all SBFL techniques localizing more faults in Defects4J within best-case, average-case, worst-case scenarios. Even though BugsInPy has a higher number of faults (493) than Defects4J (395), the absolute number of BugsInPy faults in Top-k is lower than Defects4J. We subsequently investigate whether this difference is statistically significant using the Wilcoxon rank-sum test. To use the Wilcoxon rank-sum in this setting, we compare the distribution of absolute ranks from faults in BugsInPy and Defects4J. The null hypothesis that we use for the statistical test is H_0 :the results come from the same distribution. We use a 5% significance level which means if the p-value is lower than 0.05, we can reject the null hypothesis and conclude that there is a statistically significant difference. We find that the result is statistically significant as shown in Table 3, indicating that the faults in BugsInPy are harder to localize using SBFL techniques than those in Defects4J.

Wasted effort: The wasted effort results from BugsInPy and Defects4J are shown in Table 4. The results show that for all the debugging scenarios (i.e., best-case, average-case, and worst-case), Defects4J has a lower value of wasted effort compared to BugsInPy. This indicates that the efforts that are wasted when localizing the fault using the recommendation from SBFL techniques are higher in BugsInPy. The differences between the BugsInPy and Defects4J results are statistically significant with medium to large effect size. These results indicate that the faults in BugsInPy are harder to localize using SBFL techniques compared to Defects4J, which is inline with the Top-k results.

EXAM: Table 5 shows the comparison of *EXAM* scores between Pearson et al.'s result that used Defects4J real faults and our results from BugsInPy for different types of scenarios. Our *EXAM* scores result shows an improvement compared to several previous studies (Le et al. 2013; Le et al. 2015a) that use artificial faults. However, compared to Pearson et al.'s result on Defects4J, the *EXAM* scores of the BugsInPy dataset on the best-case scenario are higher, By using Wilcoxon rank-sum statistical test, the result of the best-case scenario shows that the difference between *EXAM* scores of the evaluated techniques (Tarantula, Barinel, Ochiai, DStar, and O^P) are statistically significantly different compared to the results in the previous study. Moreover, the effect sizes are small for all the techniques, except for Tarantula and Barinel which have negligible effect sizes. For the average-case, the difference between the performance results of Tarantula, Barinel, and Ochiai is not statistically significant with negligible effect size. Meanwhile, for the O^P and DStar, both techniques have statistically significantly better performance in BugsInPy, compared to the Defects4J where the effect sizes are negligible and small, respectively. For the worst-case debugging scenario, all the techniques performance on BugsInPy are statistically significantly higher than Defects4J with negligible effect size except for DStar which has a small effect size. This shows that the scenarios affect the performance of the techniques with respect to the percentage of statements that need to be checked (i.e., the number of checked statements divided by the total number of statements).

Table 3 Top-k of BugsInPy and Defects4J, where higher percentage of fault that include in top-k indicate better performance

Technique	Top-5		Top-10		Top-200		p-value	d
	BugsInPy	Defects4J	BugsInPy	Defects4J	BugsInPy	Defects4J		
Best-case Debugging Scenario								
Tarantula	12.98%	**31.14%**	19.87%	**42.03%**	55.78%	**80.51%**	$6.1E-21$*	0.36 (M)
Barinel	12.57%	**31.14%**	19.87%	**42.03%**	55.78%	**80.51%**	$3.4E-21$*	0.37 (M)
Ochiai	14.19%	**32.41%**	19.87%	**42.53%**	50.30%	**81.77%**	$8.6E-26$*	0.41 (M)
DStar	9.53%	**32.15%**	14.19%	**42.03%**	38.74%	**82.53%**	$1.4E-43$*	**0.54** (L)
O^P	6.89%	**30.13%**	11.96%	**39.75%**	38.94%	**80.51%**	$2.6E-42$*	**0.53** (L)
Average-case Debugging Scenario								
Tarantula	5.68%	**16.96%**	9.74%	**25.32%**	33.27%	**67.34%**	$1.5E-30$*	0.44 (M)
Barinel	5.27%	**16.96%**	9.74%	**25.32%**	33.27%	**67.09%**	$8.8E-31$*	0.45 (M)
Ochiai	6.29%	**18.23%**	9.33%	**25.82%**	30.02%	**69.11%**	$4.1E-35$*	**0.48** (L)
DStar	4.46%	**18.23%**	7.51%	**26.08%**	23.94%	**69.37%**	$1.5E-46$*	**0.56** (L)
O^P	3.45%	**17.97%**	5.88%	**26.08%**	24.14%	**66.33%**	$2.5E-43$*	**0.53** (L)
Worst-case Debugging Scenario								
Tarantula	4.87%	**15.70%**	7.51%	**23.04%**	24.75%	**54.94%**	$2.2E-27$*	0.42 (M)
Barinel	4.46%	**15.70%**	7.51%	**23.04%**	24.75%	**54.94%**	$1.6E-27$*	0.42 (M)
Ochiai	5.27%	**16.71%**	7.10%	**23.04%**	22.72%	**56.20%**	$7.0E-30$*	0.44 (M)
DStar	3.85%	**16.71%**	5.68%	**23.29%**	17.85%	**56.71%**	$6.8E-37$*	**0.49** (L)
O^P	3.04%	**16.20%**	4.87%	**22.53%**	18.05%	**54.18%**	$3.3E-34$*	**0.48** (L)

Result in **bold** indicates dataset that has higher percentage of faults that are localized in top-k using the particular SBFL technique

"*" indicates the different between the absolute rank is statistically significant at 5% level

Cliff's d indicates effect size: large (L), medium (M), small (S), negligible (N)

Table 4 Wasted effort of BugsInPy and Defects4J, where the smaller wasted effort indicate better performance

Technique	BugsInPy	Defects4J	p-value	d
Best-case Debugging Scenario				
Tarantula	2158.47	429.03	$1.77E-20$*	0.36 (M)
Barinel	2158.47	429.05	$9.75E-21$*	0.36 (M)
Ochiai	2274.7	418.9	$2.64E-25$*	0.40 (M)
Dstar	2503.51	417.31	$5.03E-43$*	**0.53** (L)
O^P	2445.99	481.02	$9.14E-42$*	**0.53** (L)
Average-case Debugging Scenario				
Tarantula	5123.17	1038.62	$7.47E-29$*	0.44 (M)
Barinel	5122.98	1037.62	$4.49E-29$*	0.44 (M)
Ochiai	5184.69	1023.32	$2.35E-33$*	0.47 (M)
Dstar	5303.46	1023.79	$1.12E-44$*	**0.55** (L)
O^P	5247.37	1156.53	$1.10E-41$*	**0.52** (L)
Worst-case Debugging Scenario				
Tarantula	7321.33	2386.64	$7.15E-26$*	0.41 (M)
Barinel	7321.17	2386.64	$1.57E-27$*	0.42 (M)
Ochiai	7355.53	2377.35	$3.47E-28$*	0.43 (M)
Dstar	7433.58	2376.85	$4.09E-35$*	**0.48** (L)
O^P	7454.21	2538.31	$2.78E-32$*	0.46 (M)

"*" indicates the different between the wasted effort is statistically significant (at significance level of 5%).
Cliff's d indicates effect size: large (L), medium (M), small (S), negligible (N).

The worst-case *EXAM* score in BugsInPy shows higher performance compared to Defects4J, which suggests that BugsInPy has a lower percentage of statements that need to be checked. However, we note that the average of the total statements in the Defects4J is 14,322, while the average in BugsInPy is 34,098. As BugsInPy has a much higher number of statements, the overall number of statements that need to be checked in BugsInPy is higher compared to Defects4J, which indicates that the SBFL techniques perform worse on BugsInPy. We also compare the number of failed statements and failed test cases between BugsInPy and Defects4J. The number of statements that need to be fixed in BugsInPy averages 13.96 statements with a median of 2. Meanwhile, the number of fault statements in Defects4J has an average value of 3.56 with a median of 2. The average number of failed test cases that exist in Defects4J is 2.21 test cases with a median of 1. Comparatively, the number of failed statements of failed test cases in BugsInPy is much higher, with an average of 39 and a median of 4.

The lower result in BugsInPy may be affected by the trait of SBFL which oversimplifies the coverage information into the number of covering tests for each of the statements (Xie et al. 2016). As mentioned previously, the number of failed statements, total statements, and failed test cases are higher in BugsInPy compared to Defects4J. The oversimplification of coverage information may have bigger effects on more complex faults. In addition, SBFL techniques only consider coverage as the sole input, which means it cannot distinguish between program spectra with similar coverage (Xie et al. 2016). We observe that this

Table 5 *EXAM* scores of BugsInPy and Defects4J, where the smaller *EXAM* score indicate better performance

Technique	BugsInPy	Defects4J	p-value	d
Best-case Debugging Scenario				
Tarantula	0.064112	0.042541	0.04*	0.08 (N)
Barinel	0.064123	0.041179	0.03*	0.08 (N)
Ochiai	0.069589	0.040171	$9.3E-5$*	0.15 (S)
O^P	0.077843	0.047095	$5.5E-14$*	0.29 (S)
DStar	0.080444	0.040031	$3.9E-16$*	0.32 (S)
Average-case Debugging Scenario				
Tarantula	0.083056	0.089446	0.80	-0.01 (N)
Barinel	0.083065	0.088084	0.84	-0.01 (N)
Ochiai	0.089152	0.087853	0.22	0.05 (N)
O^P	0.096655	0.100237	$4.2E-4$*	0.14 (N)
DStar	0.099188	0.128669	$1.9E-5$*	0.17 (S)
Worst-case Debugging Scenario				
Tarantula	0.143346	0.192170	0.02*	0.08 (N)
Barinel	0.143353	0.190808	0.02*	0.09 (N)
Ochiai	0.148211	0.191270	$4.7E-3$*	0.11 (N)
O^P	0.153765	0.205877	$3.7E-4$*	0.14 (N)
DStar	0.156084	0.196445	$2.4E-5$*	0.16 (S)

"*" indicates the different between the *EXAM* scores is statistically significant (at significance level of 5%).
Cliff's d indicates effect size: large (L), medium (M), small (S), negligible (N).

situation where multiple program spectra have similar coverage (i.e., same number of fail and pass) comes up frequently in BugsInPy. This shows that BugsInPy has a different trait than Defects4J, making it an interesting aspect to analyze further. In RQ-2 and RQ-3 we analyze deeper the results of SBFL techniques in BugsInPy.

Finding 1: The faults in BugsInPy are harder to identify using SBFL techniques compared to the faults in Defects4J. This is indicated by the lower performance results of SBFL techniques in BugsInPy compared to Defects4J in terms of several metrics (Top-k, wasted effort, and *EXAM* score). The results of top-k and wasted effort metrics in BugsInPy are significantly lower with medium and large effect sizes. The percentage of faults that are included in the top-k for Defects4J is twice the percentage of BugsInPy faults included in the top-k.

4.2 RQ-2

Top-k in best-case scenario: First, we discuss the results for the best-case scenario as introduced in Section 3.2.3. We use top-k metrics to measure the performance of each SBFL technique, i.e. we measure how frequent each SBFL technique is able to rank the fault statement within top-5, top-10, and top-200. The result, shown in Table 6, is in line with

the other metrics, except for Ochiai's performance. Ochiai performs better than Tarantula and Barinel on top-5, even though it has a lower percentage than both on top-200. Applying statistical tests on the set of output fault statement ranks for every pair of techniques being compared (e.g., Tarantula versus Barinel, Barinel versus Ochiai), we find that Tarantula, Barinel, and Ochiai perform better (with $p<0.05$) than DStar and O^P for all Top-5, Top-10, and Top-200. The detailed comparison of Top-k metrics is shown in Table 20 of the Appendix.

***EXAM* scores, tournament ranking, and FLT ranking in best-case scenario:** The effectiveness of the SBFL techniques on the Python faults in terms of *EXAM* scores, tournament ranking, and FLT ranking is shown in Table 7. The lower *EXAM* score and the FLT rank, the better the technique performance. Based on the *EXAM* score and FLT ranking, we found that Tarantula performs best with an *EXAM* score of 0.06411 and an FLT ranking of 1.88. However, the differences between Tarantula and Barinel, as well as the differences between Tarantula and Ochiai, are not statistically significant. The three techniques are shown to perform better compared to DStar and O^P ($p < 0.05$). This result is in line with the previous finding where we use other metrics (i.e., *EXAM* score, FLT ranking, and tournament ranking) in which Tarantula, Barinel, and Ochiai are superior to DStar and O^P.

The result of the SBFL technique ranking in terms of the mean *EXAM* score shows consistency with the FLT ranking, except for O^P, which has a lower *EXAM* score compared to DStar. However, we note that the difference between O^P and DStar's *EXAM* scores is not statistically significant ($p = 0.76477$) and both techniques have the same ranking on the tournament ranking for *EXAM* score. Moreover, for FLT ranking, DStar is significantly better than O^P ($p = 1.06E-5$).

Improvement in best-case scenario: In addition to previous evaluations, we also investigate the improvement for each pair of SBFL techniques. The total improvement count is shown in Table 8, while the detailed improvements (i.e., 6 cases improvement) are reported in Table 21 in the Appendix. The number of improvement from Table 8 presents the number of faults in which technique A perform better than another technique B. For example, Tarantula improves a total of 123 faults over DStar. To examine the statistical significance of each reported improvement, we apply the Wilcoxon rank-sum test. The Wilcoxon rank-sum test produces a statistically significant difference for all comparisons except for improvements of Barinel against Tarantula and O^P against DStar. The total improvement count of Barinel against Tarantula is zero while the total improvement count of O^P against DStar is 3. Even though the total improvement count of Tarantula against Barinel is also small, the improvement is statistically significant, with the faults ranked lower than 5 in Barinel being improved to top-5 in Tarantula. Based on Table 8, we can see that Tarantula provides the best result, with a slight improvement compared to Barinel and Ochiai. Tarantula and Barinel produce the biggest improvements over DStar and O^P, with both techniques improve 142 faults compared to O^P and 123 faults compared to DStar. While Ochiai outperforms DStar

Table 6 Percentage of fault statements that appear within Top-5, Top-10, and Top-200 in best-case debugging scenario

Technique	Top-5	Top-10	Top-200
Tarantula	12.98%	**19.87%**	**55.78%**
Barinel	12.57%	**19.87%**	**55.78%**
Ochiai	**14.19%**	**19.87%**	50.30%
DStar	9.53%	14.19%	38.74%
O^P	6.89%	11.96%	38.94%

Result in **bold** is the best for the category

Table 7 SBFL techniques sorted by various metrics (i.e., *EXAM* score, FLT ranking, tournament ranking for *EXAM* score, and tournament ranking for FLT ranking respectively) in best-case debugging scenario

Technique	*EXAM* Score	FLT Ranking	#Better (*EXAM* Score)	#Better (FLT Ranking)
Tarantula	0.06411	1.8874	2	3
Barinel	0.06412	1.9006	2	3
Ochiai	0.06958	2.5993	2	2
DStar	0.08044	3.1622	0	1
O^P	0.07784	3.4391	0	0

A smaller value in *EXAM* score and FLT ranking indicates a better performance. While higher value on tournament ranking (i.e., # Better) indicates better performance or how many the technique perform statistically significantly better than other techniques

and O^P, there is a smaller improvement than Tarantula and Barinel. An interesting point is that there are a few faults on which DStar and O^P outperform Tarantula, Barinel, and Ochiai. In summary, each technique outperforms other techniques with respect to some bugs, with the exception of Tarantula that does not outperform Barinel on any fault. This suggests the value of combining SBFL techniques to improve fault localization performance on the BugsInPy dataset.

> **Finding 2: Tarantula performs the best on real Python faults** in terms of *EXAM* score, FLT ranking, tournament ranking, top-k, and improvement in the best-case scenario. However, the differences compared to Barinel and Ochiai are not statistically significant. While DStar and O^P generally perform worse than the other three techniques in terms of all metrics, they still boost some faults in improvement metrics, indicating that there are certain cases in which they are more effective.

Top-k in average-case and worst-case scenario: In addition to the best-case debugging scenario, we also conduct an analysis for average-case and worst-case debugging scenarios. The result for top-k in the different debugging scenarios is shown in Table 9. All three

Table 8 Improvement of SBFL techniques A over B, where A are the techniques in the top row and B are the techniques in the most left row. The number in each cell presents the number of faults in which a technique performs better than another technique. For example, Tarantula improves the ranking result of two faults against Barinel

B \ A	Tarantula	Barinel	Ochiai	DStar	O^P
Tarantula	0	0	19*	17*	20*
Barinel	2*	0	21*	19*	22*
Ochiai	44*	44*	0	7*	10*
DStar	123*	123*	92*	0	3
O^P	142*	142*	111*	19*	0

"*" indicates the difference between the improvement is statistically significant

scenarios produce the same SBFL technique ranking. In line with results from the best-case scenario, Tarantula, Barinel, and Ochiai significantly outperform DStar and O^P in both average-case and worst-case. Ochiai outperforms DStar and O^P in all Top-k evaluations (i.e., k=5, k=10, and k=200). Table 20 of the Appendix shows the detailed results of the comparison of the techniques using the top-k metric. The rank performance of SBFL techniques using top-k metrics is consistent in different debugging scenarios.

***EXAM* scores, tournament ranking, and FLT ranking in average-case and worst-case scenario:** We report the resulting *EXAM* scores for the average-case and worst-case scenarios in Table 10. For the average-case scenario, Tarantula ranks first, but without significant differences to Barinel and Ochiai. Meanwhile, for the worst-case debugging scenario, there are no significant differences between all techniques, although here Tarantula also ranks first. The value of the *EXAM* score on the average-case and worst-case scenario are higher than the best-case scenario showing lower performance. This indicates that for the *EXAM* score, the different debugging scenarios have a significant effect on the *EXAM* score result. For FLT ranking, the result is also shown in Table 10. The results on all debugging scenarios for FLT ranking produce a consistent ranking. The tournament ranking also shows the same ranking as the best-case debugging scenario. This indicates that different debugging scenarios do not affect results for the FLT ranking.

Improvement in average-case and worst-case scenario: The total improvement count metric in average-case and worst-case debugging scenarios are shown in Table 11, while the detailed improvement figures are reported in Tables 22 and 23 in the Appendix. In the best-case scenario, Tarantula and Barinel give the biggest improvement over DStar and O^P. Improvements from Ochiai over DStar and O^P are bigger than the improvements from Tarantula and Barinel over Ochiai. In the average-case and worst-case debugging scenario, Ochiai, DStar, and O^P also produce an improvement over Tarantula and Barinel. This suggests the possibility of combining these techniques to achieve a better overall result. From this result we can also infer that if a fault localization tool user already uses Tarantula, they will not benefit from also running Barinel, however, some benefits may be obtained by running Ochiai, DStar, or O^P. For this part of the analysis, the improvement results for almost all technique pairs in both average-case and worst-case are statistically significant.

Table 9 Top-k on average-case and worst-case debugging scenario

Techniques	Top-5	Top-10	Top-200
Average-case Debugging Scenario			
Tarantula	5.68%	**9.74%**	**33.27%**
Barinel	5.27%	**9.74%**	**33.27%**
Ochiai	**6.29%**	9.33%	30.02%
DStar	4.46%	7.51%	23.94%
O^P	3.45%	5.88%	24.14%
Worst-case Debugging Scenario			
Tarantula	4.87%	**7.51%**	**24.75%**
Barinel	4.46%	**7.51%**	**24.75%**
Ochiai	**5.27%**	7.10%	22.72%
DStar	3.85%	5.68%	17.85%
O^P	3.04%	4.87%	18.05%

Result in **bold** is the best for the category

Table 10 SBFL techniques sorted by various metrics (i.e., *EXAM* score, FLT ranking, tournament ranking for *EXAM* score, and tournament ranking for FLT ranking respectively) on average-case and worst-case debugging scenario

Technique	*EXAM* Score	FLT Ranking	#Better (*EXAM* Score)	#Better (FLT Ranking)
Average-case Debugging Scenario				
Tarantula	0.083056	2.00347	2	3
Barinel	0.083065	2.01018	2	3
Ochiai	0.089152	2.61555	2	2
DStar	0.099188	3.03839	0	1
O^P	0.096655	3.27146	0	0
Worst-case Debugging Scenario				
Tarantula	0.143346	2.07988	0	3
Barinel	0.143353	2.08806	0	3
Ochiai	0.148211	2.60322	0	2
DStar	0.156084	2.95583	0	1
O^P	0.153765	3.16553	0	0

The exceptions are the improvement of Barinel over Tarantula in both scenarios and the improvement of O^P over DStar in the average-case scenario.

Finding 3: Performance rankings of SBFL techniques on different debugging case scenarios are consistent, i.e. Finding 2 (Tarantula performs the best on real Python faults) holds for all 3 debugging scenarios – best-case, average-case, and worst-case.

Top-k in different types of faults: There are several types of faults in BugsInPy, such as single-line faults, multi-line faults, and faults of omission.

Table 11 Improvement on Average-case and Worst-case Debugging Scenario where the top row is the improvement against the column (e.g., in average-case scenario Tarantula improve 63 faults against DStar, in worst-case scenario Ochiai improve 6 faults against Tarantula)

B \ A	Tarantula	Barinel	Ochiai	DStar	O^P
Average-case Debugging Scenario					
Tarantula	0	0	8*	9*	9*
Barinel	2*	0	10*	11*	11*
Ochiai	26*	26*	0	4*	6*
DStar	63*	63*	44*	0	2
O^P	71*	71*	54*	10*	0
Worst-case Debugging Scenario					
Tarantula	0	0	6*	7*	9*
Barinel	2*	0	8*	9*	11*
Ochiai	18*	18*	0	4*	6*
DStar	47*	47*	35*	0	2*
O^P	53*	53*	41*	6*	0

Multi-line faults are faults where the changes to fix the fault span over more than one line. Meanwhile, single-line faults are faults that can be fixed by modifying one line. Changes to fix faults in real faults may consist of only code addition rather than the modification of existing code. In BugsInPy, there are approximately 13% of faults that have a missing statement(s) without faulty statements. We refer to these cases as the fault of omission. The location of the changes will be complicated in the fault of omission when the developer inserts the new code at some lines while there may be other locations that are also valid choices to fix the fault. Consider the fault of omission in Fig. 2, we can insert the conditional that is missing (*if* statement) before line 1, between line 2 and line 3, or after line 4.The bottom part of Fig. 2 shows the valid placement of the conditional statement.

The fault localization technique's output which includes any of the above possible statements in Fig. 2 is just as useful as showing the line the developer has chosen. Thus, we conducted manual analysis for the fault of omission and provided an alternative location where the faults can be fixed. Figure 3 shows the distribution of the type of faults based on the three categories on BugsInPy.

We wanted to further analyze whether the results of the SBFL technique would be different on these different types of faults. We find that the performance ranking of SBFL techniques is consistent for all debugging scenarios. We also consider the finding of (Pearson et al. 2017a) which states that "the best-case debugging scenario is the best approximation of user real cases". For the experiments involving the different types of faults (e.g., single-line fault, multi-line faults, and faults of omission), we only calculate the metrics in the best-case debugging scenario. We subsequently measure the top-k metrics for each SBFL technique using different types of faults. The result of these measurements are shown at Table 12. We find that the ranking of techniques on top-k measurement is consistent for each type of fault. Moreover, we also find that the ranking of techniques is in line with our findings using previous metrics. As with the previous statistical test result on all faults, the test result on different types of faults shows the statistically significant difference on Tarantula, Barinel, and Ochiai compared to DStar and O^P, where the three techniques are outperforming DStar and O^P.

***EXAM* scores, tournament ranking, and FLT ranking in different types of faults:** Table 13 show the experiment results on different types of faults using *EXAM* scores, FLT ranking, and tournament ranking. We note that the ranking of SBFL techniques on all

```
1      if (x.isExist()):
2          return true
3      if (x.isNew()):
4          return true
5 +    if (x.isRemove()):
6 +        return true
```

```
1 +  if (x.isRemove()):
2 +      return true
3    if (x.isExist()):
4        return true
5    if (x.isNew()):
6        return true
```

```
1    if (x.isExist()):
2        return true
3 +  if (x.isRemove()):
4 +      return true
5    if (x.isNew()):
6        return true
```

```
1    if (x.isExist()):
2        return true
3    if (x.isNew()):
4        return true
5 +  if (x.isRemove()):
6 +      return true
```

Fig. 2 Example of fault omission (top) with the valid placement of the fixing statements (bottom)

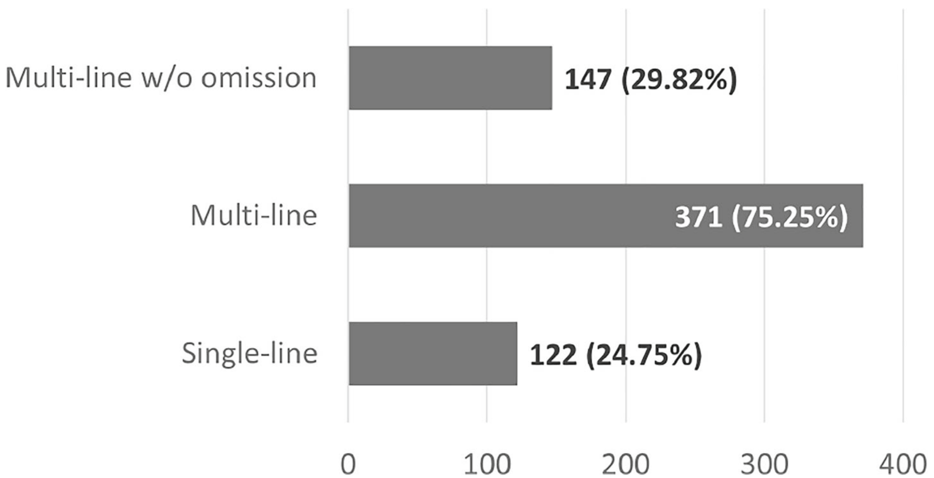


Fig. 3 Distribution of faults following Pearson et al. (2017) in BugsInPy

metrics is consistent among different types of faults, with the best scores achieved on a multi-line fault without fault of omission. This finding is expected as it is easier to determine the location of any of the multiple buggy lines (i.e., multi-line fault) compared to finding the exact location of a single buggy line (i.e., single-line fault). In case of fault of omission, SBFL may produce worse results as the buggy program itself do not actually contain faulty statements and good candidate locations for insertion of needed statement may not be considered by the SBFL technique. Overall, these results are in line with results reported by (Pearson et al. 2017a), where they found that the best *EXAM* score is achieved for the multi-line faults (excluding faults of omission), followed by all multi-line faults (including faults of omission), and single-line faults.

Finding 4: The performance ranking of SBFL techniques on all metrics is consistent among the different types of faults, with the best scores achieved on multi-line faults (excluding faults of omission), followed by all multi-line faults, and single-line faults.

4.3 RQ-3

Check the claim of SBFL effectiveness from 7 prior studies in Python real faults: For RQ-3, we investigate the 7 finding statements from prior studies that have also been checked by (Pearson et al. 2017a). The comparison of SBFL techniques obtained from our experiments on Python faults, as well as the results reported by Pearson et al. on Java faults and other preceding studies, are shown in Table 14. The second columns of Table 14 show prior comparison results using Java and C artificial faults, while the third and fourth columns show the prior comparison result from (Pearson et al. 2017a) using real faults. The rightmost two columns of Table 14 show the result of our study. The BugsInPy result shows

Table 12 Percentage of fault statements that appear within Top-5, Top-10, and Top-200 on different type of faults

Technique	Top-5	Top-10	Top-200
Single-line without omission			
Tarantula	12%	18%	52%
Barinel	11%	18%	52%
Ochiai	14%	18%	49%
DStar	9%	12%	39%
O^P	8%	11%	40%
Multi-line without omission			
Tarantula	18%	24%	66%
Barinel	18%	24%	66%
Ochiai	20%	24%	58%
DStar	14%	18%	42%
O^P	11%	16%	42%
Multi-line with omission			
Tarantula	13%	20%	57%
Barinel	13%	20%	57%
Ochiai	14%	20%	51%
DStar	10%	15%	39%
O^P	6%	12%	39%

disagreements with results of previous comparisons on Java and C artificial faults. The differences are found in the relative performances between O^P and DStar compared with Ochiai and Tarantula (rows 4, 5, 6, and 7 in Table 14). Based on BugsInPy results, O^P and DStar have worse performance (differences which are also statistically significant) compared to Ochiai and Tarantula respectively. However, the effect sizes of the differences are small, except for the comparison between DStar and Ochiai which has a negligible effect size. This finding refutes the previous finding on Java and C artificial faults where O^P and DStar perform better compared to Ochiai and Tarantula. Comparisons between Ochiai, Tarantula, and Barinel in rows 1, 2, and 3 Table 14 show that the Wilcoxon test results are not statistically significant. We also found the effect sizes to be negligible.

Check the claim of SBFL effectiveness from Pearson et al. (2016) that used Java real faults (Defects4J): We further compare the agreement of the 7 key statements between Python real faults (BugsInPy) and Java real faults (Defects4J). In Java real faults all 7 key statements are refuted by having insignificant differences for all comparisons of SBFL techniques in the key statements. On the contrary, based on our result in Python's real faults, we find that there are key statements that are refuted with statistically significant differences. These cases are found on the key statements which compare O^P and DStar with Ochiai and Tarantula. Results for the average-case and worst-case scenarios, shown in Table 24 and 25 in the Appendix, also refuted the 7 key statements in prior works.

Table 13 SBFL techniques sorted by various metrics (i.e., *EXAM* score, FLT ranking, tournament ranking for *EXAM* score, and tournament ranking for FLT ranking respectively) on different type of faults

Technique	*EXAM* Score	FLT Ranking	#Better (*EXAM* Score)	#Better (FLT Ranking)
Single-line fault that were not fault of omission				
Tarantula	0.0732	1.9508	2	3
Barinel	0.0733	1.9877	2	3
Ochiai	0.0764	2.6352	2	2
DStar	0.0915	3.0861	0	1
O^P	0.0882	3.3361	0	0
Multi-line faults that were not fault of omission				
Tarantula	0.030513	1.7312	2	3
Barinel	0.030516	1.7448	2	3
Ochiai	0.035092	2.5782	2	2
DStar	0.046242	3.3299	0	1
O^P	0.045512	3.6292	0	0
Multi-line faults that included fault of omission				
Tarantula	0.061101	1.8665	2	3
Barinel	0.061102	1.8719	2	3
Ochiai	0.067339	2.5876	2	2
DStar	0.076799	3.1873	0	1
O^P	0.074429	3.4730	0	0

Finding 5: Similar to Pearson et al.'s 2017a study on Java bugs, our study on Python programs does not show any support to the 7 key finding statements reported in prior works. Our finding shows there are statistically significant differences that refute the key statements that compare O^P and DStar with Ochiai and Tarantula. The results are different from findings of (Pearson et al. 2017a) that do not uncover statistically significant differences.

The average-case scenario results of BugsInPy also produce statistical differences on key statements that compare O^P and DStar with Ochiai and Tarantula with small and negligible effect sizes. Meanwhile, in the worst-case scenario, the key statements are all refuted by having statistically insignificant differences between key statements with a negligible effect size for all comparisons. Our results show a consistency of SBFL technique rank (i.e., the rank of SBFL technique by mean *EXAM* score) for all the scenarios. Compared to the Defects4J results, the SBFL techniques that perform better (i.e., having a lower mean *EXAM* score) are different, and at the same time, the differences are insignificant for all statements in different scenarios. For example, for the best-case scenario, the mean *EXAM* score of DStar is lower than Ochiai while for the average-case DStar has a higher mean *EXAM* score than Ochiai.

In our experiments, the relative performances of the different techniques on real Python faults do not match with the results on artificial faults reported by previous studies (Le et al. 2013; Abreu et al. 2009b; Le et al. 2015b; Naish et al. 2011a; Wong et al. 2016;

Table 14 Comparison results of preceding studies using Java/C artificial faults (second column), Defects4J Java real faults (third and fourth columns), and BugsInPy Python real faults (fifth and sixth columns) in best-case debugging scenario

No.	Previous comparisons on Java or C artificial faults	Defects4J Results (2017a)		BugsInPy Results	
	Winner > Loser	Agree?	Effect size	Agree?	Effect size
1.	Ochiai > Tarantula*	*(insig.)*	-0.02 (N)	*(insig.)*	0.04 (N)
2.	Barinel > Ochiai (Abreu et al. 2009b)	*(insig.)*	0.02 (N)	*(insig.)*	-0.04 (N)
3.	Barinel > Tarantula (Abreu et al. 2009b)	*(insig.)*	$1.9E{-}5$ (N)	*(insig.)*	0.002 (N)
4.	O^P > Ochiai (Naish et al. 2011a)	*(insig.)*	0.04 (N)	no	0.15 (S)
5.	O^P > Tarantula †	*(insig.)*	0.02 (N)	no	0.21 (S)
6.	DStar > Ochiai‡	*(insig.)*	-0.03 (N)	no	0.14 (N)
7.	DStar > Tarantula**	*(insig.)*	-0.02 (N)	no	0.19 (S)

∗(Le et al. 2013; Le et al. 2015b; Naish et al. 2011a; Wong et al. 2016; Xuan and Monperrus 2014b)

†(Naish et al. 2011a; Moon et al. 2014) ‡(Le et al. 2015b; Wong et al. 2016)

∗∗(Le et al. 2015b; Wong et al. 2016; Ju et al. 2014)

Whether the results *agrees* indicates p-value: $p<0.05$, *($p \geq 0.05$)*.

Cliff's d indicates effect size: large (L), medium (M), small (S), negligible (N)

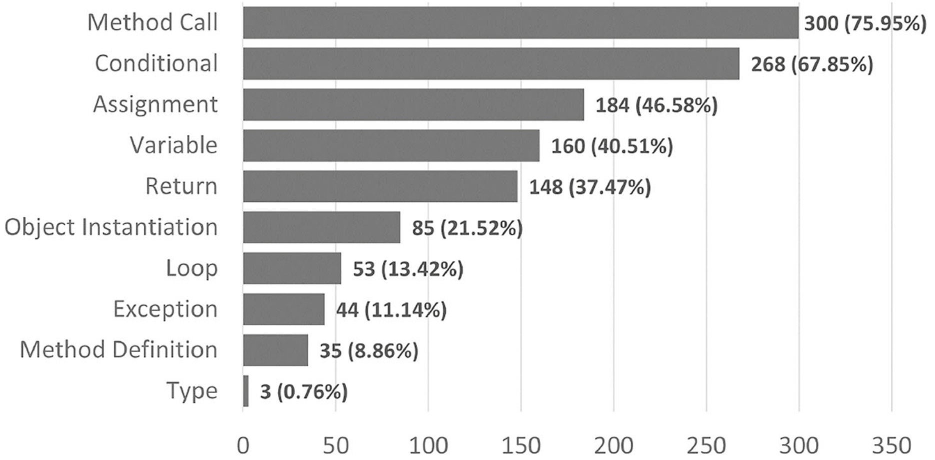
Xuan and Monperrus 2014b; Moon et al. 2014; Ju et al. 2014), except for Barinel having a lower *EXAM* score than Ochiai. On the other hand, our results agree with Pearson et. al.'s observations on Java's real faults, namely, the performance of SBFL techniques on artificial faults is different from the real faults. This further corroborates their observation that the SBFL techniques' performances on artificial faults are not useful predictors for their performance on real faults.

> **Finding 6:** Claim from the previous studies regarding the performance of SBFL techniques does not hold on Python real faults (BugsInPy) on all debugging scenarios (i.e., best-case, average-case, and worst-case).

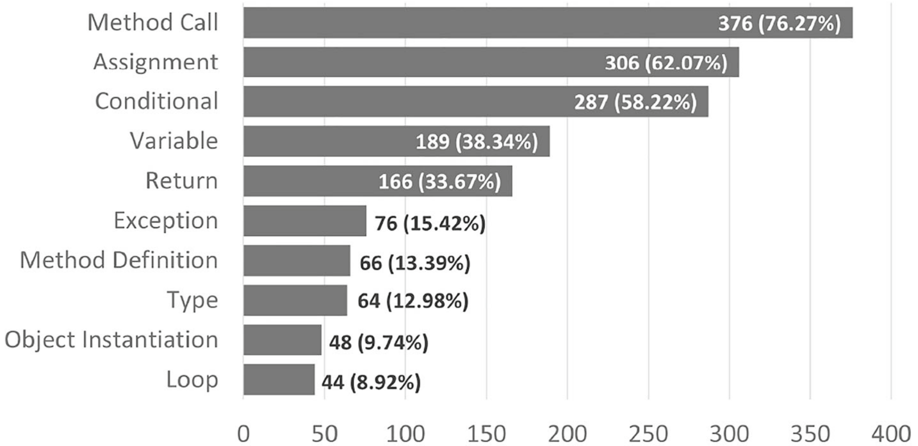
5 Discussion and Implications

5.1 On the fairness of comparison between BugsInPy and Defects4J

There are no significant differences between the distribution of faults in BugsInPy and Defects4J (except for Type, Object Instantiation, and Assignment categories). Following the categorization of faults that are described in Section 5.2, we would like to investigate the fairness of comparison between BugsInPy and Defects4J. For this purpose, we collect the distribution of the fault categories from the labeled faults. The distribution of the category of faults in Defects4J and BugsInPy can be found in Fig. 4. Based on the distribution,



(a)



(b)

Fig. 4 Distribution of fault category in (a) Defects4J and (b) BugsInPy

we observe that for both BugsInPy and Defects4J, the fault category with the highest occurrences is the Method Call, occurring in 76.27% and 75.95% of the datasets, respectively. We also find that many of the fault categories in BugsInPy have similar distribution with those in Defects4J, such as the Variable category (40.51% in Defects4J and 38.34% in BugsInPy) and Return category (37.47% in Defects4J and 33.67% in BugsInPy). The differences in the distribution of faults category are found in the Type and Assignment category, which is

more prevalent in BugsInPy which has 64 faults (16.20%) and 306 faults (62.07%) respectively, compared to the Defects4J which only has 3 faults (0.61%) and 184 faults (46.58%) respectively. Another difference is from the Object Instantiation category where Defects4J has a higher number of faults which is 85 (21.52%), compared to BugsInPy which has 48 faults (9.74%). Nevertheless, the difference between the number of faults in Defects4J and BugsInPy for each category is small, ranging from 0.3 to 15%.

We also run a Chi-Square test (Tallarida and Murray 1987) with a 1% significance level (Rayson et al. 2004; Martinez and Monperrus 2015; Ruthruff et al. 2005) to determine whether the differences between the distribution for every fault category are significant. The p-value results of the Chi-Square test are higher than 0.01 for 7 fault categories, which are Loop (p-value = 0.036), Method Definition (p-value = 0.076), Exception (p-value = 0.138), Return (p-value = 0.221), Variable (p-value = 0.431), Conditional (p-value = 0.019), and Method Call (p-value = 0.736). The p-value of more than 0.01 indicates that there are no significant differences between the faults in BugsInPy and Defects4J for the 7 fault categories. Meanwhile, for Object Instantiation, Type, and Assignment fault categories, the p-value results from Chi-Square test are smaller than 0.01. Specifically, the p-values are $2.43E-06$, $1.25E-10$, and 0.003 respectively. Through these observations, we find that the faults in BugsInPy and Defects4J have similar distributions except for the Object Instantiation, Type, and Assignment categories.

The SBFL performance on the subset of fault categories are inline with the whole dataset. Furthermore, we run additional experiments using the subset of fault categories for a fair comparison. In these additional experiments, we compare the performance of the SBFL techniques for each subset of the dataset based on the category of faults (e.g., comparing evaluation results from Defects4J Method Call faults against BugsInPy Method Call faults, etc.). Table 15 shows the results in the Top-k metric for the three most frequent fault categories in both datasets (i.e., Method call, Assignment, and Conditional). From these additional experiments on the subset of the dataset, we find that the results are inline with our previous results on the whole dataset. Specifically, we find that the differences between the fault localization evaluation results are statistically significantly higher in Defects4J (except for the Type category where we found to have higher results in Top-5 and Top-10 in BugsInPy compared to Defects4J).

5.2 On the potential factors that affect the SBFL results

We analyze the faults in BugsInPy to determine the factors that affect the SBFL results. We find that there are several factors that may lead to changes in the performance of SBFL techniques, including (1) the faults nature, (2) the quality of the test cases, and (3) the programming language nature.

The fault's nature in the dataset may affect the performance of the SBFL technique. Based on the RQ-1 results, we found that the number of failed test cases and the number of statements that need to be fixed are higher on the BugsInPy compared to the Defects4J. These may indicate that the faults in BugsInPy are more complicated than those in Defects4J, as the effort required to fix the fault (i.e., in terms of the number of statements that need to be changed) is higher. This may be one of the reasons that make the SBFL techniques perform better in Defects4J compared to the BugsInPy.

High code coverage (i.e., the absolute ratio of covered code) does not always correlate with good SBFL results. One of the ways to measure the quality of the test cases is by using code coverage. Code coverage determines whether the test cases are covering the code and how much of the code is covered by the test cases. For example, if we have 10 statements

Table 15 Top-k of BugsInPy and Defects4J for three highest faults type (i.e., Method Call, Assignment, and Conditional) in best case debugging scenario, where higher percentage of fault that include in top-k indicate better performance

Technique	Top-5		Top-10		Top-200		p-value	d
	BugsInPy	Defects4J	BugsInPy	Defects4J	BugsInPy	Defects4J		
Method Call								
Tarantula	15.16%	**31.33**%	22.07%	**41.33**%	56.12%	**80.67**%	1.39E-14*	0.34 (M)
Barinel	14.63%	**31.33**%	22.07%	**41.33**%	56.12%	**80.67**%	7.92E-15*	0.35 (M)
Ochiai	16.22%	**32.33**%	22.07%	**41.67**%	50.53%	**81.67**%	3.59E-18*	0.39 (M)
Dstar	10.37%	**32.0**%	15.43%	**41.0**%	39.63%	**82.33**%	1.96E-32*	**0.53** (L)
O^P	7.71%	**31.0**%	13.03%	**39.67**%	39.89%	**80.33**%	4.17E-32*	**0.53** (L)
Assignment								
Tarantula	12.75%	**30.43**%	20.59%	**41.3**%	55.88%	**82.61**%	1.72E-12*	0.38 (M)
Barinel	12.42%	**30.43**%	20.59%	**41.3**%	55.88%	**82.61**%	1.31E-12*	0.38 (M)
Ochiai	14.05%	**32.07**%	20.92%	**41.85**%	50.33%	**83.7**%	1.69E-15*	0.43 (M)
Dstar	9.15%	**31.52**%	14.71%	**41.85**%	39.22%	**84.78**%	2.52E-25*	**0.56** (L)
O^P	6.21%	**29.35**%	12.09%	**39.13**%	39.54%	**83.7**%	1.47E-25*	**0.56** (L)
Conditional								
Tarantula	11.5%	**29.48**%	18.12%	**40.67**%	54.7%	**79.48**%	3.25E-14*	0.37 (M)
Barinel	11.15%	**29.48**%	18.12%	**40.67**%	54.7%	**79.48**%	2.28E-14*	0.37 (M)
Ochiai	11.85%	**30.97**%	18.12%	**41.04**%	50.52%	**80.6**%	2.30E-17*	0.42 (M)
Dstar	7.67%	**30.6**%	12.54%	**39.93**%	37.28%	**81.72**%	4.35E-29*	**0.55** (L)
O^P	5.57%	**29.1**%	10.1%	**38.06**%	37.28%	**80.22**%	3.69E-28*	**0.54** (L)

Result in **bold** indicates dataset that has higher percentage of faults that are localized in top-k using the particular SBF technique

* indicates the different between the absolute rank is statistically significant at 5% level

Cliff's d indicates effect size: large (L), medium (M), small (S), negligible (N)

in file A, with test case B executing/covering 7 statements from file A, then the statement coverage is 70%. We calculate the statement coverage of both datasets (i.e., BugsInPy and Defects4J). For projects in BugsInPy, we use coverage.py to calculate the statement coverage. Meanwhile, we use GZoltar to calculate the coverage of Java code in Defects4J. We measure statement coverage as we evaluate the SBFL on the statement granularity.

Table 16 shows the statement coverage from projects in Defects4J, while Table 17 shows the statement coverage from projects in BugsInPy. The average value of statement coverages for Defects4J is 66% while BugsInPy has 71% statement coverage. Both measurements show comparatively high statement coverage. However, the SBFL performance metrics for BugsInPy are lower compared to those in Defects4J. This indicates that good coverage calculated by the absolute ratio of covered code does not always correlate with good SBFL results. For example, we can see that the Math project from Defects4J has the lowest statement coverage while having the highest percentage of faults that are localized in the Top-200. Meanwhile, the Mockito project has the highest statement coverage but has the highest EXAM score, which indicates a lower performance of the SBFL technique. These occurrences are also found in BugsInPy, where the black project has a high coverage but the EXAM score is comparatively high compared to the other projects.

Language nature has some impacts toward the SBFL performance. We found that some characteristics of the programming language affect the SBFL performance. For example, in Java, each variable is associated with a fixed data type (as Java employs static type checking). If the variable is an integer type, then only the integer can be assigned to the variable throughout the execution of the program. Each variable in Java also needs to be declared before being used, which means that the variable is bound to its data type. Meanwhile, in Python, it is not possible to declare a variable. Rather, values are directly assigned to the variable through assignment, without the need of indicating the variable's type. The type of variables may also change during the program execution in Python (as Python employs dynamic type checking). For example, we can assign a string to the variable, use the variable as a string, and then assign an integer to the same variable.

During the analysis of the changes that fix the fault, we found that many faults occurred because the type of variable is not the one expected in the test case. To fix this type of fault, developers need to convert the type of the variable to the expected type. The conversion of variable type when fixing the fault is usually done using a method call that does explicit type conversion. This can be done using the built-in functions of Python such as *str()*, *int()*, *float()*, etc. For example, we can convert a string to an integer by calling the function *int()*. This will also check whether the string that is being converted to an integer only contains a numerical value. If it detects otherwise, it will raise an error. We found that this type of

Table 16 Statement coverage of Defects4J with the Top-K and *EXAM* score for each project using Tarantula

Project	Statement Coverage	Top-5	Top-10	Top-200	*EXAM*
Chart	61.10%	26.92%	57.69%	88.46%	0.048
Closure	71.64%	15.79%	24.81%	67.67%	0.023
Lang	73.69%	46.15%	60%	90.77%	0.059
Math	48.99%	36.79%	46.23%	91.51%	0.036
Mockito	79.41%	34.21%	42.11%	76.32%	0.071
Time	77.83%	48.15%	51.85%	74.07%	0.01

Table 17 Statement coverage of BugsInPy with the Top-K and *EXAM* score for each project using Tarantula

Project	Statement Coverage	Top-5	Top-10	Top-200	*EXAM*
ansible	51%	0.00%	0.00%	11.11%	0.062
black	91%	13.33%	20.00%	53.33%	0.133
cookiecutter	79%	0.00%	0.00%	75.00%	0.107
fastapi	65%	18.75%	18.75%	43.75%	0.049
httpie	84%	0.00%	0.00%	60.00%	0.163
keras	69%	0.00%	0.00%	46.67%	0.055
luigi	64%	0.00%	0.00%	39.39%	0.035
matplotlib	71%	0.00%	0.00%	23.33%	0.106
pandas	69%	12.43%	18.93%	63.31%	0.048
PySnooper	43%	0.00%	0.00%	66.67%	0.23
sanic	90%	0.00%	0.00%	60.00%	0.077
scrapy	78%	45.00%	60.00%	92.50%	0.033
spacy	64%	10.00%	30.00%	60.00%	0.156
thefuck	81%	56.25%	87.50%	93.75%	0.011
tornado	82%	0.00%	0.00%	37.50%	0.112
tqdm	78%	11.11%	44.44%	88.89%	0.128
youtube-dl	51%	0.00%	2.33%	27.91%	0.099

change to fix the fault involving explicit type conversion is unique in Python (BugsInPy) and is not found in Defects4J.

To further determine the impact of this fault characteristic, we analyzed the faults in BugsInPy and categorized them. Specifically, we followed the previous study (Sobreira et al. 2018; Vancsics et al. 2020; Pan et al. 2009) that investigated how specific faults were fixed by the developers. In other words, we categorized the faults based on the repair action of a developer. We used a classification scheme from (Sobreira et al. 2018) that was previously used to categorize faults in Defects4J. (Sobreira et al. 2018) proposed the following categories: Assignment, Conditional, Loop, Method Call, Method Definition, Object Instantiation, Exception, Return, Variable, and Type. Their definitions adjusted for the context of Python programs are provided in Table 18. As an example, considering the code changes shown in Figure 5, we find that the changes to fix the fault are related to the following categories:

– Conditional: line 9, Conditional Branch Addition (addition of a simple *if*)
– Exception: line 10, Exception Addition (addition of *raise* statement)
– Method Call: line 9, Method Call Addition (addition of method call inside *if* statement)

We analyze the changes that fix the fault to have a better understanding of the changes that were done by the developer. For every changed file, the two authors label the data independently to prevent bias in the labeling process. If there are differences in the label, the authors conduct a discussion to resolve the differences. The inter-rater agreement value calculated using Cohen's kappa (Cantor 1996) is 0.76, which is interpreted as excellent agreement (DeVellis 2005). Note that one fault can have multiple labels as it may need multiple repair actions to fix it.

Table 18 Fault Categories based on Repair Patterns and Their Descriptions

Category	Description
Assignment	This category includes changes made on simple assignment operator, unary increment, decrement operator, and assignment compound of an arithmetic operator. The changes can be derived from the addition, modification, and removal of assignments. For example, modification of increment to decrement.
Conditional	This category includes the changes that are made on the conditional statement such as *if*, *if-else*, *if-elif-else*, and *else*. The changes can be in the form of conditional branch addition or removal. It can also be the modification of the conditional expression.
Loop	The loops considered in Python are for and while. The changes that are related to this category are loop addition, removal, and modification.
Method Call	The changes included in this category are the ones related to the method call. This includes cases containing modification of a method call (e.g., moving method call, changing a parameter in a method call, etc.), adding an invocation of a method call, and removing an existing method call.
Method Definition	The changes related to the method definitions (MD) are MD addition, MD removal, MD modification (e.g., change method parameters, method renaming, etc.)
Object Instantiation	In Java this category is observed by the keyword new. Meanwhile, Python does not utilize the new keyword. Rather, the object instantiation pattern in Python is similar to a method call. Therefore, to determine whether the statement is an object instantiation, we need to check whether the invoked method call is the name of a class or not. The following are some examples of object instantiation in Python:

```
1 Class Point:
2     def __init__(self):
3         self.x = 0
4         self.y = 0
5 p = Point() #Object instantiation of type
      Point
```

```
1 from point import Point q = Point() #Object
      instantiation of imported type Point
```

Category	Description
Exception	The changes that are included in this category are related to the exception handling, such as addition or removal of a try-catch block or raise statement.
Return	The changes that are included in this category are those whose changes are related to the *return* statement.
Variable	Changes that change the variable declaration and modification in the usage of variables are included in this category. For example, adding new variable declarations, replacing the usage of a variable with another variable, replacing the usage of a method call with a variable, etc. In Python, the pattern for variable declarations is the same as assignments. Thus, we need to check whether the variable already existed (i.e., already declared) before in the code to determine whether it is considered as an assignment or variable declaration.
Type	Changes that utilize explicit type conversion. The explicit type conversion is done by using a method call to convert variable types such as *str()*, *int()*, *long()*, etc. For example, *str()* can be used to change the variable type from integer to string. The fault-fixes that are included in this category are related to the explicit type conversion, such as addition, removal, or modification.

```
1  diff --git a/pandas/core/dtypes/cast.py
2  b/pandas/core/dtypes/cast.py
3  --- a/pandas/core/dtypes/cast.py
4  +++ b/pandas/core/dtypes/cast.py
5  @@ -823,6 +823,8 @@ def astype_nansafe(arr, dtype, copy: bool = True, skipna: bool =
        False):
6          if is_object_dtype(dtype):
7              return tslib.ints_to_pydatetime(arr.view(np.int64))
8          elif dtype == np.int64:
9  +           if isna(arr).any():
10 +               raise ValueError("Cannot convert NaT values to integer")
11             return arr.view(dtype)
```

Fig. 5 Example of code changes from Pandas 101 (https://github.com/pandas-dev/pandas/commit/27b713ba677869893552cbeff6bc98a5dd231950)

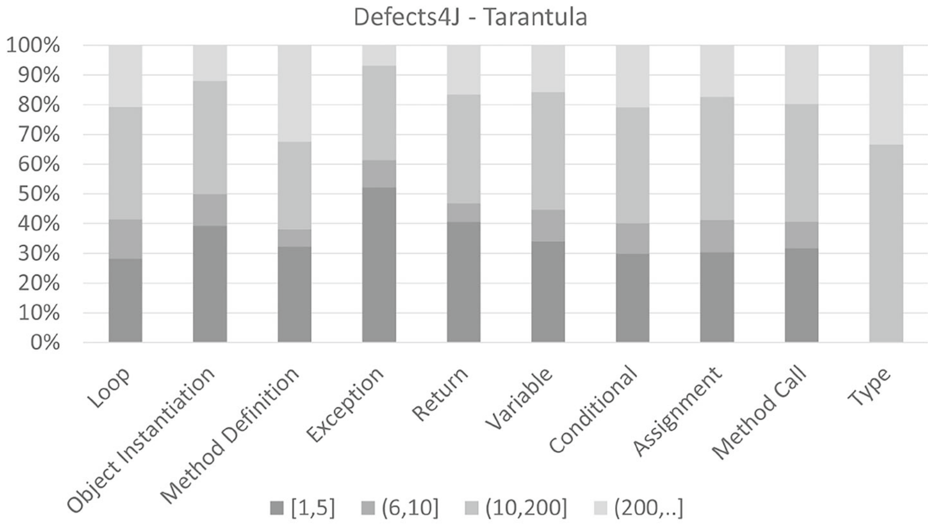
Then, following the categorization, we run SBFL techniques on the subset of faults category for both BugsInPy and Defects4J. We counted faults that are localized into non-overlapping intervals of [1; 5], (5; 10], (10; 200], or (200;...]. For example, [1;5] means that the fault is localized in between 1 to 5 position, (5;10] mean that the fault is localized between 6 to 10 position, and so on. This *interval statistic* is shown in Figure 6. The color of the bar in Fig. 6 indicates the faults that are localized at a specific interval. For example, from the figure we can observe that for BugsInPy, using Tarantula, there are 22% of faults that are localized in the Top-5 for the Return category.

Based on the analysis of the subset of faults category, we find that some differences are due to the language nature. Specifically, we find differences in the Type and Object Instantiation categories. Our analysis found that the three faults in the Type category from Defects4J are different from the one on BugsInPy. Specifically, we found the following Type faults in Defects4J:

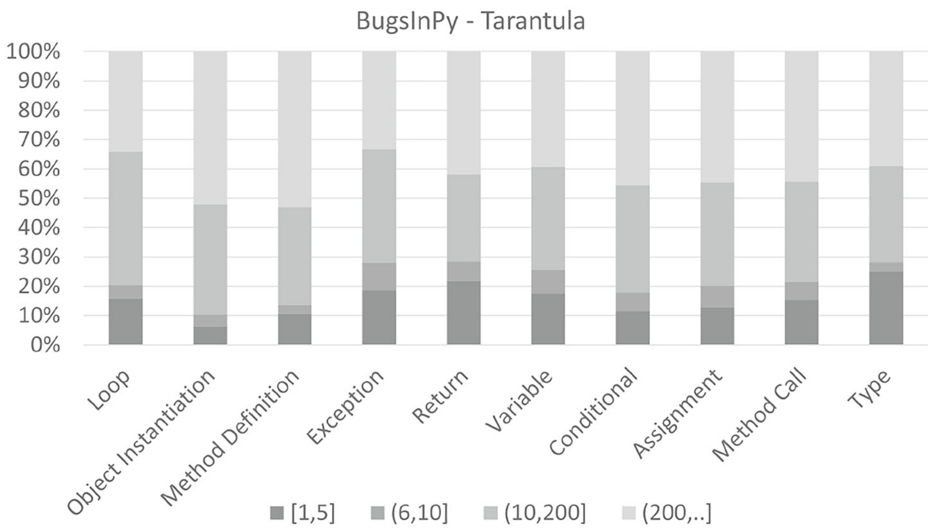
- Type addition: shown in Fig. 7.
- Type implemented interface modification: shown in Fig. 8 and 9.

Meanwhile, in BugsInPy there are 64 faults out of 493 (13%) which are fixed by explicitly changing the variable type into another type using the built-in type conversion function in Python. We compare the evaluation results of SBFL techniques for faults of the Type category to those of the other categories. We found that the Type category has the highest percentage of faults that are localized in the Top-5 and Top-10 by SBFL techniques. Meanwhile, the performance of SBFL techniques for the Type category subset in Defects4J does not localize any of the faults in Top-5 and Top-10. This highlights that Type category faults are localized better using the SBFL techniques in BugsInPy (i.e., Top-5 and Top-10).

Another difference based on the language nature is found in the object instantiation pattern. Object instantiation in Python does not utilize the "new" keyword. Rather, it is done similarly to a method call. Comparing the performance of the Object Instantiation category between Defects4J and BugsInPy, we find that BugsInPy has much worse performance. Specifically, Object Instantiation is the second-best performing category in terms of Top-5 and Top-10 in Defects4J. Meanwhile, in BugsInPy, the Object Instantiation category performs the worst, far different from the category's performance in Defects4J. It is possible that the different pattern in the object instantiation between Python and Java results in the worse performance in BugsInPy.



(a)



(b)

Fig. 6 Interval statistic for every subset fault category in (a) Defects4J and (b) BugsInPy using Tarantula. The color of the bar indicate the faults that are localized on the specific interval. For example, there are 16% of faults that are localized in Top-5 for the Loop category in BugsInPy

```
1 +  abstract class SerializableAnswer implements Answer<Object>, Serializable {
2 +  }
```

Fig. 7 Changes to fix fault in Mockito 23, type addition

```
1 -   Map<TemplateType, JSType> inferred =
2 +   Map<TemplateType, JSType> inferred = Maps.filterKeys(
3 -       inferTemplateTypesFromParameters(fnType, n);
4 +       inferTemplateTypesFromParameters(fnType, n),
5 +       new Predicate<TemplateType() {
6 +         @Override
7 +         public boolean apply(TemplateType key) {
8 +           return keys.contains(key);
9 +         }}
10 +      );
```

Fig. 8 Changes to fix fault in Closure 112, type implemented interface modification

5.3 On the impact of potential biases toward SBFL techniques

The set of faults in BugsInPy does not bias the results toward a particular SBFL technique. We also investigate whether the set of faults in BugsInPy bias the results towards a particular technique. To analyze the impact of this potential bias, we conduct an additional evaluation to check this hypothesis. For this purpose, we apply the five SBFL techniques for each category of fault subset in BugsInPy (e.g., Method Call, Conditional, Assignment, etc.). Table 19 shows the results in the three most commonly found categories of faults (i.e., Method Call, Assignment, and Conditional). We find that the results of the evaluation on each fault category are inline with the previous finding, where we found that the Tarantula SBFL technique performs the best but does not have a statistically significant difference compared to Barinel and Ochiai. Considering that we have the same findings in the evaluation using the whole dataset and using the subset of the dataset, we believe that the set of faults in BugsInPy does not bias the results towards a particular technique.

5.4 On the effect of fault categories toward SBFL performance

The category of faults in BugsInPy affects the performance of SBFL technique. We are also interested in whether the different category of faults in BugsInPy has any effect on the performance of the SBFL technique. To analyze this, we conduct an evaluation by applying the SBFL techniques on the subset of faults based on the categories described in Section 5.2. Figure 10 shows the results from Ochiai and DStar SBFL techniques. It shows the number of faults that are localized into non-overlapping intervals of [1; 5], (5; 10], (10; 200], or (200;...]. For example, [1;5] means that the fault is localized in between 1 to 5 position, (200,...] means that the fault is localized in the position higher than 200, and so on. We find that in the Type category, followed by the Return, Exception, and Variable, SBFL techniques localize the highest percentage of faults in top-5 and Top-10 compared to the other fault categories. We also find that for Object Instantiation and Method Definition categories, it is more likely that the faults are not localized in the top-5 and top-10 positions.

```
1 -   implements RandomGenerator {
2 +   implements RandomGenerator,
3 +           Serializable{
```

Fig. 9 Changes to fix fault in Math 12, type implemented interface modification

Table 19 Percentage of fault statements that appear within Top-5, Top-10, and Top-200 on different type of faults (repair action) in best case debugging scenario

Technique	Top-5	Top-10	Top-200
Method Call			
Tarantula	15%	**22%**	**56%**
Barinel	15%	**22%**	**56%**
Ochiai	**16%**	**22%**	51%
Dstar	10%	15%	40%
O^P	8%	13%	40%
Assignment			
Tarantula	13%	**21%**	**56%**
Barinel	12%	**21%**	**56%**
Ochiai	**14%**	**21%**	50%
Dstar	9%	15%	39%
O^P	6%	12%	40%
Conditional			
Tarantula	11%	**18%**	**55%**
Barinel	11%	**18%**	**55%**
Ochiai	**12%**	**18%**	51%
Dstar	8%	13%	37%
O^P	6%	10%	37%

Result in **bold** indicates dataset that has higher percentage of faults that are localized in top-k

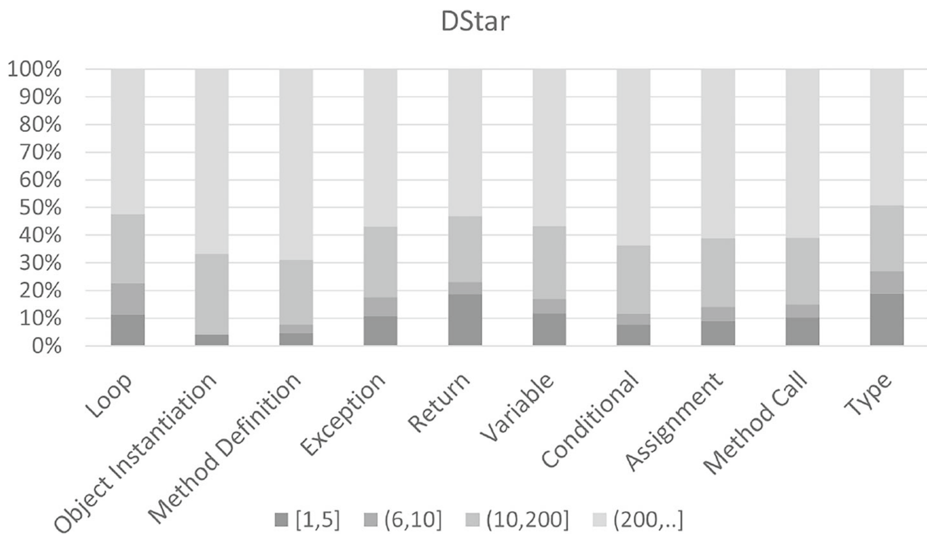
These findings indicate that there is a relationship between the category of faults and the effectiveness of fault localization.

5.5 Implications of Our Findings

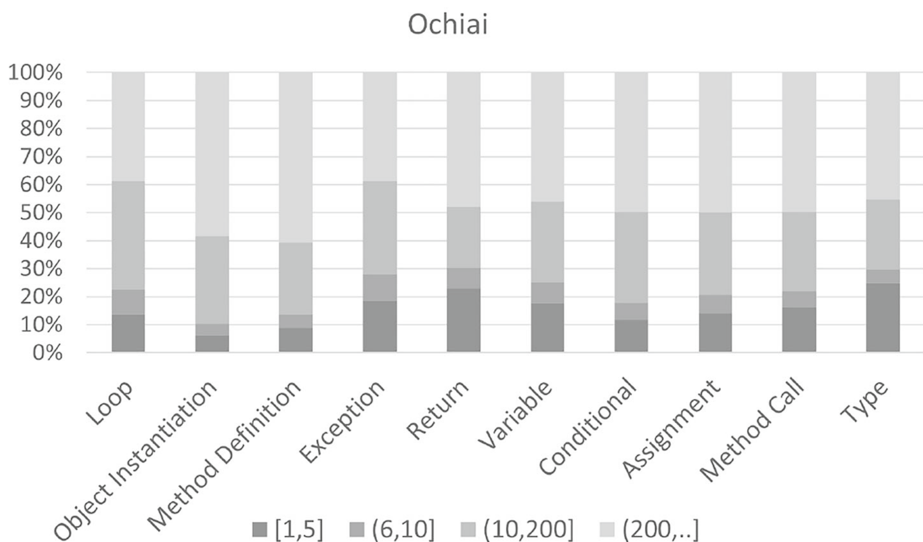
Our study corroborates Pearson et al.'s findings on Java real faults, where the relative performances of examined SBFL techniques on Python real faults either do not match the findings of preceding studies on artificial faults or have statistically insignificant differences. This has several implications for practitioners and researchers:

Seeking the "absolute best" SBFL technique is not so important in practice. For practitioners, the lack of significant difference in effectiveness between the top 3 popular SBFL techniques (Tarantula, Barinel, and Ochiai) on real-world Python faults implies that the selection of SBFL tools should not be overly focused on the technique used, at least for Python projects. As long as the tools being considered to use one of the top three techniques, selection efforts should be focused on other factors such as ease-of-use, ability to integrate with a currently-used development environment, or quality of documentation.

Investigation of other features in addition to code coverage may prove to be beneficial. From our evaluation, we find that the faults in BugsInPy are harder to identify compared to faults in Defects4J. A cause of this is the occurrences of faults that are difficult to localize due to different program spectra having the same coverage. Such faults occur with high enough frequency to cause statistically significant differences between the



(a)



(b)

Fig. 10 Interval statistic for every subset fault category in BugsInPy using (a) DStar and (b) Ochiai. The color of the bar indicate the faults that are localized on the specific interval. For example, there are 23% of faults that are localized in Top-10 for the Loop category using DStar

examined SBFL techniques' effectiveness on BugsInPy faults and their effectiveness on Defects4J faults. This highlights a practical issue with the application of these SBFL techniques on real-world projects, and emphasizes the need to research ways to mitigate this issue. For example, work by (Sohn and Yoo 2017) extend SBFL with code and change metrics, such as size, age, and code churn. They use machine learning to process SBFL suspiciousness values from existing SBFL formulas with code and change metrics as features. In addition, this also emphasizes the need to examine the prevalence of similar issues in other popular languages, such as JavaScript, Ruby, or C#. An empirical study of real-world projects written in those languages will enable researchers and practitioners alike to gain a more realistic view of the general level of performance that can be expected from current popular SBFL techniques, particularly in their language of choice.

Appropriate metrics are important when evaluating SBFL techniques. Our experimental results measured using different metrics highlight how the choice of metrics can affect the ranking of a technique. For example, DStar produces a better result than O^P on top-k and improvement metrics. However, O^P *EXAM* score is lower than DStar, which indicates that O^P performs better. The discrepancy highlights the value of studying the suitability of existing metrics for different use cases to facilitate the usage of consistent metrics across different studies. This investigation may be conducted with help of practitioners, for example as with a recent study by (Parnin and Orso 2011) which finds that practitioners value the absolute ranking of the fault localization result rather than the percentage.

Evaluations of SBFL techniques should be done on real faults. Our finding that newer techniques do not translate to better performance on real Python faults (and may even be outperformed by old technique) emphasizes that researchers should evaluate future SBFL technique developments on real faults, to ensure that the new techniques will indeed perform significantly better in real-world situations. This echoes observations in (Pearson et al. 2017a). Since there may be situations in which the collection of a large amount of real faults is not feasible, it is also important for researchers to gain a better understanding of the characteristics of real faults and ways to generate more realistic artificial faults (along the lines of works by (Patra and Pradel 2021) and (Tufano et al. 2020)). These will enable researchers to use artificial faults in SBFL experiments while getting results that more closely mirror how the techniques will perform on real faults.

There is value in investigating effective SBFL technique combination. Our study shows that in certain cases, lower-performing SBFL techniques can still produce an improvement over the higher-performing techniques. This highlights the benefit of allocating more research effort to better ways to combine the techniques. Currently, there have been some works proposing a combination of SBFL techniques (Xuan and Monperrus 2014a; Lucia et al. 2014), as well as augmentation of SBFL techniques, for example through the addition of PageRank algorithm (Zhang et al. 2017). More broadly, there have been works proposing combinations involving SBFL and other families of fault localization techniques. These include a hybrid between SBFL and mutation-based fault localization (MBFL) proposed by (Pearson et al. 2017b), and a learning-to-rank approach proposed by (Zou et al. 2019) to combine techniques from different families of fault localization techniques. However, while such approaches to combine techniques from different families are reported to yield good results, a limitation of such combination is that some technique families (particularly MBFL) can be orders of magnitude slower than SBFL, which is a lightweight technique. Therefore, research into a more effective combination or augmen-

tation of SBFL will remain important for applications where performance is a priority. Further, the improvements produced by lower-performing techniques also indicate a need to better understand how specific fault characteristics benefit one SBFL technique over another. Improved understanding of the interaction between fault characteristics and different techniques' performance will also benefit practitioners, as they will be able to make a more informed choice regarding technique based on their specific project and common fault types.

5.6 Threats to Validity

A source of threat to construct validity is the suitability of the metrics we use for our evaluation. A number of metrics have been proposed to measure the performance of fault localization techniques. To ensure the suitability of the metrics we use, we include metrics that are frequently used in prior works related to the evaluation of SBFL techniques such as improvement (Horváth et al. 2020), *EXAM* score (Abreu et al. 2009b; Naish et al. 2011a; Wong et al. 2016; Pearson et al. 2017a; Le et al. 2015a), top-k (Pearson et al. 2017a; Horváth et al. 2020; Le et al. 2015a), FLT ranking (Pearson et al. 2017a), and tournament ranking (Pearson et al. 2017a).

A source of threat to internal validity is the possibility of faults in the SBFL score computation. At the time of writing, we are not aware of any Python libraries that provide the implementation of SBFL techniques, so we use our own implementation of the techniques. We mitigate the risk of incorrect implementation using two approaches. First, for each of the SBFL techniques, we follow the formula defined in prior works (Abreu et al. 2006; Jones et al. 2001; Wong et al. 2013; Naish et al. 2011b; Abreu et al. 2009b). Second, to avoid errors in coverage calculation, we utilize coverage.py[6], a popular library to compute coverage of a Python project, before applying the different SBFL techniques to the resulting coverage. We believe this reduces the threat of potential implementation errors. Further, we have also created a replication package so that other researchers can validate our findings. The replication package is available at https://github.com/soarsmu/Evaluating_SBFL_BugsInPy.

Another threat to internal validity relates to the construction of the Python fault dataset (BugsInPy). In Defects4J, to make sure that the fault is isolated, the authors did a preprocessing step to make sure that each version only contains a single fault. For example, if version A contains two different faults F1 and F2, version A is divided into two different buggy versions that handle each fault individually. Similarly, in the creation of BugsInPy, to make sure that each version is isolated, the authors of BugsInPy analyze whether the fault fixing for every committed version is isolated (i.e., the changes do not include other fault fixing or cosmetic changes). If the changes to fix the fault are not isolated, the authors of BugsInPy remove the fault from the dataset. For each version of fault, two authors of BugsInPy labeled the data independently to reduce bias. They only take the fault version that is labeled as isolated by all the labelers. Considering this approach, we believe that the threat of the versions that are not isolated for BugsInPy and Defects4J is minimal.

[6] https://coverage.readthedocs.io/en/coverage-5.1/

The threat to the external validity of our study relates to the generalizability of our findings. We attempt to mitigate this threat by choosing a dataset that comprises a diverse range of software and faults. In this work we focus on Python programs and the findings here may not generalize to other programming languages. We encourage more researchers to replicate our findings in additional popular languages (e.g., Javascript).

In this study, we find several differences in the performance of SBFL techniques between BugsInPy and Defects4J datasets, which may indicate that there are some differences in the fault patterns between these two datasets written in Java and Python, respectively. However, our findings do not indicate that the differences are only solely due to the programming language differences. Rather, we find that there are multiple factors that contribute to the differences (i.e., programming language, faults' nature, etc.). Future research is encouraged in this direction.

6 Conclusion and Future Work

In this work, we perform an evaluation of five popular SBFL techniques on a set of 493 real faults in 17 real-world Python projects. The results of our SBFL technique comparison using Python real faults (in BugsInPy) are in line with the findings of (Pearson et al. 2017a) on Java (in Defects4J), which contradicts some claims made in preceding studies done on artificial faults. Our results suggest that the claim from the previous studies regarding the performance of SBFL techniques does not hold on Python real faults (in BugsInPy). This emphasizes that future developments of SBFL techniques should be done in conjunction with evaluations on real faults to ensure new techniques can indeed produce better performance (as compared to classic ones like Tarantula) in real-world situations. Further, our finding indicates the need to understand the characteristics of real faults that are currently not well-represented in artificial fault datasets. Our finding also suggests that given the choice between tools that implement Tarantula, Barinel, or Ochiai, practitioners can simply choose based on characteristics such as ease-of-use or documentation quality, due to the statistically insignificant performance difference between the three techniques. Our analysis of real faults from the BugsInPy dataset we use reveals that the faults are harder to identify using SBFL techniques compared to Java's real faults in Defects4J. This finding (together with the fact that Python is now more popular than Java) highlights that BugsInPy is a challenging and realistic dataset that SBFL researchers may want to consider in the future.

We encourage future research to investigate factors and characteristics of defects and program spectra that most significantly impact SBFL performance. Another direction for future work is to investigate the most effective way to combine different SBFL techniques to boost overall performance.

Appendix

Table 20 Cliff's d effect size and Wilcoxon rank-sum test results on top-k metrics in all debugging scenarios (i.e., best-case, average-case, worst-case)

Technique	Tarantula			Barinel			Ochiai			DStar			OP2		
	Top-5	Top-10	Top-200	Top-5	Top-10	Top-200	Top-5	Top-10	Top-200	Top-5	Top-10	Top-200	Top-5	Top-10	Top-200
Best-case Debugging Scenario															
Tarantula	–			*(-0.05) N*	*(-0.03) N*	*(-0.004) N*	*(0.07) N*	*(0.05) N*	*(-0.04) N*	(-0.31)* S	(-0.27)* S	(-0.28)* S	**-0.49* L**	-0.42* M	-0.34* M
Barinel	*(0.05) N*	*(0.03) N*	*(0.004) N*	–			*(0.12) N*	*(0.08) N*	*(-0.04) N*	(-0.28)* S	(-0.26)* S	(-0.28)* S	-0.46* M	-0.41* M	-0.33* M
Ochai	*(-0.07) N*	*(-0.05) N*	*(0.04) N*	*(-0.12) N*	*(-0.08) N*	*(0.04) N*	–			-0.38* M	(-0.32)* S	(-0.26)* S	**-0.56* L**	-0.47* M	(-0.32)* S
DStar	(0.31)* S	(0.27)* S	(0.28)* S	(0.28)* S	(0.26)* S	(0.28)* S	0.38* M	(0.32)* S	(0.26)* S	–			(-0.28)* S	(-0.24)* S	*(-0.08) N*
OP2	**0.49* L**	0.42* M	0.34* M	0.46* M	0.41* M	0.33* M	**0.56* L**	0.47* M	(0.32)* S	**(0.28)* S**	(0.24)* S	*(0.08) N*	–		
Average-case Debugging Scenario															
Tarantula	–			*(-0.12) N*	*(-0.06) N*	*(-0.005) N*	*(0.10) N*	*(0.05) N*	*(-0.05) N*	(-0.23) S	(-0.22)* S	(-0.29)* S	-0.44* M	-0.42* M	-0.35* M
Barinel	*(0.12) N*	*(0.06) N*	*(0.005) N*	–			*(0.21) S*	*(0.10) N*	*(-0.04) N*	*(0.15) N*	(0.18) S	(0.28)* S	-0.38* M	-0.38* M	-0.35* M
Ochai	*(-0.10) N*	*(-0.05) N*	*(0.05) N*	(-0.21) S	*(-0.10) N*	*(0.04) N*	–			(-0.31)* S	(-0.27)* S	(-0.26)* S	**-0.50* L**	-0.45* M	(-0.34)* S
DStar	(0.23) S	(0.22)* S	(0.29)* S	*(-0.15) N*	(-0.18) S	(-0.28)* S	(0.31)* S	(0.27)* S	(0.26)* S	–			(-0.28) S	(-0.26) S	*(-0.09) N*
OP2	0.44* M	0.42* M	0.35* M	0.38* M	0.38* M	0.35* M	**0.50* L**	0.45* M	(0.34)* S	(0.28) S	(0.26) S	*(0.09) N*	–		
Worst-case Debugging Scenario															
Tarantula	–			*(-0.14) N*	*(-0.08) N*	*(-0.007) N*	*(0.12) N*	*(0.06) N*	*(-0.04) N*	(-0.22) S	(-0.21) S	(-0.29)* S	-0.43* M	-0.38* M	-0.34* M
Barinel	*(0.14) N*	*(0.08) N*	*(0.007) N*	–			(0.24) S	*(0.14) N*	*(-0.03) N*	*(-0.13) N*	(-0.16) S	(-0.28)* S	-0.35* M	-0.33* M	-0.34* M
Ochai	*(-0.12) N*	*(-0.06) N*	*(0.04) N*	(-0.24) S	*(-0.14) N*	*(0.03) N*	–			(-0.31)* S	(-0.27)* S	(-0.26)* S	**-0.50* L**	-0.43* M	(-0.33)* S
DStar	(0.22) S	(0.21) S	(0.29)* S	*(0.13) N*	(0.16) S	(0.28)* S	0.38* M	(0.32)* S	(0.26)* S	–			(-0.29) S	(-0.23) S	*(-0.08) N*
OP2	0.43* M	0.38* M	0.34* M	0.35* M	0.33* M	0.34* M	**0.50* L**	0.43* M	(0.33)* S	(0.29) S	(0.23) S	*(0.08) N*	–		

"*" Indicates that the difference between the top-k distribution is statistically significant at 5% level, i.e. $p<0.05$. "(negligible(N)/small(S)/medium(M)/large(L))" denote the category of the effect size.

The "-" symbol means that the techniques in the column performs better than those in the row. For example, comparison on Tarantula (column) with Barinel (row) that have value *(-0.05) N*, meaning Tarantula is the winner without significant differences and negligible effect size

Table 21 Improvement on SBFL Techniques

	[∞,201]→[200,11]	[∞,201]→ [10,6]	[∞,201]→ [5,1]	[200,11]→ [10,6]	[200,11]→ [5,1]	[10,6]→ [5,1]	Total
	Tarantula						
Barinel	0	0	0	0	0	2	2
Ochiai	35	0	0	6	1	2	44
DStar	85	4	5	12	12	5	123
O^P	86	4	5	17	111	11	142
	Barinel						
Tarantula	0	0	0	0	0	0	0
Ochiai	35	0	0	6	1	2	44
DStar	85	4	5	12	12	5	123
O^P	86	4	5	17	111	11	142
	Ochiai						
Tarantula	8	0	0	2	5	4	19
Barinel	8	0	0	2	5	6	21
DStar	56	1	5	11	14	5	92
O^P	57	1	5	16	21	11	111
	DStar						
Tarantula	11	1	0	2	2	3	17
Barinel	11	1	0	2	2	5	19
Ochiai	4	1	0	1	1	0	7
O^P	1	0	0	5	7	6	19
	O^P						
Tarantula	11	1	0	3	2	3	20
Barinel	11	1	0	3	2	5	22
Ochiai	6	1	0	3	2	5	10
DStar	2	0	0	1	0	0	3

Table 22 Improvement on Average-case Debugging Scenario

	[∞,201]→ [200,11]	[∞,201]→ [10,6]	[∞,201]→ [5,1]	[200,11]→ [10,6]	[200,11]→ [5,1]	[10,6]→ [5,1]	Total
	Tarantula						
Barinel	0	0	0	0	0	2	2
Ochiai	19	0	0	6	1	0	26
DStar	47	3	1	5	6	1	63
O^P	47	3	1	9	9	2	71
	Barinel						
Tarantula	0	0	0	0	0	0	0
Ochiai	19	0	0	6	1	0	26
DStar	47	3	1	5	6	1	63
O^P	47	3	1	9	9	2	71
	Ochiai						
Tarantula	3	0	0	1	4	0	8
Barinel	3	0	0	1	4	2	10
DStar	32	0	1	2	7	2	44
O^P	32	0	1	7	11	3	54
	DStar						
Tarantula	5	0	0	2	2	0	9
Barinel	5	0	0	2	2	2	11
Ochiai	3	0	0	0	1	0	4
O^P	0	0	0	5	4	1	10
	O^P						
Tarantula	6	0	0	2	1	0	9
Barinel	6	0	0	2	1	2	11
Ochiai	4	0	0	1	1	0	6
DStar	1	0	0	1	0	0	2

Table 23 Improvement on Worst-case Debugging Scenario

	[∞,201]→ [200,11]	[∞,201]→ [10,6]	[∞,201]→ [5,1]	[200,11]→ [10,6]	[200,11]→ [5,1]	[10,6]→ [5,1]	Total
Tarantula							
Barinel	0	0	0	0	0	2	2
Ochiai	13	0	0	4	1	0	18
DStar	36	2	1	3	5	0	47
O^P	36	2	1	5	8	1	53
Barinel							
Tarantula	0	0	0	0	0	0	0
Ochiai	13	0	0	4	1	0	18
DStar	36	2	1	3	5	0	47
O^P	36	2	1	5	8	1	53
Ochiai							
Tarantula	3	0	0	0	3	0	6
Barinel	3	0	0	0	3	2	8
DStar	26	0	1	1	6	1	35
O^P	26	0	1	3	9	2	41
DStar							
Tarantula	5	0	0	1	1	0	7
Barinel	5	0	0	1	1	2	9
Ochiai	3	0	0	0	1	0	4
O^P	0	0	0	2	3	1	6
O^P							
Tarantula	6	0	0	2	1	0	9
Barinel	6	0	0	2	1	2	11
Ochiai	4	0	0	1	1	0	6
DStar	1	0	0	1	0	0	2

Table 24 Prior Results Comparison on Average-case Scenario

No.	Previous comparisons on Java or C artificial faults	Defects4J Results 2017a		BugsInPy Results	
	Winner > Loser	Agree?	Effect size	Agree?	Effect size
1.	Ochiai > Tarantula*	*(insig.)*	-0.01 (N)	*(insig.)*	0.04 (N)
2.	Barinel > Ochiai (Abreu et al. 2009b)	*(insig.)*	0.01 (N)	*(insig.)*	-0.04 (N)
3.	Barinel > Tarantula (Abreu et al. 2009b)	*(insig.)*	-0.0002 (N)	*(insig.)*	0.001 (N)
4.	O^P > Ochiai (Naish et al. 2011a)	*(insig.)*	0.03 (N)	**no**	0.11 (N)
5.	O^P > Tarantula†	*(insig.)*	0.02 (N)	**no**	0.16 (S)
6.	DStar > Ochiai‡	*(insig.)*	0.002 (N)	**no**	0.11 (N)
7.	DStar > Tarantula**	*(insig.)*	-0.001 (N)	**no**	0.15 (S)

*(Le et al. 2013; Le et al. 2015b; Naish et al. 2011a; Wong et al. 2016; Xuan and Monperrus 2014b)

‡(Naish et al. 2011a; Moon et al. 2014) ‡(Le et al. 2015b; Wong et al. 2016)

**(Le et al. 2015b; Wong et al. 2016; Ju et al. 2014)

[1]Whether the results *agrees* indicates p-value: **p<0.05**, *($p \geq 0.05$)*.

[1]Cliff's d indicates effect size: large (L), medium (M), small (S), negligible (N)

Table 25 Prior Results Comparison on Worst-case Scenario

No.	Previous comparisons on Java or C artificial faults	Defects4J Results 2017a		BugsInPy Results	
	Winner> Loser	Agree?	Effect size	Agree?	Effect size
1.	Ochiai > Tarantula*	*(insig.)*	-0.005 (N)	*(insig.)*	0.02 (N)
2.	Barinel > Ochiai (Abreu et al. 2009b)	*(insig.)*	0.005 (N)	*(insig.)*	-0.02 (N)
3.	Barinel > Tarantula (Abreu et al. 2009b)	*(insig.)*	$6.4E-6$ (N)	*(insig.)*	0.0009 (N)
4.	O^P > Ochiai (Naish et al. 2011a)	*(insig.)*	0.03 (N)	*(insig.)*	0.04 (N)
5.	O^P > Tarantula†	*(insig.)*	0.02 (N)	*(insig.)*	0.06 (N)
6.	DStar > Ochiai‡	*(insig.)*	-0.0008 (N)	*(insig.)*	0.04 (N)
7.	DStar > Tarantula **	*(insig.)*	-0.005 (N)	*(insig.)*	0.07 (N)

*(Le et al. 2013; Le et al. 2015b; Naish et al. 2011a; Wong et al. 2016; Xuan and Monperrus 2014b)

†(Naish et al. 2011a; Moon et al. 2014) ‡(Le et al. 2015b; Wong et al. 2016)

**(Le et al. 2015b; Wong et al. 2016; Ju et al. 2014)

Whether the results *agrees* indicates p-value: **p<0.05**, *($p \geq 0.05$)*.

Cliff's d indicates effect size: large (L), medium (M), small (S), negligible (N)

References

Abreu R, Van Gemund AJ (2009) A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In: SARA, vol 9, Citeseer, pp 2–9

Abreu R, Zoeteweij P, Golsteijn R, Van Gemund ArjanJC (2009a) A practical evaluation of spectrum-based fault localization. J Syst Softw 82(11):1780–1792

Abreu R, Zoeteweij P, van Gemund AJC (2007) On the accuracy of spectrum-based fault localization. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, IEEE Computer Society, USA, TAICPART-MUTATION '07, pp 89–98

Abreu R, Zoeteweij P, Van Gemund AJ (2006) An evaluation of similarity coefficients for software fault localization. In: 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06), IEEE, pp 39–46

Abreu R, Zoeteweij P, Van Gemund AJ (2009b) Spectrum-based multiple fault localization. In: 2009 IEEE/ACM International Conference on Automated Software Engineering, pp 88–99, IEEE

Ali S, Andrews JH, Dhandapani T, Wang W (2009) Evaluating the accuracy of fault localization techniques. In: 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE, PP 76–87

Baah GK, Podgurski A, Harrold MJ (2010) The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans Softw Eng 36(4):528–545

Bouillon P, Krinke J, Meyer N, Steimann F (2007) Ezunit: A framework for associating failed unit tests with potential programming errors. In: International Conference on Extreme Programming and Agile Processes in Software Engineering, Springer, PP 101–104

Briand LC, Labiche Y, Liu X (2007) Using machine learning to support debugging with tarantula. In: The 18th IEEE International Symposium on Software Reliability (ISSRE'07), pp 137–146

Cantor AB (1996) Sample-size calculations for cohen's kappa. Psychol Methods 1(2):150

Chaki S, Groce A, Strichman O (2004) Explaining abstract counterexamples. In: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, pp 73–82

Chen D, Stolee KT, Menzies T (2019) Replication can improve prior results: A github study of pull request acceptance. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp 179–190, IEEE

Cifuentes C, Hoermann C, Keynes N, Li L, Long S, Mealy E, Mounteney M, Scholz B (2009) Begbunch: Benchmarking for c bug detection tools. In: Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), pp 16–20

Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol Bull 114(3):494

D'Agostino R, Pearson ES (1973) Tests for departure from normality. Empirical results for the distributions of b^2 and $\sqrt{b1}$. Biometrika 60(3):613–622

D'Agostino RB (1971) An omnibus test of normality for moderate and large sample sizes. Biometrika 58(34):1–348

Debroy V, Wong WE, Xu X, Choi B (2010) A grouping-based strategy to improve the effectiveness of fault localization techniques. In: 2010 10th International Conference on Quality Software, IEEE, pp 13–22

DeVellis RF (2005) Inter-rater reliability. encyclopedia of social measurement. Elsevier Academic Press, Oxford

Durieux T, Abreu R (2019) Critical review of bugswarm for fault localization and program repair. arXiv preprint arXiv: 1905.09375

Ghanbari A, Benton S, Zhang L (2019) Practical program repair via bytecode mutation. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 19–30

Gouveia C, Campos J, Abreu R (2013) Using html5 visualizations in software fault localization. In: 2013 First IEEE Working Conference on Software Visualization (VISSOFT), pp 1–10.,

Hao D, Zhang L, Zhang L, Sun J, Mei H (2009) Vida: Visual interactive debugging. In: 2009 IEEE 31st International Conference on Software Engineering, IEEE, pp 583–586

He H, Ren J, Zhao G, He H (2020) Enhancing spectrum-based fault localization using fault influence propagation. IEEE Access 8:18497–18513

Horváth F, Beszédes A, Vancsics B, Balogh G, Vidács L, Gyimóthy T (2020) Experiments with interactive fault localization using simulated and real users. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 290–300

Hutchins M, Foster H, Goradia T, Ostrand T (1994) Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In: Proceedings of 16th International conference on Software engineering, IEEE, pp 191–200

Jiang J, Xiong Y, Zhang H, Gao Q, Chen X (2018) Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 298–309

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp 273–282

Jones JA, Harrold MJ, Stasko JT (2001) Visualization for fault localization. In: Proceedings of ICSE 2001 Workshop on Software Visualization, Citeseer

Ju X, Jiang S, Chen X, Wang X, Zhang Y, Cao H (2014) Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices. J Syst Softw 90:3–17

Just R (2014) The major mutation framework: Efficient and scalable mutation analysis for java. In: Proceedings of the 2014 international symposium on software testing and analysis, pp 433–436
Just R, Jalali D, Ernst MD (2014a) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2014, pp 437–440, https://doi.org/10.1145/2610384.2628055
Just R, Jalali D, Ernst MD (2014b) Defects4J: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp 437–440
Just R, Parnin C, Drosos I, Ernst MD (2018) Comparing developer-provided to user-provided tests for fault localization and automated program repair. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2018, pp 287–297. https://doi.org/10.1145/3213846.3213870
Kim J, Lee E (2014) Empirical evaluation of existing algorithms of spectrum based fault localization. In: The International Conference on Information Networking 2014 (ICOIN2014), IEEE, pp 346–351
Kitchenham B (2008) The role of replications in empirical software engineering—word of warning. Empir Softw Eng 13(2):219–221
Koca F, Sözer H, Abreu R (2013) Spectrum-based fault localization for diagnosing concurrency faults. In: IFIP International Conference on Testing Software and Systems, Springer, pp 239–254
Kochhar PS, Xia X, Lo D, Li S (2016) Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp 165–176
Könighofer R, Bloem R (2011) Automated error localization and correction for imperative programs. In: 2011 Formal Methods in Computer-Aided Design (FMCAD), IEEE, pp 91–100
Le TB, Thung F, Lo D (2013) Theory and practice, do they match? a case with spectrum-based fault localization. In: 2013 IEEE International Conference on Software Maintenance, pp 380–383.
Le T-DB, Lo D, Li M (2015a) Constrained feature selection for localizing faults. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 501–505
Le T-DB, Lo D, Thung F (2015b) Should i follow this fault localization tool's output? Empirical Softw. Engg. 20(5):1237–1274. https://doi.org/10.1007/s10664-014-9349-1
Le T-DB, Thung F, Lo D (2013) Theory and practice, do they match? a case with spectrum-based fault localization. In: 2013 IEEE International Conference on Software Maintenance, IEEE, pp 380–383
Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The manybugs and introclass benchmarks for automated repair of c programs. IEEE Trans Softw Eng 41(12):1236–1256
Lindsay RM, Ehrenberg AS (1993) The design of replicated studies. The American Statistician 47(3):217–228
Lo D, Jiang L, Budi A et al (2010) Comprehensive evaluation of association measures for fault localization. In: 2010 IEEE International Conference on Software Maintenance, IEEE, pp 1–10
Long F, Rinard M (2016) An analysis of the search spaces for generate and validate patch generation systems. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, pp 702–713
Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y (2005) Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the evaluation of software defect detection tools, vol 5
Lucia, Lo D, Xia X (2014) Fusion fault localizers. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp 127–138
Martinez M, Monperrus M (2015) Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng 20(1):176–205
Moon S, Kim Y, Kim M, Yoo S (2014) Ask the mutants: Mutating faulty programs for fault localization. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, IEEE, pp 153–162
Naish L, Lee HJ, Ramamohanarao K (2011a) A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. 20(3)
Naish L, Lee HJ, Ramamohanarao K (2011b) A model for spectra-based software diagnosis. ACM Transactions on software engineering and methodology (TOSEM) 20(3):1–32
Pan K, Kim S, Whitehead EJ (2009) Toward an understanding of bug fix patterns. Empirical Softw. Engg. 14(3):286–315. https://doi.org/10.1007/s10664-008-9077-5
Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 international symposium on software testing and analysis, pp 199–209
Patra J, Pradel M (2021) Semantic bug seeding: a learning-based approach for creating realistic bugs. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 906–918

Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B (2017) Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp 609–620.

Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B (2017) Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, pp 609–620

Planning S (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology

Rayson P, Berridge D, Francis B (2004) Extending the cochran rule for the comparison of word frequencies between corpora. In: 7th International Conference on Statistical analysis of textual data (JADT 2004), pp 926–936

Ren L, Shan S, xu X, Liu (2020) Starin: An approach to predict the popularity of github repository, pp 258–273. https://doi.org/10.1007/978-981-15-7984-4_20

Renieres M, Reiss SP (2003) Fault localization with nearest neighbor queries. In: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings., IEEE, pp 30–39

Romano J, Kromrey JD, Coraggio J, Skowronek J, Devine L (2006) Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. In: annual meeting of the Southern Association for Institutional Research, Citeseer, pp 1–51

Ruthruff JR, Burnett M, Rothermel G (2005) An empirical study of fault localization for end-user programmers. In: Proceedings of the 27th International Conference on Software Engineering, pp 352–361

Saha RK, Lyu Y, Lam W, Yoshida H, Prasad MR (2018) Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp 10–13

Santos A, Vegas S, Uyaguari F, Dieste O, Turhan B, Juristo N (2020) Increasing validity through replication: an illustrative tdd case. arXiv preprint arXiv: 2004.05335

Shull FJ, Carver JC, Vegas S, Juristo N (2008) The role of replications in empirical software engineering. Empir Softw Eng 13(2):211–218

Sobreira V, Durieux T, Madeiral F, Monperrus M, de Almeida Maia M (2018) Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 130–140

Sohn J, Yoo S (2017) Fluccs: Using code and change metrics to improve fault localization. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 273–283

Steimann F, Frenkel M, Abreu R (2013) Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp 314–324

Tallarida RJ, Murray RB (1987) Chi-square test. In: Manual of pharmacologic calculations, Springer, pp 140–142

Tomassi DA, Dmeiri N, Wang Y, Bhowmick A, Liu Y-C, Devanbu PT, Vasilescu B, Rubio-González C (2019) Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 339–349

Tufano M, Kimko J, Wang S, Watson C, Bavota G, Di Penta M, Poshyvanyk D (2020) Deepmutation: A neural mutation tool. In: 42nd ACM/IEEE International Conference on Software Engineering: Companion, ICSE-Companion 2020, Institute of Electrical and Electronics Engineers Inc., pp 29–33

Vancsics B, Szatmári A, Beszédes A (2020) Relationship between the effectiveness of spectrum-based fault localization and bug-fix types in javascript programs. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 308–319

Vessey I (1985) Expertise in debugging computer programs: A process analysis. International Journal of Man-Machine Studies 23(5):459–494

Wen M, Chen J, Wu R, Hao D, Cheung S-C (2018) Context-aware patch generation for better automated program repair. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, pp 1–11

Widyasari R, Sim SQ, Lok C, Qi H, Phan J, Tay Q, Tan C, Wee F, Tan JE, Yieh Y, et al (2020) 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, pp 1556–1560

Wilcoxon F (1992) Individual comparisons by ranking methods. In: Breakthroughs in statistics, Springer, pp 196–202

Wong E, Wei T, Qi Y, Zhao L (2008) A crosstab-based statistical method for effective fault localization. In: 2008 1st international conference on software testing, verification, and validation, IEEE, pp 42–51

Wong WE, Debroy V, Gao R, Li Y (2013) The dstar method for effective software fault localization. IEEE Trans Reliab 63(1):290–308

Wong WE, Debroy V, Golden R, Xu X, Thuraisingham B (2011) Effective software fault localization using an rbf neural network. IEEE Trans Reliab 61(1):149–169

Wong WE, Debroy V, Surampudi A, Kim H, Siok MF (2010) Recent catastrophic accidents: Investigating how software was responsible. In: 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement, IEEE, pp 14–22

Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Trans Softw Eng 42(8):707–740

Wright CS, Zia TA (2011) A quantitative analysis into the economics of correcting software bugs. In: Computational Intelligence in Security for Information Systems, Springer, pp 198–205

Xia X, Bao L, Lo D, Li S (2016) “automated debugging considered harmful” considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 267–278, IEEE

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

Xie X, Liu Z, Song S, Chen Z, Xuan J, Xu B (2016) Revisit of automatic debugging via human focus-tracking analysis. In: Proceedings of the 38th International Conference on Software Engineering, pp 808–819

Xuan J, Monperrus M (2014a) Learning to combine multiple ranking metrics for fault localization. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp 191–200, IEEE

Xuan J, Monperrus M (2014b) Test case purification for improving fault localization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 52–63

Zhang M, Li X, Zhang L, Khurshid S (2017) Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 261–272

Zou D, Liang J, Xiong Y, Ernst MD, Zhang L (2019) An empirical study of fault localization families and their combinations. IEEE Trans Softw Eng 47(2):332–347

Affiliations

Ratnadira Widyasari[1] · Gede Artha Azriadi Prana[1] · Stefanus Agus Haryono[1] · Shaowei Wang[2] · David Lo[1]

Gede Artha Azriadi Prana
arthaprana.2016@smu.edu.sg

Stefanus Agus Haryono
stefanusah@smu.edu.sg

Shaowei Wang
shaowei.wang@umanitoba.ca

David Lo
davidlo@smu.edu.sg

[1] School of Computing and Information Systems, Singapore Management University, 80 Stamford Rd, Stamford, 178902, Singapore

[2] Department of Computer Science, University of Manitoba, Winnipeg, Canada