Historical Spectrum based Fault Localization

Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han and Shing-Chi Cheung

Abstract—Spectrum-based fault localization (SBFL) techniques are widely studied and have been evaluated to be effective in locating faults. Recent studies also showed that developers from industry value automated SBFL techniques. However, their effectiveness is still limited by two main reasons. First, the test coverage information leveraged to construct the spectrum does not reflect the root cause directly. Second, SBFL suffers from the tie issue so that the buggy code entities can not be well differentiated from non-buggy ones. To address these challenges, we propose to leverage the information of version histories in fault localization based on the following two intuitions. First, version histories record how bugs are introduced to software projects and this information reflects the root cause of bugs directly. Second, the evolution histories of code can help differentiate those suspicious code entities ranked in tie by SBFL. Our intuitions are also inspired by the observations on debugging practices from large open source projects and industry.

Based on the intuitions, we propose a novel technique HSFL (historical spectrum based fault localization). Specifically, HSFL identifies bug-inducing commits from the version history in the first step. It then constructs historical spectrum (denoted as Histrum) based on bug-inducing commits, which is another dimension of spectrum orthogonal to the coverage based spectrum used in SBFL. HSFL finally ranks the suspicious code elements based on our proposed Histrum and the conventional spectrum. HSFL outperforms the state-of-the-art SBFL techniques significantly on the Defects4J benchmark. Specifically, it locates and ranks the buggy statement at Top-1 for 77.8% more bugs as compared with SBFL, and 33.9% more bugs at Top-5. Besides, for the metrics MAP and MRR, HSFL achieves an average improvement of 28.3% and 40.8% over all bugs, respectively. Moreover, HSFL can also outperform other six families of fault localization techniques, and our proposed Histrum model can be integrated with different families of techniques and boost their performance.

Index Terms—Fault Localization, Version Histories, Bug-Inducing Commits

1 Introduction

Software debugging is time-consuming and labor-intensive. According to a recent study [1], this process costs nearly 50% of developers' time and efforts. To mitigate the problem, automated debugging attracts much attention, where fault localization (FL) has been recognized as an important step [2], [3], [4]. Xia *et al.* [5] recently conducted an empirical study and found that FL can actually help developers save debugging time in practice. Another recent study also revealed that developers from industry value automated FL techniques [6]. Specifically, more than 97% of the developers consider it essential or worthwhile to leverage automated FL techniques. Besides, FL techniques are essential for automated program repair (APR) techniques (e.g., [7], [8], [9], [10]), which rely mostly on FL to generate a fault space at ***statement*** granularity. The effectiveness of FL greatly affects the performance of APR [7], [10]. Therefore, there are strong demands for better FL to improve APR's performance. As a result, various recent efforts (e.g., [11], [12], [13]) have been made to advance FL.

Spectrum-based fault localization (SBFL) is a major category of FL techniques (e.g., [11], [12], [14], [15], [16]). It constructs an *coverage based spectrum* by running the passing and failing tests, and then uses the spectrum to compute the suspicious score for each code entity (e.g., statement or method). It assumes that *the code entities covered by more failing tests but fewer passing tests are more likely to be buggy*. Due to its effectiveness, SBFL has been used by developers for debugging in practice [15], [17], [18].

Even though successes in locating faults by SBFL have been demonstrated, the effectiveness of SBFL is still compromised due to two main reasons [19], [20], [21], [22]. First, SBFL is based only on test coverage information. Although test coverage has been leveraged to approximate a bug's root cause, it does not pinpoint the root cause of a bug directly [19], [20]. Second, SBFL widely suffers from the tie issue [21], [22]. One typical tie example is that the statements in the same program block have the same suspicious score, since they are equally covered by tests. In such cases, the buggy code entities cannot be differentiated from the non-buggy ones in the same program block.

We propose to overcome these limitations by taking a novel perspective from project version histories. First, a bug's root cause can be directly reflected in the version

- *Ming Wen is with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China. E-mail: mwenaa@hust.edu.cn*
- *Yongqiang Tian and Shing-Chi Cheung are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China. E-mail: {ytianas, scc}@cse.ust.hk.*
- *Junjie Chen is with the College of Intelligence and Computing, Tianjin University, Tianjin, China. E-mail: junjiechen@tju.edu.cn*
- *Rongxin Wu is with the Department of Cyber Space Security, Xiamen University, Xiamen, China. E-mail: wurongxin@xmu.edu.cn*
- *Dan Hao is with the Key Laboratory of High Confidence Software Technologies and Institute of Software, EECS, Peking University, Beijing, China. E-mail: haodan@pku.edu.cn.*
- *Shi Han is with Microsoft Research Asia, Beijing, China. E-mail: shihan@microsoft.com*

Manuscript received xxx, 2018; revised xxx, 2018.

history. A bug was introduced into a software project by either the initial code commit or subsequent code commits when the software evolves [23]. In particular, the commit introducing a bug is called a *bug-inducing commit* [23], [24], and the associated *bug-revealing tests* start to fail after the bug-inducing commit is adopted [25]. Intuitively, identifying the bug-inducing commit will help locate the root cause (i.e., those buggy statements). Second, the version histories of code entities can help differentiate those suspicious code entities, since different code entities (even in the same block) could have different evolution histories (i.e., modified by different commits). Therefore, it greatly increases the chances to break the tie issue in SBFL.

Our intuition is also inspired by the observations from the debugging practices of popular projects. For example, we observed that developers in project GCC often try to locate the bug-inducing commits first when they work on a reported bug. Comments such as *"Confirmed, started with `r239357`"* [26] are often left in bug reports. Similar practices are also observed among other projects. For instance, when debugging `SOLR-2606` [27], a developer located the corresponding bug-inducing commit and left a message *"I'm fairly certain this is caused by the enhancements made in `SOLR-1297` to add sorting functions"*. Such message reveals that the bug was caused by the code committed to implement enhancements requested by issue `SOLR-1297`. After obtaining such knowledge, developers located and resolved this bug quickly. Our observations are also confirmed by the feedbacks from industry (see Section 2.1).

Based on the above intuition and observations, we propose **H**istorical **S**pectrum-based **F**ault **L**ocalization (HSFL) in this study, which leverages the information of version histories in fault localization. HSFL first identifies the bug-inducing commit in the version history for each bug-revealing test. In other words, it finds the first commit in the version history from which the bug-revealing test cases start to fail. However, code commits are usually tangled [28]. They are often large in size, but only a small part of the code elements introduced in these commits are related to the fault. Therefore, it is very challenging to distill the root causes from the bug-inducing commits. Besides, the time gap between when the bug-inducing commit is checked in and the *target version* (i.e., the version subject to fault localization) might be large, and lots of commits can be adopted during the period. Therefore, it brings the challenge to trace their evolutions to the target version for fault localization.

To address these challenges, HSFL builds a **His**torical Spec**trum** (denoted as *Histrum*) for each suspicious code entity introduced in the bug-inducing commits. The Histrum traces the evolutions for each suspicious code element from the inducing version (i.e., the version after the bug-inducing commit is adopted) to the target version via history slicing [29]. Specifically, it leverages the information of non-inducing commits (i.e., those commits do not introduce the bug) in the version histories to filter out those noises in the bug-inducing commits. HSFL then computes the suspicious score for each code entity based on the Histrum via leveraging those techniques proposed for SBFL (e.g., Ochiai [30]), where a bug-inducing commit and a non-inducing commit are analogous to a failing test and a passing test, respectively. The key insight of our approach is that *those code entities modified by more bug-inducing commits but fewer non-inducing commits are more likely to be the root cause of the bug*. HSFL further examines whether those suspicious code entities evolved from bug-inducing commits have been executed by bug-revealing tests in the target version to filter out potential noises for better fault localization.

We evaluated HSFL on 357 real bugs from the DEFECTS4J [31] benchmark. Specifically, we applied HSFL to each of the bugs and located the faulty code entities at the ***statement*** level, which is the granularity widely adopted by existing SBFL techniques (i.e., [11], [12], [14], [15], [16]), and required by automated program repair techniques to generate the fault space [7], [8]. We compared the results generated by HSFL with the state-of-the-art SBFL techniques [11]. Our evaluation results show that HSFL can significantly improve SBFL's performance. For example, HSFL locates and ranks the buggy statement at Top-1 for 77.8% more bugs compared with SBFL, and 33.9% more bugs for Top-5. HSFL also performs significantly better than SBFL for the evaluation metrics MAP and MRR, with an improvement of 28.3% and 40.8% respectively. We also applied other SBFL techniques [32], [33], [34], [35] in the Histrum model, and found that HSFL also achieves significant better performances using other techniques such as Tarantula [32], Op2 [34], Barinel [35] and DStar [33]. We also compared HSFL with other six families of fault localization techniques, including *mutation-based, slicing-based, stack trace-based, predicate switching-based, hybrid-based and learning-to-rank-based* techniques. Our extensive evaluations show that our proposed approach can not only outperform existing baselines from different families, but also it can boost the performance of existing techniques. Moreover, the results generated by HSFL can significantly improve the performance of the state-of-the-art search-based APR techniques. Specifically, the first correct patch can be searched 3.02 times faster via leveraging the fault space generated by HSFL compared with that generated by SBFL.

In summary, our major contributions are as follows.

- ***Observation***: We made observations from both open source communities and industry that version histories contain useful debugging information and bug-inducing commits are helpful to understand and locate software bugs.
- ***Originality:*** We are the first to leverage bug-inducing commits in facilitating fault localization. Specifically, we propose a novel model called historical spectrum, which builds a spectrum along the version histories in orthogonal to the conventional coverage based spectrum.
- ***Implementation:*** We implement the proposed idea as a fault localization technique, HSFL, which leverages existing techniques (e.g., Ochiai) to rank all suspicious code entities based on the historical spectrum.
- ***Evaluation:*** We evaluate HSFL on the DEFECTS4J benchmark and compare it extensively with the state-of-the-art FL techniques from seven different families. The results show that our proposed approach can not only outperform existing baselines from different families, but also it can boost the performance of existing techniques. More importantly, it can also significantly boost the performance of the state-of-the-art automated program repair techniques to find the correct patches.

The rest of the paper is structured as follows. Section

2 presents the motivation and challenges of this work. In Section 3, we present our approach in detail. Experimental setup is introduced in Section 4, and Section 5 presents the experimental results which demonstrate the usefulness of HSFL. In Section 6, we discuss several points related to the performance of our proposed tool. Section 7 discusses the related works and Section 8 concludes this work.

2 Motivation and Challenges

In this section, we present our observations and the motivation of this study together with the potential challenges.

2.1 Debugging Practice

Version control systems are widely used to manage software evolution. The version histories record how faults are introduced into the software. Such information is important and usually leveraged by developers in debugging. We observe that developers of open source projects often discuss about the information of version histories, especially the *bug-inducing commits*, in bug reports. A bug-inducing commit is the one that introduces a bug [23]. It causes some tests, called *bug-revealing tests*, start to fail until the bug is fixed. After the bug-inducing commit is submitted, the bug-revealing test cases start to fail. Specifically, We found that substantial bug reports, 821 and 1733 bug reports from GCC and Apache projects respectively, contain discussions about bug-inducing commits by searching the keywords of “started with”, “caused by” and “introduced by” among the bug reports tracked in the associated bug-tracking system. We selected these three keywords since by sampling a small set of bug reports randomly, we observed that developers in our selected projects mostly used these keywords to deliver the information of bug-inducing commits. Examples of these bug reports [26], [27] are shown in Section 1. We also observed that the root cause of a bug is frequently correlated with its bug-inducing commits. For instance, we found that for 78.9% of those bugs, at least one statement in their bug-fixing commits have been modified by the associated bug-inducing commits. Inspired by this, we further surveyed developers from industry to understand the role of the information of version histories and bug-inducing commits in general practices of debugging and fault localization.

To understand current debugging practices in industry, we designed an online survey following the methodology of an existing work [36] and distributed it to the developers at Microsoft. Before distributing our survey, we conducted pilot interviews with 2 experienced engineers at Microsoft to discuss whether our designed questions and answers are appropriate. Based on the collected comments and feedback, we refined our survey questions in order to ensure that our designed questions are relevant and clear[1]. For instance, we used “traces” and “running log” in two options in the first question at the first beginning. However, the involved engineers suggested that these two terms are hard to differentiate in practice and thus might not be good answers. We modified these answers accordingly, and then distributed our survey through the discussion groups at Microsoft, which cover nearly 1,500 developers from multiple products. The survey was posted for a week. We set the time to one week since we observe that there is no increasing number of feedback received after one week. Finally, 109 valid responses were received, and we kept the 103 responses submitted from those developers who have at least 2 years of industrial software development experiences in our analysis. We consider these developers as experienced ones in terms of debugging. The response rate is hard to measure since this survey was posted on discussion groups, which is not mandatory. Besides, it is hard to measure exactly how many developers have viewed the post during the one week.

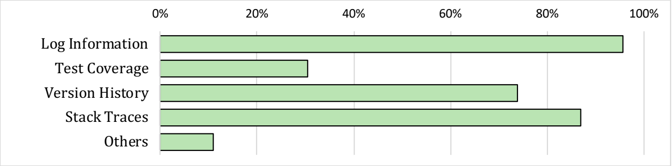


Fig. 1: What Information Have You Ever Used for Debugging?

We are first curious about what information is useful and has been leveraged by developers in debugging in practice. Five options are provided, which are *log information*, *test coverage*, *stack traces*, *version histories* and *others*. The design of these five options is motivated by the findings of existing studies [23], [37], [38], [39], [40] and refined after the pilot interviews. Figure 1 shows the results, and we can see that among the 103 responses analyzed, 75.7% (78/103) of the developers have ever leveraged version histories for debugging, and the ratio is comparable with the ratios of stack traces and log information. This demonstrates the usefulness of version histories in debugging. We are then curious to know what specific information of version histories that these 78 developers think is useful for debugging, and Figure 2 shows the results. Specifically, 93.6% (73/78) of them think bug-inducing commits are useful for debugging, and 74.4% of them (58/78) find that regression range (i.e., the range of commits between the last known good version to the first known bad version of the bug) is useful. These results show that the majority of developers (73/103) find bug-inducing commits providing useful debugging information. For those 73 developers, we further asked them in which ways have they leveraged such information for debugging in practice. Figure 3 shows the statistical results of the usages of bug-inducing commits by these developers. 95.9% of them (70/73) have leveraged bug-inducing commits to understand the root causes of the bugs and further locate the faults. However, we find that 74.3% (52/73) of these developers conduct the process of fault localization manually due to the lack of automated tool support. We also observe that a substantial of developers mention that they leveraged the built-in tool “git bisect” to search among version histories.

Since the conducted survey is not the major contributions of this study, we only disccused partial results in this section. Detailed survey results are available online.[2] Nev-

[1] The survey is available at https://www.wjx.cn/jq/19791453.aspx

[2] https://github.com/justinwm/HSFL/blob/master/survey.pdf

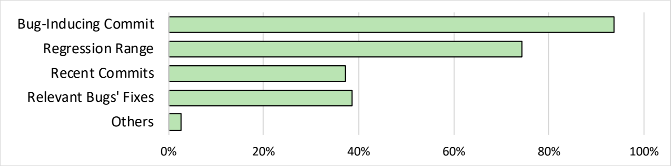


Fig. 2: What Information of Version Histories is Useful for Debugging?

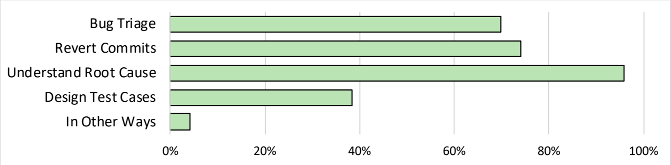


Fig. 3: In Which Ways Have You Ever Leveraged Bug-Inducing Commits?

ertheless, the above discussed results confirm our intuition and reveal the following three points. First, the information of version histories, especially the bug-inducing commits, is useful for developers to debug in practice. Second, bug-inducing commits contain rich information of the root causes of software bugs, which is helpful for fault localization. Third, the majority of developers lack automated tool supports to leverage such information.

As revealed by our survey, it is a common practice for developers to leverage "git bisect" to search for the information of bug-inducing commits when debugging for large-scale projects like LLVM and Lucene. Actually, we also observe that such practice can also be generalized to many open-source project communities. Specifically, we selected three large-scale open source projects: Lucene, LLVM and Accumulo. And then we searched their bug reports using the keyword "bisect". We observe that many bug reports (i.e., in total nearly 100 for the three projects) directly contain such information and deliver the message that developers actually adopted such a heuristic in practice to identify the inducing commits when debugging. Be noted that not all developers who adopted "git bisect" to identify bug-inducing commits will leave such messages on the associated bug reports. Several examples are selected as follows for each project:

For Lucene:

"git bisect blames commit 26d2ed7c4ddd2 on SOLR-10989" [3]
"According to git bisect, this was broken by SOLR-8728" [4]

For LLVM:

"bisect indicates that r146856 is the first bad commit (constexpr handling improvements.)" [5]
"A bisect points to r115374" [6]
"My bisect also pointed to r149641 is the first bad commit." [7]

For Accumulo:

"Using git bisect, found the breaking commit to be 659a33e8 as a part of ACCUMULO-4596" [8]
"git bisect revealed f599b46 to have introduced this problem (ACCUMULO-3929)." [9]

The above examples demonstrate that the practice of searching bisectly among version histories to look for bug-inducing commits is common and feasible, even for large-scale open-source projects. These examples also shed lights on the design of our approach in this study.

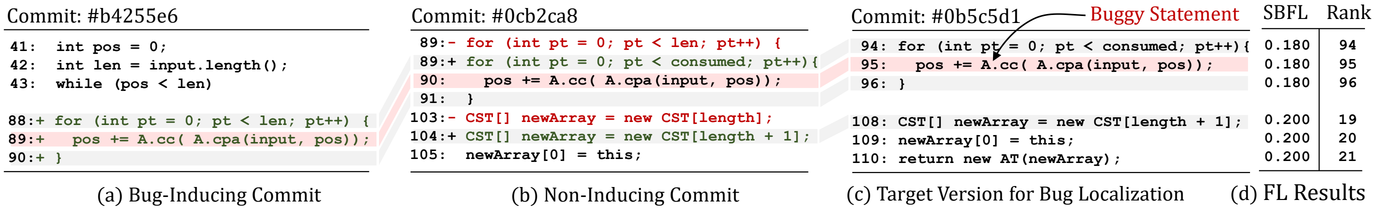
2.2 Motivating Example

The previous subsection presents our observations from both open source communities and industry, which motivates us to leverage the information of bug-inducing commits for automated fault localization. However, commits are often large in size and tangled with code modifications for multiple purposes [28]. For example, we investigated the identified bug-inducing commits for the Chart project from DEFECTS4J [31], and found their average size (i.e., number of modified statements) is 436.2 (with a median value of 165). However, the average size of the fixing patches of the corresponding bugs is 3.92 (with a median size of 2). Therefore, locating the buggy code entities in a bug-inducing commit is challenging.

We propose to build a historical spectrum along the version histories to address the challenge. Specifically, we leverage those commits, which are made after the bug-inducing commit but neither introduce nor fix the bug, to help pinpoint the buggy code entities. Those commits are referred as *non-inducing commits*. Figure 5 shows the concept of historical spectrum. Suppose v_t is the *target version* for fault localization, and c_i is the bug-inducing commit since the bug-revealing tests start to fail since version v_i after c_i is committed. Those commits made after c_i but neither introduce nor fix the bug are non-inducing commits (e.g., c_{i+1}). We build a historical spectrum by analyzing those code entities modified in the bug-inducing commits and non-inducing commits (i.e., those commits displayed in shadow as shown in Figure 5). Our key insight is that *those code entities modified by more bug-inducing commits but fewer non-inducing commits are more likely to be the root cause of the bug.*

Let us illustrate our insight using a concrete example shown in Figure 4, which is adapted from the bug `Lang 6` in the DEFECTS4J benchmark [31]. In this example, the target version for bug localization is `#0b5c5d1`, and the buggy statement is line 95. However, the suspicious value of the buggy statement reported by the state-of-the-art technique [11] using formula Ochiai [30] is 0.180. It is only ranked at 98^{th} in the suspicious statement list, and there are many ties (e.g., lines 94, 95, 96). These indicate that, conventional SBFL cannot effectively locate the fault. The bug-inducing commit of this bug is `#b4255e6`, and the bug-revealing tests start to fail after this commit is adopted in the Lang project. Intuitively, those statements introduced by this commit (i.e., statements 88 and 89 in Figure 4(a)) are more likely to be the root cause of this bug. Therefore, we should

[3] https://jira.apache.org/jira/browse/SOLR-11020
[4] https://jira.apache.org/jira/browse/SOLR-8788
[5] https://bugs.llvm.org/show_bug.cgi?id=11614
[6] https://bugs.llvm.org/show_bug.cgi?id=8284
[7] https://bugs.llvm.org/show_bug.cgi?id=12581
[8] https://jira.apache.org/jira/browse/ACCUMULO-4674
[9] https://jira.apache.org/jira/browse/ACCUMULO-3942



Commit: #b4255e6

```
41:  int pos = 0;
42:  int len = input.length();
43:  while (pos < len)

88:+ for (int pt = 0; pt < len; pt++) {
89:+   pos += A.cc( A.cpa(input, pos));
90:+ }
```

(a) Bug-Inducing Commit

Commit: #0cb2ca8

```
 89:- for (int pt = 0; pt < len; pt++) {
 89:+ for (int pt = 0; pt < consumed; pt++){
 90:    pos += A.cc( A.cpa(input, pos));
 91:  }
103:- CST[] newArray = new CST[length];
104:+ CST[] newArray = new CST[length + 1];
105:  newArray[0] = this;
```

(b) Non-Inducing Commit

Commit: #0b5c5d1

```
 94: for (int pt = 0; pt < consumed; pt++){
 95:   pos += A.cc( A.cpa(input, pos));
 96: }

108: CST[] newArray = new CST[length + 1];
109: newArray[0] = this;
110: return new AT(newArray);
```

(c) Target Version for Bug Localization

SBFL	Rank
0.180	94
0.180	95
0.180	96
0.200	19
0.200	20
0.200	21

(d) FL Results

Fig. 4: An Adapted Example from Bug Lang 6

increase the suspiciousness of statements 94 and 95 in the target version correspondingly (since statement 94 and 95 in #0b5c5d1 are evolved from statement 89 and 90 in #b4255e6, respectively). Meanwhile, we also observe another commit #0cb2ca8 as shown in Figure 4(b), which was made after the bug-inducing commit #b4255e6 but before the target version (commit #0b5c5d1). It changed statement 89 (i.e., evolved to statement 94 in the target version). However, this commit did not change the status of the bug-revealing tests. This indicates that statement 94 in the target version is less likely to be the root cause as compared with statement 95. Therefore, we can decrease the suspiciousness of statement 94. As a result, we can break the tie of lines 94, 95, and 96, which further confirms our intuition that the version histories can help relieve the tie issue. In this way, we can better locate the buggy statement (i.e., statement 95 in Figure 4(c)). The priority of other statements that are irrelevant to the bug in the bug-inducing commit #b4255e6 can be similarly lowered.

2.3 Challenges

Three major challenges hinder the process of leveraging version histories in fault localization.

1) ***Identifying bug-inducing commits precisely is difficult***. Whether we can identify the bug-inducing commits for all bug-revealing tests is unknown since some tests might be complex in their designs and cannot be successfully executed on previous versions. To address this challenge, we minimize the testing logic for each bug-revealing test before executing it to make it runnable on more previous versions (see Section 3.1).
2) ***Precisely tracking code evolution is challenging***. Precisely mapping code entities from the inducing version (i.e., the version after the bug-inducing commit is made) to the target version is challenging since the gap between these two versions might be large. As shown in the example in Figure 4, the buggy statement is line 89 at the inducing version while it evolves to line 95 at the target version. To resolve this challenge, we leverage history slicing [29] to track the evolutions of code entities from the inducing version to the target version (see Section 3.2).
3) ***Handling the noises of tangled commits is non-trivial***. commits are usually tangled with other irrelevant code modifications [28] and large in their sizes, and thus it is challenging to differentiate relevant statements from them. For example, we investigated the identified bug-inducing commits for the Chart project from Defects4J, and found their average size (i.e., the number of modified statements) is 436.2 (with a median of 165). However, the average size of the fixing patches for the corresponding bugs is 3.92 (with a median of 2). Those irrelevant statements might bring noises and thus decrease the performance of fault localization. To tackle this challenge, we apply those techniques designed for conventional SBFL (i.e., Ochiai [14] and Tarantula [32]) on the historical spectrum, to differentiate those buggy statements from other irrelevant ones that are modified in the bug-inducing commits. We also examine whether those suspicious code entities evolved from bug-inducing commits have been executed by bug-revealing tests in the target version in order to further reduce noises in the historical spectrum (see Section 3.3).

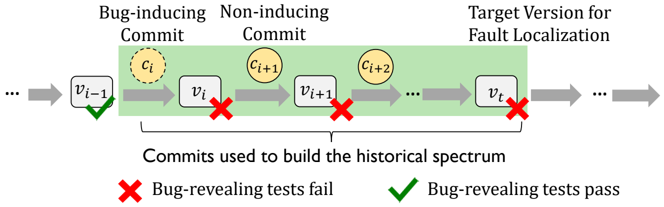


Fig. 5: Concept of Historical Spectrum

3 Approach

We propose Historical Spectrum based Fault Localization (HSFL) in this paper. HSFL takes the source code, the version history and the associated test suite of a project as inputs. It works at the **statement** level and contains three steps. The overview of HSFL is shown in Figure 6.

First, it identifies the bug-inducing commit from the version histories for each bug-revealing test case to identify a set of suspicious code entities. Second, HSFL constructs a historical spectrum (i.e., denoted as Histrum) to trace the evolutions of each suspicious code entity from the *bug-inducing version* (i.e., the version after the bug-inducing commit is adopted) to the target version via history slicing [29]. Third, HSFL computes the suspicious score for each code entity based on Histrum. In particular, it works like SBFL, where a bug-inducing commit and a non-inducing commit are analogous to a failing test and a passing test in SBFL, respectively. As such, the ranking formulae designed for SBFL (e.g., Ochiai [30] and Tarantula [32]) can be deployed to compute the suspicious score based on the Histrum. HSFL further leverages the conventional coverage based spectrum used in SBFL to further differentiate buggy code entities from non-buggy ones in Histrum to generate the final rankings.

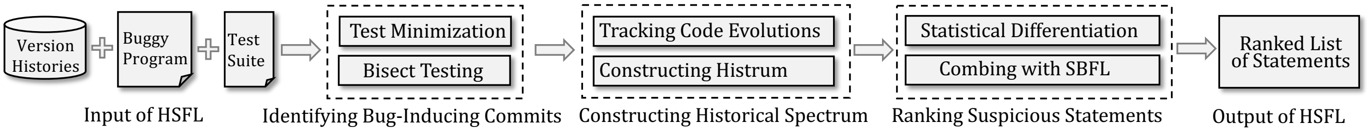


Fig. 6: Overview of HSFL

3.1 Identifying Bug-Inducing Commits

As observed from the open source community, developers identify bug-inducing commit by finding the first commit on which the bug-revealing test starts to fail. For instance, debugging activities such as *"Confirmed, the test passes before this commit (LUCENE-6758) and fails after"* [41] can be frequently observed in bug reports. Based on such observations and a recent study [25], we formally define bug-inducing commits as follows in this study:

Definition 1. Given a bug manifested by a bug-revealing test t_f, the associated ***bug-inducing commit*** is the commit before which t_f passes and after which t_f fails.

To identify the bug-inducing commits, HSFL conducts binary search on the complete version history (automated by `git bisect`) following the heuristic used by existing approaches [25], [42], [43]. Such a heuristic is also adopted by developers from open source community as observed in the debugging practices discussed in Section 2.1. Specifically, we extract the bug-revealing test t_f from the target version and then execute it on older versions of the program. However, identifying the bug-inducing commit precisely for a bug-revealing test t_f can be non-trivial due to the following reasons.

```
    @Test
    public void testCreateNumber() {
19:    // LANG-521
20:    Float number = Float.valueOf("2.");
21:    assertEquals("createNumber LANG-521 failed",
           number, NumberUtils.createNumber("2."));
23:    // LANG-693
24:    assertEquals("createNumber LANG-693 failed",
           Double.valueOf(Double.MAX_VALUE),
           NumberUtils.createNumber("" + Double.MAX_VALUE));
    }
```

Fig. 7: A Test Case of Project Lang

First, a unit test case might involve the testing logics of several bugs. To ease the explanation, we refer to the manifestation of a bug revealed by a bug-revealing test as the "failing signature", which includes the information about the point of the failure and the error message generated in the failing test run. For example, the test method `testCreateNumber()` in project `Lang` tests the functionalities of multiple bugs (i.e., issues `Lang-521` and `Lang-693`) as shown in Figure 7. Suppose our *target bug* for fault localization is `Lang-521` here. However, running such a test case on previous versions might fail due to different bugs, manifested by different failing signatures thrown by the test (e.g., "*createNumber LANG-521 failed, expected..., but ...*" or "*createdNumber LANG-693 failed, expected..., but ...*"). This is because some bug fixes (e.g., fix for `Lang-693`) might be reverted if we roll back to previous versions. As a result, the test case will fail if it is executed on these versions. This will hinder us to identify the bug-inducing commit for the target bug since the bug-revealing test fails due to another bug (i.e., `Lang-693`). Our approach takes the following steps to address this challenge. It first analyzes the failing signature of the bug-revealing test executed on the target version to obtain the failure point triggering the target bug (e.g., the assertion statement line at 21 in Figure 7). It then comments out other assertion statements (e.g., line 24) within the method to remove those testing logics for other bugs. Those statements constructing the data structures for assertion statements (e.g., line 20) will be kept to make the code of the bug-revealing assertions runnable.

Second, some test cases might require extra self-defined features to construct complex objects for testing. These test cases might not be able to be executed successfully on previous versions if the required features have not been implemented on that version. For example, some test cases of project `Lang` require an extra class `FormatFactory` to construct objects to test the functionalities in class `ExtendedMessageFormat`. However, class `FormatFactory` is introduced in version `#695289c`. Therefore, those tests requiring this class cannot be run successfully on those versions prior to `#695289c`. For such cases, identifying the bug-inducing commit precisely is difficult. To handle these cases, we introduce the concept of *likely-inducing commits*. Likely-inducing commits include the first commit on which the bug-revealing test fails with the targeted failing signature and those commits on which the bug-revealing test is unable to run successfully.

For each bug-revealing test t_f, we can identify a bug-inducing commit or a range of likely-inducing commits. The identified bug-inducing commit can be either an initial code commit or a subsequent code commit during software evolution [23]. If the bug-inducing commit is an initial code commit, it indicates that the first version of the source file contains the bug. The bug is introduced by subsequent code modifications otherwise. The initial code commit is usually larger in its size compared with subsequent commits [23]. Different types of inducing commits would have the impacts on the performance of fault localization, which is discussed in Section 6.2. For a target bug, we can identify a set of bug-inducing commits $\mathcal{C}_I$ or a set of likely-inducing commits $\mathcal{C}_L$, since there might be ***multiple*** bug-revealing tests for the target bug. Those commits submitted prior to the target version and do not belong to either $\mathcal{C}_I$ or $\mathcal{C}_L$ are denoted as *non-inducing commits* $\mathcal{C}_N$ with respect to the target bug since they do not change the status of the bug-revealing tests. For each commit $c \in \mathcal{C}_I \vee \mathcal{C}_L$, we denote the statements introduced (i.e., modified or originally added) in it as *suspicious code entities* (i.e., denoted as $\mathcal{S}_H$).

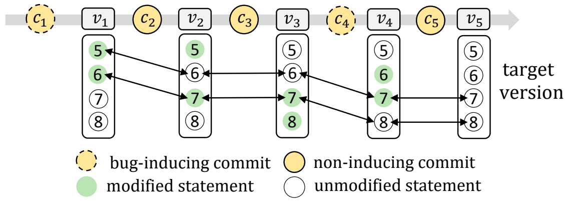
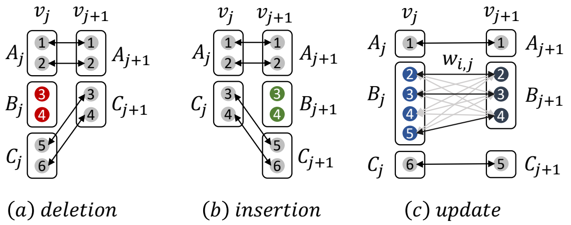


Fig. 8: An example of Historical Spectrum



(*a*) *deletion* (*b*) *insertion* (*c*) *update*

Fig. 9: Line Mappings between two Consecutive Versions of Deleted, Added and Updated Change Hunks

3.2 Constructing Historical Spectrum

To leverage the suspicious code entities $\mathcal{S}_H$ obtained from bug-inducing commits to locate faults at the target version v_t, HSFL constructs Histrums for $\mathcal{S}_H$ so as to map the statements in $\mathcal{S}_H$ to the ones in the target version v_t. It also tracks the evoluations of $\mathcal{S}_H$ to see if they have been further modified by other commits subsequent to the bug-inducing commit.

Figure 8 shows an example of a constructed historical spectrum. Suppose c_1 is a bug-inducing commit, and statements 5 and 6 are modified by c_1. In order to leverage such information to locate faults in the target version v_5, Histrum tracks the evolution of each statement to see whether it has been further modified by other commits. For example, statement 6 has been further modified by two non-inducing commit (i.e., c_2 and c_3) and evolves to statement 8 at v_5. Statement 5 has been further modified by one bug-inducing commit (i.e., c_4) and evolves to statement 7 at v_5. Here, c_4 is another bug-inducing commit identified by other bug-revealing test for the same bug.

To construct historical spectrum, we use history slicing [29] to track the modification of $\mathcal{S}_H$ from the bug-inducing version to the target version. Without loss of the generality, we suppose the version history is $\langle v_1, ..., v_j, v_{j+1}, ..., v_n\rangle$, where v_1 is the bug-inducing version and v_n is the target version. For each pair of two consecutive versions $\langle v_j, v_{j+1}\rangle$, we use the function $\mathcal{M}_{j\mapsto j+1}(s)$ to represent the statement in v_{j+1} that is mapped from the statement s in v_j. We leverage GUMTREE [44] to analyze the changes between two versions and remove those non-semantic changes (e.g., formatting or modification of comments). There are three different types of changes made between any two versions, which are *deletion*, *insertion*, and *update*. Figure 9 shows the change hunks for these three types of changes, where A and C denote the contextual part and B denotes the changed part. Since the statements in hunks A_j and C_j in version v_j are unchanged, we can directly map them to those in hunks A_{j+1} and C_{j+1} in the next version v_{j+1}. There are three cases of the changed hunks. The mappings for statements in a deleted or inserted hunk can be readily built as follows. In the case of deletion, $\mathcal{M}_{j\mapsto j+1}(s) = null, s \in B_j$, since the statements in hunk B_j are deleted and thus the mappings are null. In the case of insertion, there are no statements in v_j that can be mapped to the statements in hunk B_{j+1} at version v_{j+1}. The case for update is more complicated. A continuous set of statements B_j are modified to B_{j+1} as shown in Figure 9(c). To find the optimum mappings from B_j to B_{j+1}, we follow the work of history slicing [29] and approach it as the problem of finding the minimum matching of a weighted bipartite graph. The weight between any two lines as shown in Figure 9(c) is computed as their *Levenshtein Edit Distance* [45].

A bipartite graph is a graph whose vertices can be divided into two disjoint and independent sets such that every edge connects a vertex in one set to another. In our setting, we regard each statement as a vertex, and thus we have two disjoint sets of vertices B_j and B_{j+1}. We connect each statement in B_j to each of the statements in B_{j+1} with the weight of *Levenshtein Edit Distance* [45] between these two lines of statements. For example, as shown in Figure 9 (c), we connect line 2 in B_j to each of the statements in B_{j+1} (i.e., line 2 to 4). To obtain the Levenshtein Edit Distance, we first tokenize each line of statement to a vector of words following existing heuristics [23]. We then calculate the minimum number of single-word edits (insertions, deletions or substitutions) required to change one vector of words into the other. The smaller the number, the higher similarity between these two statements. We finally find the minimum weight bipartite matching using the Kuhn-Munkres algorithm [46], and record the identified best mapping between these two hunks in function $\mathcal{M}_{j\mapsto j+1}$. In our example shown in Figure 9(c), $\mathcal{M}_{j\mapsto j+1}(5) = 4$ and $\mathcal{M}_{j\mapsto j+1}(4) = null$.

Our goal is to obtain $\mathcal{M}_{1\mapsto N}(s)$, which finds the statement in v_n that is mapped from the statement s in v_1. Using the function $\mathcal{M}_{j\mapsto j+1}(s)$ for each two consecutive versions, we can gradually calculate $\mathcal{M}_{1\mapsto N} = \mathcal{M}_{N-1\mapsto N} \circ \mathcal{M}_{N-2\mapsto N-1} \circ ... \circ \mathcal{M}_{1\mapsto 2}(s)$. Note that not all statements in $\mathcal{S}_H$ can be mapped to the target version since some statements might be deleted during evolution and the mapping function will return *null* for such cases. Using the function $\mathcal{M}_{1\mapsto N}$, we can successfully map the statements in $\mathcal{S}_H$ in the bug-inducing version to the statements in the target version. Similar procedures are conducted for those likely-inducing commits in $\mathcal{C}_L$.

3.3 Ranking Suspicious Statements

After mapping $\mathcal{S}_H$ to the target version based on the Histrum, we can obtain a set of suspicious statements $\mathcal{S}_C$ at the target version v_t. Specifically, $\mathcal{S}_C = \{\mathcal{M}_{1\mapsto N}(s), \forall s \in \mathcal{S}_H\}$. HSFL then ranks the statements in $\mathcal{S}_C$ to locate faults. The main challenge is to differentiate the buggy statements from those irrelevant ones since $\mathcal{S}_C$ might contain noises (i.e., statements irrelevant to the bugs).

To address this challenge, HSFL first leverages the historical spectrum built in the second step. Specifically, we leverage the history spectra information to compute their suspiciousness of being the root cause of the targeted bug. The intuition is that those code entities modified by more bug-inducing commits but fewer non-inducing commits are more likely to be the root cause of the bug. It works like SBFL, where a bug-inducing commit and a non-inducing commit are analogous to a failing test and a passing test in SBFL, respectively. As such, the techniques designed for SBFL (e.g., Ochiai [30] and Tarantula [32]) can be deployed to compute the suspicious score based on the historical spectrum. Specifically, we use Ochiai [30] by default to compute the suspicious score for each statement $s \in \mathcal{S}_C$ in HSFL since it has been reported to be the optimum formula for SBFL [2]. In particular, we investigate the impact of different SBFL formulae on HSFL in Section 5.2. Suppose that c_i is the bug-inducing commit of s and it has been further modified by a list of commits $\mathcal{C} = \langle c_i, ..., c_j, c_{j+1}, ..., c_n \rangle$, we can calculate its suspicious as:

$$Histrum(s, c_i) = \frac{induce(s)}{\sqrt{N_I * (induce(s) + noninduce(s))}} \quad (1)$$

in which $induce(s)$ denotes the number of inducing commits that modified statement s, specifically, $induce(s) = |\{c : c \in \mathcal{C}_I \wedge c \in \mathcal{C}\}|$; $noninduce(s)$ is the number of non-inducing commits that modified statement s, specifically, $noninduce(s) = |\{c : c \in \mathcal{C} \wedge c \notin \mathcal{C}_I\}|$; N_I denotes the total number of bug-inducing commits which is $|\mathcal{C}_I|$. Let us further illustrate this using our example shown in Figure 8. The suspicious score for statement 7 at the target version is 1, which is calculated as $2/\sqrt{2 * (2 + 0)}$, while the suspicious score for statement 8 is 0.408 $(1/\sqrt{2 * (1 + 2)})$. Therefore, statement 7 is more likely to be the root cause of the bug compared with 8.

Therefore, a statement s at the target version might have multiple values obtained from the Histrum model since it is possible that s is affected by multiple bug-inducing commits. For the example shown in Figure 8, HSFL will also build another historical spectrum starting from v_4 after the bug-inducing commit c_4 is adopted. Therefore, statement 7 in the target version will have another suspicious value. We use the maximum value of $Histrum(s, c_i)$ as the final score for statement s. Specifically, $Histrum(s) = \max\{Histrum(s, c_i), c_i \in \mathcal{C}_I\}$.

To further help differentiate buggy statements from irrelevant ones in $\mathcal{S}_C$, HSFL leverages the conventional coverage based spectrum used in SBFL. This intuition follows that of existing FL techniques [2], [11], where buggy statements are more likely to be executed by failing tests than passing tests in the target version v_t. By integrating this with Histrum, HSFL produces the final results:

$$HSFL(s) = \begin{cases} (1 - \alpha) * SBFL(s) & s \in \mathcal{A} \wedge s \notin \mathcal{S}_C \\ (1 - \alpha) * SBFL(s) + \alpha * Histrum(s) & s \in \mathcal{A} \wedge s \in \mathcal{S}_C \\ 0 & otherwise \end{cases} \quad (2)$$

in which $\mathcal{A}$ denotes the set of suspicious statements executed by the bug-revealing tests at v_t, and α is the weight of combining Histrum and SBFL. By default, we set α to 0.5. We investigate the effect of α in the overall performance of HSFL in Section 5.3. In Equation 2, we set the final scores as 0 for those statements that have not been executed by the bug-revealing tests on v_t. The intuition is that a statement is unlikely to be the root cause if it has not been executed by any of the bug-revealing tests on the targeted version following existing studies [2], [11], [33]. In this way, HSFL can further eliminate the nosies in $\mathcal{S}_C$ caused by the potential tangled statements in the bug-inducing commits.

For likely-inducing commits in $\mathcal{C}_L$ and the corresponding suspicious buggy statements $\mathcal{S}_H$, similar procedures are conducted. However, since those commits do not definitely introduce the bug, we decrease the effects of the Histrum model by adding a weight to the value obtained from Equation 1. Specifically, $Histrum_L(s) = Histrum(s)/|\mathcal{C}_L|$. The larger range of the likely-inducing commits, the smaller weight it gets. The intuition is that the likelihood of each likely-inducing commit in set $\mathcal{C}_L$ to be the bug-inducing commit is decreasing with the increase of size of $\mathcal{C}_L$.

Using the final scores obtained by HSFL(s), we then rank all the suspicious statements at the targeted version v_t.

4 Experiment Setup

4.1 Subjects

We evaluate the effectiveness of HSFL on the benchmark dataset Defects4J [31]. This benchmark contains substantial real bugs extracted from large open source projects, and it was built to facilitate controlled experiments in software debugging and testing research [31]. Defects4J has been widely adopted by recent studies on fault localization and program repair (e.g., [11], [47], [48], [49]). Following existing studies [12], [13], we use all the five projects in Defects4J of *version* 1.0.1 with a total of 357 real bugs as subjects for our experiments. Their demography is shown in Table 1.

TABLE 1: Subjects for Evaluation

Subject	#Bugs	KLOC	Test KLOC	#Test Cases
Commons **Lang**	65	22	6	2,245
JFree**Chart**	26	96	50	2,205
Commons **Math**	106	85	19	3,602
Joda-**Time**	27	28	53	4,130
Closure Compiler	133	147	104	7,929
Total	357	378	232	20,111

4.2 Measurements

To measure the effectiveness of HSFL, we adopt the following three widely-used metrics in our study [12], [13], [23].

Top-N: This metric reports the number of bugs, whose buggy entities (i.e., statements in our evaluation setting) can be discovered by examining the top N(N=1,2,3,...) of the returned suspicious list of code entities. The higher the value, the less efforts required for developers to locate the bug, and thus the better performance.

MRR: *Mean Reciprocal Rank* [50] is the average of the reciprocal ranks of a set of queries. This is a statistic for evaluating a process that produces a list of possible responses to a query [51]. The reciprocal rank of a query is the multiplicative inverse of the rank of the first relevant answer found. This metric is widely used to evaluate the ability to locate the first buggy statement for a bug [13], [23]. The larger the MRR value, the better the performance.

MAP: *Mean Average Precision* [52] is by far the most commonly used traditional information retrieval metric. It provides a single value measuring the quality of information retrieval performance [51]. It takes all the relevant answers into consideration with their ranks for a single query. This metric is also widely used to evaluate the ability of approaches to locate all the buggy statements of a bug [13], [23]. The larger the MAP value, the better the performance.

When multiple statements have the same suspicious score, we use the average rank to present their final rankings, following the strategy to handle the tie issues widely adopted by existing fault localization techniques [11], [12], [53], [54].

4.3 Research Questions

This study aims to answer the following research questions.

- **RQ1**: How does HSFL perform in locating real bugs?

To answer this question, we apply HSFL to the 357 real bugs from DEFECTS4J as shown in Table 1, and then evaluate the results using the metrics described in Section 4.2. We also compare our results with those obtained by conventional SBFL reported by the state-of-the-art technique [11]. We select SBFL techniques as the baseline in this RQ, since SBFL is the most widely investigated technique and has been reported to achieve good performance compared with the others [55]. We compare HSFL with other baselines systematically in the subsequent research questions.

- **RQ2**: How do different formulae affect the performance of HSFL?

We use Ochiai [30] by default in HSFL since it has been reported to be the best formula for SBFL [2], [11]. However, there are multiple formulae proposed and it is yet unknown whether Ochiai is the optimum for HSFL. In this research question, we choose the five best-studied SBFL formulae [2], [11] and investigate how these different formulae affect the performance of HSFL. The five adapted SBFL formulae for HSFL are shown in Table 2. In these formulae, $induce(s)$, $noninduce(s)$ and N_I are defined in the same way as described in Equation 1, and N_N denotes the total number of non-inducing commits for the target bug.

TABLE 2: The Five Adapted SBFL Formulae for HSFL

Name	Formula
Tarantula [32]	$S(s) = \frac{induce(s)/N_I}{induce(s)/N_I + noninduce(s)/N_N}$
Ochiai [30]	$S(s) = \frac{induce(s)}{\sqrt{N_I * (induce(s) + noninduce(s))}}$
Op2 [34]	$S(s) = induce(s) - \frac{noninduce(s)}{(N_N + 1)}$
Barinel [35]	$S(s) = 1 - \frac{noninduce(s)}{noninduce(s) + induce(s)}$
DStar [33]	$S(s) = \frac{induce(s)^{\star}}{noninduce(s) + N_I - induce(s)}$

We used $\star = 2$, the most thoroughly explored value [11]

- **RQ3**: How does the weight α affect HSFL's performance?

HSFL sets the weight to 0.5 by default to let the coverage based spectrum has the same weight as Histrum. However, it is yet unknown whether the default value is the optimum. In this research question, we investigate the effect of this weight α on the overall performance of HSFL. Specifically, we change the weight from 0.0 to 1.0 with a step size of 0.1, and then examine the performance of HSFL based on Ochiai.

- **RQ4**: Can HSFL improve the performance of search-based automated program repair?

Automated program repair techniques extensively rely on SBFL to generate the fault space [7], [9], [47], [48], which affects the search space of search-based APR techniques [7]. Existing search-based APR techniques are known to suffer from the search space explosion problem [8]. Therefore, better fault space is always demanded to improve the efficiency for searching the correct patches [56]. This research question investigates the practical usefulness of HSFL in improving the performance of the state-of-the-art search-based APR techniques.

- **RQ5**: Can HSFL outperform other families of fault localization techniques?

Besides spectrum-based fault localization techniques, there are many other families of techniques proposed over the years with the aim to locate suspicious code elements at the **statement** level. Based on recent studies and a systematic survey [11], [55], we summarize existing techniques to the following nine families:

TABLE 3: Popular Families of FL Techniques

Family	Techniques	Description
Spectrum-based	Ochiai [30] Dstar [33]	utilizing test coverage information
Mutation-based	MUSE [57] Metallaxis [58]	utilizing test results from mutating the program
Slicing-based	Union [55], [59], Intersection [55], [59], Frequency [55], [59]	utilizing dynamic program dependencies
Stack trace-based	StackTrace [37]	utilizing crash reports embedded in bug reports
Predicate swithcing	PredicateSwitching [55], [60]	utilizing test results from mutating the results of conditional expressions
IR-based	BugLocator [61]	utilizing the token information of bug reports
History-based	BugSpots [62]	utilizing the development history
Hybrid	MCBFL [11]	combing different techniques using heuristics
Learning-based	Learning-to-rank [55]	combing different techniques utilizing machine learning techniques

Different techniques are proposed via leveraging divergent sources of information, such as the *test coverage*, *test results from mutating the program*, *dynamic program dependencies*, *crash reports* and so on. Moreover, recent studies proposed the hybrid techniques, which leverage multiple sources of information. For instance, Pearson *et al.* proposed to combine mutation-based and spectrum-based techniques together [11]. Learning-based techniques also combine multiple sources of information. Specifically, they treat the results of each technique as individual features, and then leverage machine learning techniques to learn the optimum way to combine them automatically. Learning-based techniques differ themselves from hybrid ones in that they need a separate set of data for model training.

In this RQ, we compare HSFL with these techniques with the aim to further investigate the effectiveness of HSFL. IR-based and History-based are excluded in our comparison since these two families of techniques have been reported to achieve extremely poor performance at the statement level [55]. We compare and integrate HSFL with the state-of-the-art learning-based technique in a separate research question (i.e., RQ7) since it requires to divide the data into a training and a testing part and then involves a training process. As a result, 10 baselines from six different families have been selected for comparison in this RQ in total. We compare their performances with HSFL based on the DEFECTS4J dataset in terms of metrics MAP, MRR and Top-N.

- **RQ6**: Can our Histrum model boost the performance of other families of fault localization techniques?

Our proposed Histrum model provides a novel perspective to locate faults in terms of evolution history, and it can actually be combined with any families of fault localization techniques besides SBFL. In this RQ, we investigate whether Histrum can boost the performance of other types of fault localization as displayed in Table 3. Specifically, similar to Equation 2, we integrate Histrum with a FL technique as follows:

$$Boost(s) = \begin{cases} (1-\alpha) * FL(s) & s \in \mathcal{A} \wedge s \notin \mathcal{S}_C \\ (1-\alpha) * FL(s) + \alpha * Histrum(s) & s \in \mathcal{A} \wedge s \in \mathcal{S}_C \\ 0 & otherwise \end{cases} \quad (3)$$

in which FL denotes an existing FL technique (e.g., MUSE [57], MCBFL [11]), $\mathcal{A}$ denotes the set of suspicious statements identified by the FL technique, and α is the combining weight. We then compare the boosted results after integrating Histrum with the results of the original FL technique on the DEFECTS4J benchmark.

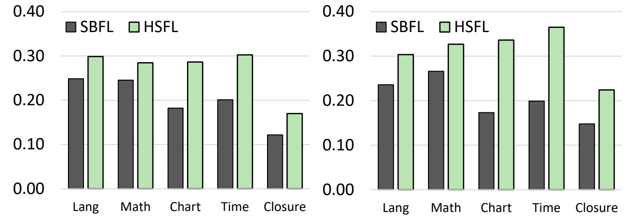
- **RQ7**: Can our approach boost the performance of learning-to-rank techniques?

Learning-to-rank techniques were proposed to combine multiple FL techniques together [55], [63]. The basic idea is to treat the suspicious score produced by each technique as a unique feature and then use machine learning techniques to find the model that ranks the faulty statement as high as possible. In this RQ, we investigate whether our proposed technique can be leveraged as a feature to boost the performance of existing learning-to-rank techniques. Specifically, we choose the state-of-the-art learning-to-rank technique [55] for investigation.

Pearson *et al.* have evaluated multiple SBFL techniques on the DEFECTS4J recently [11]. They provided the oracle (i.e., the buggy statements) and the conventional coverage-based spectrum for each bug. Zou *et al.* recently have conducted a systematic empirical study to investigate different families of FL techniques and their combinations. They provided their experimental data and the results of different families of FL techniques as well as the newly proposed learning-to-rank technique. To facilitate the reproduction of our evaluation results, we leverage those publicly available dataset to generate the results of SBFL and other baseline approaches instead of instrumenting and computing by ourselves. We implemented HSFL in Java. Our experiments are run on a CentOS server with 2x Intel Xeon E5-2450 CPU@2.1GHz and 192GB physical memory. All the experimental data are publicly available at: **https://github.com/justinwm/HSFL**

5 EVALUATION RESULTS

In this section, we answer the four designed research questions.



(a) MAP (b) MRR

Fig. 10: Results of MAP and MRR of HSFL and SBFL

5.1 RQ1:Effectiveness of HSFL

To answer RQ1, we present the results of HSFL evaluated on the five projects shown in Table 1. Specifically, we use Ochiai [30] in Histrum and the default combining weight α=0.5 in HSFL. The results of SBFL are also generated using Ochiai. Figure 10 displays the results of HSFL and SBFL in terms of MAP and MRR, which show that HSFL outperforms SBFL for all the five subjects in terms of both of the two metrics. The improvement of MAP ranges from 16.0% to 57.2%, and the improvement of MRR ranges from 22.8% to 94.0%. On average, HSFL is able to achieve an improvement of 28.3% and 40.8% for MAP and MRR respectively.

Figure 11 shows the results of HSFL and SBFL evaluated by metric Top-N. HSFL ranks the buggy statement at Top-1 for 64 bugs, which is 28 more than SBFL, achieving an significant improvement of 77.8%. HSFL ranks the buggy statements for 146 bugs within top 5, 177 within top 10, and 205 within top 20, achieving an improvement of 33.9%, 18.0% and 10.8% respectively. The rankings of buggy statements is crucial to measure the usefulness of fault localization techniques. Developers usually only spend the efforts to inspect the top-ranked suspicious statements. e.g., over 70% developers only check Top-5 ranked elements [6]. These results shown in Figure 10 and Figure 11 indicate that HSFL is more effective in locating bugs compared with SBFL and is more useful for developers in practice.

5.2 RQ2:Effect of Different Formulae in HSFL

To answer RQ2, we present the results of HSFL using different formulae as shown in Table 2 in the Histrum model. For the combining weight α, we still use the default value 0.5. Note that when we use a formula (e.g., DStar [33]) in Histrum, we also adopt the same formula for the conventional SBFL in Equation 2. Table 4 and 5 show the results of MAP/MRR and Top-N respectively. In these

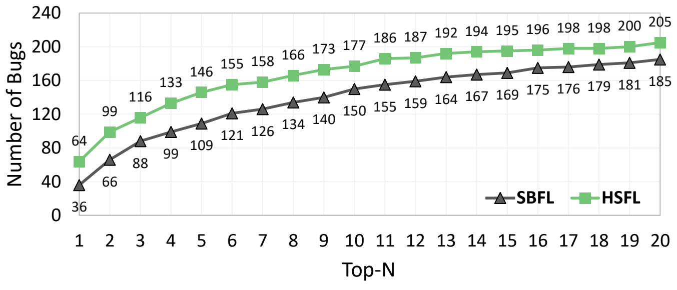


Fig. 11: Results of Top-N of HSFL and SBFL

TABLE 4: Performance of HSFL and SBFL using Different Formulae Evaluated by MAP and MRR

Pr	Tech	MAP			MRR		
		SBFL	HSFL	Impr(%)	SBFL	HSFL	Impr(%)
Lang	Tarantula	0.244	0.282	↑15.32%	0.236	0.292	↑23.91%
	Ochiai	0.248	0.299	↑20.27%	0.236	0.303	↑28.72%
	Dstar	0.248	0.303	↑22.14%	0.236	0.318	↑34.99%
	Op2	0.240	0.289	↑20.48%	0.228	0.296	↑29.72%
	Barinel	0.250	0.300	↑20.08%	0.241	0.309	↑28.12%
Math	Tarantula	0.247	0.283	↑14.75%	0.261	0.309	↑18.43%
	Ochiai	0.245	0.285	↑16.01%	0.266	0.327	↑22.83%
	Dstar	0.226	0.267	↑17.78%	0.253	0.305	↑20.94%
	Op2	0.217	0.255	↑17.18%	0.228	0.278	↑21.61%
	Barinel	0.247	0.284	↑15.06%	0.261	0.314	↑20.24%
Chart	Tarantula	0.160	0.215	↑33.97%	0.127	0.232	↑82.63%
	Ochiai	0.182	0.286	↑57.16%	0.173	0.336	↑94.00%
	Dstar	0.182	0.234	↑19.01%	0.173	0.259	↑31.48%
	Op2	0.154	0.230	↑49.82%	0.157	0.261	↑66.84%
	Barinel	0.179	0.254	↑41.54%	0.147	0.271	↑84.91%
Time	Tarantula	0.212	0.280	↑32.31%	0.190	0.303	↑59.52%
	Ochiai	0.201	0.302	↑50.36%	0.199	0.365	↑83.42%
	Dstar	0.207	0.302	↑45.62%	0.210	0.357	↑69.75%
	Op2	0.198	0.248	↑17.63%	0.222	0.309	↑39.98%
	Barinel	0.212	0.285	↑34.44%	0.190	0.330	↑73.77%
Closure	Tarantula	0.111	0.166	↑50.55%	0.135	0.203	↑50.82%
	Ochiai	0.122	0.170	↑40.12%	0.148	0.225	↑52.20%
	Dstar	0.133	0.139	↑4.66%	0.170	0.178	↑4.44%
	Op2	0.143	0.146	↑1.63%	0.171	0.175	↑2.42%
	Barinel	0.111	0.159	↑43.57%	0.135	0.209	↑55.10%
Average	Tarantula	0.187	0.234	↑25.50%	0.194	0.260	↑34.11%
	Ochiai	0.192	0.246	↑28.27%	0.205	0.288	↑40.81%
	Dstar	0.191	0.226	↑17.80%	0.212	0.261	↑23.23%
	Op2	0.188	0.218	↑16.14%	0.201	0.244	↑21.24%
	Barinel	0.189	0.238	↑25.95%	0.197	0.272	↑38.32%

The highest improvement for each subject is highlighted in 🟩

TABLE 5: Performance of HSFL and SBFL using Different Formulae Evaluated by Top-N

Pr	Tech	Top-1		Top-5		Top-10		Top-20	
		SBFL	HSFL	SBFL	HSFL	SBFL	HSFL	SBFL	HSFL
Lang	Tarantula	5	11	26	28	35	37	42	46
	Ochiai	4	11	28	29	36	39	42	47
	Dstar	4	11	29	32	38	41	44	51
	Op2	4	11	26	29	35	39	41	47
	Barinel	5	12	27	28	36	39	43	47
Math	Tarantula	18	18	36	50	47	63	59	70
	Ochiai	19	21	35	49	47	61	59	70
	Dstar	18	19	32	47	44	60	56	69
	Op2	16	16	29	43	39	57	52	65
	Barinel	18	19	36	50	47	63	59	70
Chart	Tarantula	0	1	5	11	12	16	15	16
	Ochiai	1	4	7	14	14	17	16	17
	Dstar	2	3	7	10	13	14	14	14
	Op2	1	3	6	10	12	13	15	14
	Barinel	0	2	6	12	13	17	16	17
Time	Tarantula	1	5	12	12	14	14	17	15
	Ochiai	1	6	13	16	15	17	17	18
	Dstar	1	5	13	17	15	18	18	19
	Op2	2	5	12	12	14	13	16	14
	Barinel	1	6	12	13	14	14	17	16
Closure	Tarantula	10	19	24	35	35	43	49	55
	Ochiai	11	22	26	38	38	43	51	53
	Dstar	16	18	27	29	39	37	48	45
	Op2	15	18	28	27	40	38	52	44
	Barinel	10	20	24	37	35	42	49	52
Sum	Tarantula	34	54 ↑59%	103	136 ↑32%	143	173 ↑21%	182	202 ↑11%
	Ochiai	36	64 ↑78%	109	146 ↑34%	150	177 ↑18%	185	205 ↑11%
	Dstar	41	56 ↑37%	108	135 ↑25%	149	170 ↑14%	180	198 ↑10%
	Op2	38	53 ↑39%	101	121 ↑20%	140	159 ↑14%	176	184 ↑5%
	Barinel	34	59 ↑74%	105	140 ↑33%	145	175 ↑21%	184	202 ↑10%

two tables, different columns represent different metrics evaluated while different rows represent the projects with different techniques used. The bottom portions of the two tables show the summary of the results (i.e., the weighted average results for MAP and MRR, and the sum numbers for Top-N). Figure 12 displayed the weighted average results over the five subjects of SBFL and HSFL in terms of MAP and MRR using the five different formulae.

In terms of MAP and MRR, HSFL achieves better performance than SBFL for all subjects using different formulae. On average, adopting Ochiai in HSFL achieves the optimum performance (i.e., with an average MAP of 0.246 and MRR of 0.288), and also achieves the optimum improvement compared with SBFL (i.e., with an average improvement of 28.3% for MAP and 40.8% for MRR). The formula Barinel achieves the second best performance with an average MAP of 0.238 and MRR of 0.272, and it achieves an average improvement of 26.0% and 38.3% for MAP and MRR respectively. Adopting the formulae Tarantula, DStar and Op2 in HSFL also achieves better results compared with SBFL. Specifically, the improvements for MAP are 25.5%, 17.8% and 16.1% while the improvements for MRR are 34.1%, 23.2% and 21.2%.

Similar results are observed using the metric Top-N. Adopting the formula Ochiai in HSFL achieves the optimum performance (e.g., locating 64 bugs at Top-1 and 146 at Top-5), followed by the formula Barinel (e.g., locating 59 bugs at Top-1 and 140 at Top-5). HSFL also outperforms SBFL by adopting the other three formulae in the Histrum model.

Adopting Ochiai [30] in HSFL achieves the best performance on average, we then use the one-sided Mann-Whitney U-Test [64] to see whether it is significantly better than the other four techniques. The results show that the performance of adopting Ochiai [30] in HSFL is only significantly better than that of adopting Op2 ($p < 0.05$) while the difference is not significant for the other three techniques. These indicate that techniques Ochiai [30], Tarantula [32], DStar [33] and Barinel [35] are suggested to be applied in HSFL. However, all these techniques are designed for the spectrum built from conventional testing coverage, whether our proposed historical spectrum requires specific designed techniques to achieve the optimum performance remains unknown since these two types spectrum are different intrinsically. We leave the design of specific techniques for Histrum as our future work.

5.3 RQ3: Effects of the Combining Weight α

To answer RQ3, we present the results of HSFL using the formula Ochiai [30] while using a series of different weights α (i.e., from 0.0 to 1.0 with a step size of 0.1) to combine Histrum and SBFL in HSFL. Figure 13 plots the results of HSFL with respect to MAP and MRR for all the subjects. Specifically, the $x-$axis represents the weight α used in Equation 2 and the $y-$axis represents the values of MAP and MRR. From the results, we can see that the performance HSFL share similar patterns for the five different subjects. The performance of HSFL is increasing when α is small (≤ 0.5). It reaches its peak when α is around 0.5 and 0.7, and then starts to decrease when α continues increasing. When averaged over all the five subjects, HSFL achieves its optimum performance when α is 0.5. These results indicate that HSFL obtain its best performance when the Histrum

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2948158, IEEE Transactions on Software Engineering

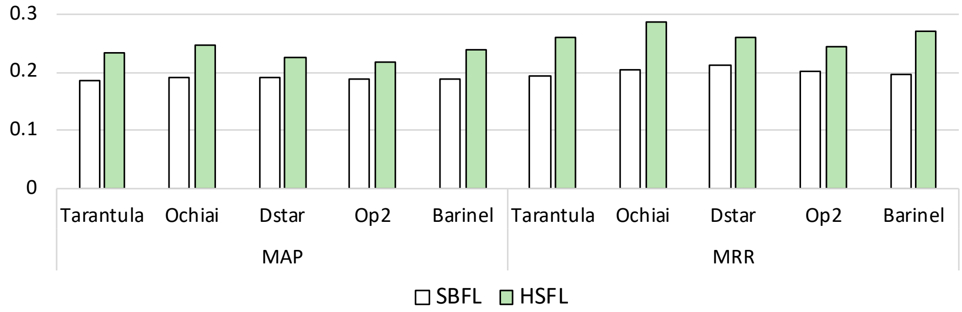
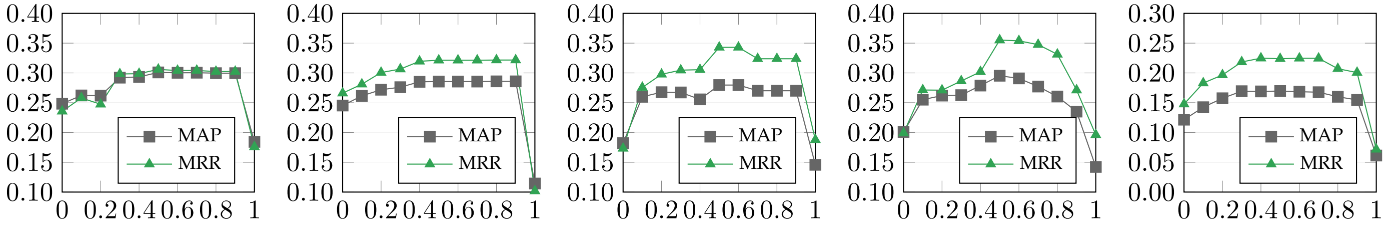


Fig. 12: Average Results of SBFL and HSFL in terms of MAP and MRR using Different Formulae



(a) Lang (b) Math (c) Chart (d) Time (e) Closure

Fig. 13: The Performance of HSFL with Different Combination Weight α of Histrum

model and SBFL model are considered to be similarly important.

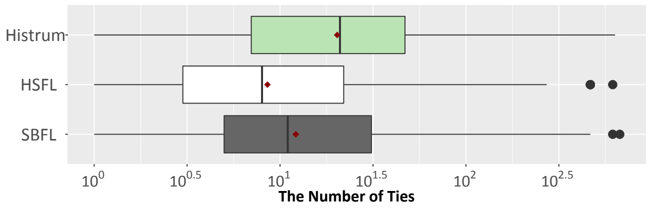


Fig. 14: The Number of Ties of the FL Results

When α=0, HSFL is equivalent to the conventional SBFL. When α=1, HSFL only includes Histrum model. As we can see, only using one of the two models in HSFL cannot achieve the optimum performance, while combining both models with similar weights performs the best. We found that using only the SBFL model or the Histrum model will result in serious tie issues via further investigation, which is caused by the limited number of bug-revealing tests. Due to the limited number of bug-revealing tests, SBFL is known to suffer from the tie issue problem [21], [22]. The Histrum model also suffers from this problem since the number of bug-inducing commits is limited as a result of the limited number of bug-revealing tests (note that HSFL only identifies one bug-inducing commit for one bug-revealing test). These two models complement well to each other, and thus the combination relieves the tie issue. The combination of Histrum and SBFL model is expected to relieves the tie issue, since they use two different spectrum from two divergent dimensions. This is confirmed by the results displayed in Figure 14, which shows the number of non-buggy statements that are ranked in tie with the buggy statement from the results returned by only the Histrum model, SBFL and the combining of them, HSFL. The number of ties returned by HSFL is significantly smaller compared with that returned by SBFL ($p = 1.3e\text{-}06$) using the one-sided Mann-Whitney U-Test [64]. The number is also significantly smaller compared with that returned by Histrum ($p = 2.6e\text{-}14$). These results show that our proposed Histrum model can significantly mitigate the tie issue of conventional SBFL.

5.4 RQ4: Usefulness of HSFL in Automated Program Repair

We evaluate the usefulness of HSFL based on the state-of-the-art search-based APR technique CAPGEN [7] to answer this RQ. It is well-known that the *search-space explosion* and *overfitting* are the two long-standing open challenges for search-based APR [7], [8]. In this RQ, we focus on investigating whether the improvements made by HSFL over the fault space can alleviate the search-space explosion problem. Specifically, we leverage HSFL to generate the fault space for CAPGEN instead of using SBFL. We then investigate the number of candidate patches required to be validated for CAPGEN in order to find the first correct patch, and compare it with that of CAPGEN using SBFL. To avoid the side effects caused by the overfitting problem, we only selected those bugs that can be correctly repaired by CAPGEN for comparison (i.e., those patches that are semantically equivalent to developers-provided ones via manual checking). In particular, two authors were involved in the process of manual checking, and we also measured the inter-rater reliability score (i.e., mean pairwise Cohen's kappa [65]) following the latest work to measure the reliability of this process [66]. Among all the 190 generated plausible patches [7], the authors are asked to label them as "correct" and "incorrect", and the obtained score of the mean pairwise Cohen's kappa is 0.957. Such a high score demonstrated the reliability of our manual checking process [65].

This setup for this experiment follows the existing study which evaluates the effectiveness of fault localization using automated program repair [10], and Figure 15 shows the results for all the bugs that are correctly repaired by CAPGEN. The results indicate that the searching efficiency of CAPGEN can be significantly improved by leveraging HSFL. For example, for bug `Lang 57`, due to the improvement of the fault space generated by HSFL, the correct patch can be searched 30 times faster. Only for three bugs (`Math 57`, `Math 70` and `Math 63`), the searching efficiency of CAPGEN cannot be improved since the fault spaces have not been improved by HSFL. We further investigated the reasons behind and found that the bug-inducing commits identified for these three bugs are "initial" or "likely", and such commits are large in their sizes and contain lots of noises (i.e., irrelevant non-buggy statements). As a result, it makes it extremely challenging for HSFL to differentiate buggy statements from non-buggy statements. Such negative cases motivate us to investigate the effects of different types of identified bug-inducing commits on the performance of HSFL (see Section 6.2 for more discussions). However, the first correct patch is ranked only slightly lower (e.g., ranging from 0.04% to 0.09%) using the fault space generated by HSFL than that by SBFL for these three cases, and the first correct patch can be searched 3.02 times faster averaged over all bugs. These results still demonstrate the usefulness of HSFL in improving the searching efficiency of automated program techniques, which is significant since search space explosion is a long-haunting challenge in the domain of program repair.

HSFL is also expected to improve the CAPGEN's effectiveness (i.e., in terms of the number of correctly repaired bugs) via prioritizing correct patches before incorrect plausible ones. However, since CAPGEN already ranks the correct patches before the incorrect plausible ones for 95.5% of the patched bugs, the improvement that can be made by HSFL is marginal.

5.5 RQ5: Comparing with Other Families of FL

To answer RQ5, we present the results of HSFL and other families of FL techniques (i.e., as shown in Table 3) using the DEFECTS4J dataset. Similar to RQ1, Ochiai [30] is used in Histrum and the combining weight is set to α=0.5 by default in HSFL. The results of all baselines are reproduced based on the dataset of existing studies [11], [55]. Figure 16 displays the results in terms of MAP and MRR, which shows that HSFL outperforms all baselines in terms of both of the two metrics. The best family among all the baselines is the hybrid technique MCBFL, which combines spectrum-based and mutation-based techniques together [11]. Compared with MCBFL, HSFL is able to achieve an average improvement of 8.5% and 8.4% for MAP and MRR respectively. The families SBFL (i.e., Ochiai [30] and Dstar [33]) and MBFL (i.e., Metallaxis [58] and MUSE [57]) achieves even poorer performance than MCBFL. The average improvement achieved by HSFL over these two families of the techniques is at least 28.3% and 34.0% for MAP and MRR respectively. The performance of other families of techniques are poorer as displayed in Figure 16, and this is consistent with a recent survey [55].

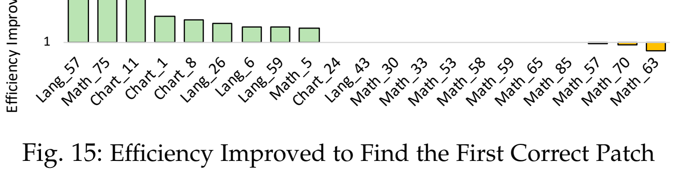


Fig. 15: Efficiency Improved to Find the First Correct Patch

Similar results have been observed in terms of the metric Top-N as displayed in Table 6. Among all the techniques, HSFL achieves the optimum performance. Second to HSFL, the hybrid technique MCBFL achieves the second optimum performance. Specifically, MCBFL ranks the buggy statements for 59 bugs within top 1, 130 bugs within top 5 and 164 within top 10. The improvement of HSFL over MCBFL is 8.5%, 12.3%, 7.9% and 4.1% for Top-1, Top-5, Top-10 and Top-20 respectively.

The results displayed in Figure 16 and Table 6 demonstrate the effectiveness of HSFL in locating bugs compared with different families of FL techniques.

5.6 RQ6: Combining with Other Families of FL

Our proposed model, Histrum, constructs spectrum along the version histories, which provides information in terms of a novel perspective to locate faults. As previously investigated in RQ1 and RQ2, it complements well to existing SBFL techniques. In this RQ, we investigate whether the Histrum model can also boost the performance of other families of FL techniques. The Histrum model can be easily integrated into other FL techniques as specified in Equation 3. To answer this RQ, we compare the result of each FL technique with that after integrating Histrum using Equation 3 on the DEFECTS4J benchmark.

Table 7 displays the results in terms of MAP, MRR and Top-N. Specifically, column "FL" denotes the original performance of a FL technique without integrating Histrum, while column "Boost" denotes the results after combining Histrum. As revealed by the results, the Histrum model is able to boost the performance of any family of FL techniques. For the hybrid techniques, the performance can be improved by 24.6% and 25.2% for MAP and MRR respectively after integrating Histrum; For SBFL techniques, the results have been well discussed in previous research questions; For mutation-based techniques, the performance can be improved by at least 29.0% and 33.5% for MAP and MRR respectively after combining Histrum; For slicing-based techniques, the Histrum model can boost the performance by at least 61.4% and 89.3% for MAP and MRR respectively; The improvements for stack trace-based techniques are 65.4% and 75.0% in terms of MAP and MRR respectively after incorporating Histrum, and these ratios are 120.6% and 109.3% for predicate switching-based techniques.

One interesting point as revealed by the results is that the hybrid techniques are able to achieve the optimum performance after combining it with Histrum (as displayed with the 🟩 background in Table 3). Specifically, it achieves an average MAP of 0.283 and MRR of 0.333, which outperforms HSFL by 14.9% and 15.6% respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2948158, IEEE Transactions on Software Engineering

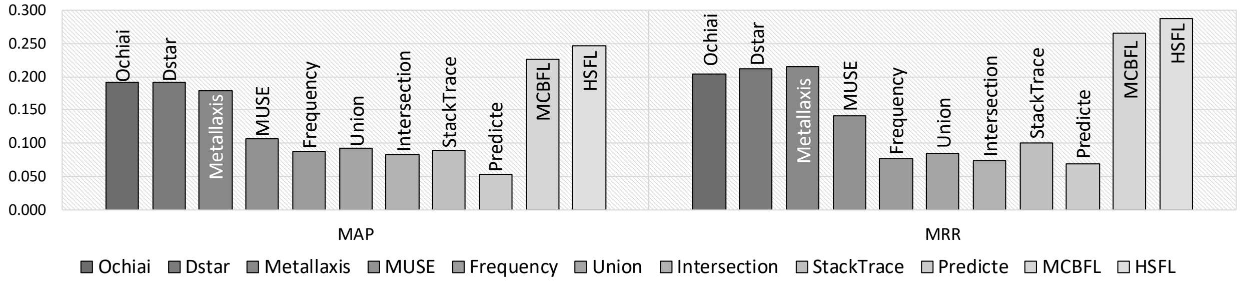


Fig. 16: The Performance of Different Families of Fault Localization Techniques in terms of MAP and MRR.

Rank: Top-N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
HSFL	64	99	116	133	146	155	158	166	173	177	186	187	192	194	195	196	198	198	200	205
Ochiai	37	66	87	98	107	119	124	132	138	148	153	156	161	164	166	172	173	177	179	183
Dstar	38	68	88	98	106	118	122	131	137	148	154	158	162	164	166	171	172	175	177	181
Metallaxis	46	68	86	103	111	117	121	124	135	141	145	150	151	155	158	160	166	171	171	173
MUSE	27	43	56	65	75	80	82	84	90	96	103	107	111	116	120	123	127	129	132	133
Union	8	22	34	44	53	58	64	66	72	75	76	79	81	84	90	93	98	100	100	101
Frequency	7	20	31	39	47	51	60	63	68	70	71	73	76	79	85	88	94	95	99	99
Intersection	7	19	31	39	46	50	54	56	61	64	66	67	69	72	78	81	86	88	89	90
StackTrace	23	31	36	42	46	49	49	49	51	59	64	66	70	72	74	75	77	78	80	81
Predicte	13	18	21	29	30	34	36	36	39	44	51	53	57	61	65	66	68	71	72	75
MCBFL	59	93	107	119	130	136	144	151	160	164	170	175	176	180	184	186	188	193	195	197

TABLE 6: The Performance of Different Families of Fault Localization Techniques in terms of Top-N.

These results indicate that our proposed Histrum model complements well to existing techniques. Not only can it boost the performance of SBFL, but also the other families of FL techniques. Specifically, combining Histrum with hybrid techniques is able to achieve the optimum performance.

5.7 RQ7:Integrating with Learning-to-Rank Techniques

Recently, Zou *et al.* proposed to combine all families of FL techniques via learning-to-rank techniques [55]. Specifically, it treats the suspicious score generated by each FL technique as an individual feature, and then leverages *rankSVM* [67] to learn the optimum model based on a separated training dataset. In total, the results of the following 10 different FL techniques have been selected as features in their experiments:

techniques selected as individual features : *Ochiai, DStar, Metallaxis, MUSE, Union, Intersection, Frequency, StackTrace, PredicateSwitching, BugSpots.*

The details of these techniques have been well explained in Table 3. Ten-fold validation has then been adopted to evaluate the performance of the learned model.

To answer this RQ, we compare the original results using the above 10 techniques and that after integrating HSFL, in which case, the results of HSFL have been considered as the 11th feature. The experimental results show that the MAP and MRR can be improved by 3.7% and 5.4% respectively, and the results in terms of Top-N are displayed in Figure 17. In order for comparison, we also display the results of HSFL and MCBFL since it achieves the optimum performance among all the baselines as displayed in Table 7. In summary, the performance of the learning-to-rank technique has been improved after incorporating our proposed approach as an individual feature.

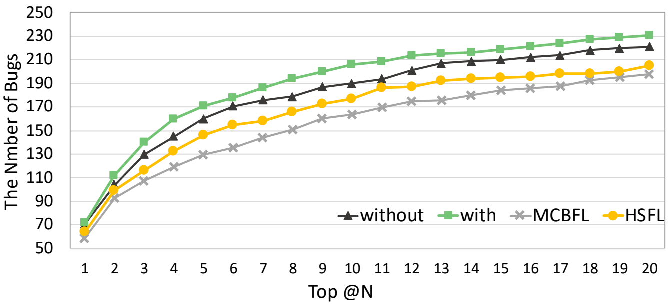


Fig. 17: Integrating with Learning to Rank Technique. 'with' Denotes the Results Obtained with HSFL as a Feature while 'without' Represents the Results without Considering HSFL.

6 Discussions

6.1 Effectiveness of Identifying Bug-Inducing Commits

Our strategy to identify the bug-inducing commits is adopted by existing works [25] and also those developers from open source projects (see Section 3.1). Actually, the detected bug-inducing commits identified by this strategy can be classified into the three types as mentioned in Section 3.1: 1) the bug-inducing commit is precisely identified and it is the initial commit of the buggy source file (i.e., denoted as *initial commit*); 2) the bug-inducing commit is precisely identified and it is one of the subsequent commits made to the buggy source file (i.e., denoted as *subsequent commit*); or 3) a set of likely-commits are identified as the approximation of bug-inducing commits since it is hard to identify the precise one. Figure 18 shows the distribution ratios of these three different types of inducing commits for the five subjects. The majority of the bug-inducing commits identified are the initial commits, which account for 59.7% of the bugs. 26.6% of the bug-inducing commits identified

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2948158, IEEE Transactions on Software Engineering

TABLE 7: Performance of Different Families of Fault Localization Techniques and Their Combinations with Our Proposed Histrum Model. 🟩 Denotes the Best Performance in terms of Different Metrics. 🟨 Denotes the Performance of HSFL. 'FL' Denotes the Performance of a Original FL Technique without Histrum. 'Boost' Denotes the Performance after Combining our Histrum Model.

Family	Technique	Project	MAP		MRR		Top@1		Top@5		Top@10		Top@20	
			FL	Boost	FL	Boost	FL	Boost	FL	Boost	FL	Boost	FL	Boost
SBFL	Ochiai	Chart	0.182	0.286	0.173	0.336	1	4	7	14	14	17	16	17
		Time	0.201	0.302	0.199	0.365	1	6	13	16	15	17	17	18
		Closure	0.122	0.170	0.148	0.225	11	22	26	38	38	43	51	53
		Math	0.245	0.285	0.266	0.327	19	21	35	49	47	61	59	70
		Lang	0.248	0.299	0.236	0.303	4	11	28	29	36	39	42	47
		Ave/Sum	**0.192**	**0.246**	**0.205**	**0.288**	**36**	**64**	**109**	**146**	**150**	**177**	**185**	**205**
	Dstar	Chart	0.197	0.234	0.197	0.259	2	3	7	10	13	14	14	14
		Time	0.207	0.302	0.210	0.357	1	5	13	17	15	18	18	19
		Closure	0.133	0.139	0.170	0.178	16	18	27	29	39	37	48	45
		Math	0.226	0.267	0.253	0.305	18	19	32	47	44	60	56	69
		Lang	0.248	0.303	0.236	0.318	4	11	29	32	38	41	44	51
		Ave/Sum	**0.192**	**0.226**	**0.212**	**0.261**	**41**	**56**	**108**	**135**	**149**	**170**	**180**	**198**
MBFL	Metallaxis	Chart	0.183	0.253	0.176	0.305	2	5	7	9	10	14	13	17
		Time	0.173	0.245	0.280	0.378	4	7	11	14	15	17	20	19
		Closure	0.070	0.122	0.089	0.157	7	14	16	28	23	34	26	38
		Math	0.236	0.282	0.265	0.330	18	22	40	52	47	58	63	67
		Lang	0.310	0.357	0.379	0.436	15	19	37	41	46	47	51	52
		Ave/Sum	**0.179**	**0.231**	**0.215**	**0.287**	**46**	**67**	**111**	**144**	**141**	**170**	**173**	**193**
	MUSE	Chart	0.130	0.220	0.157	0.321	2	6	6	10	8	13	10	17
		Time	0.052	0.154	0.056	0.206	0	3	4	10	5	12	7	12
		Closure	0.061	0.101	0.081	0.139	6	15	16	21	19	26	27	36
		Math	0.132	0.206	0.158	0.237	7	12	28	39	37	51	50	64
		Lang	0.179	0.248	0.271	0.340	12	18	21	24	27	35	39	43
		Ave/Sum	**0.108**	**0.172**	**0.142**	**0.223**	**27**	**54**	**75**	**104**	**96**	**137**	**133**	**172**
Slicing	Frequency	Chart	0.160	0.220	0.146	0.247	1	4	6	9	10	11	13	12
		Time	0.050	0.190	0.044	0.254	0	5	2	9	2	9	5	12
		Closure	0.003	0.045	0.002	0.056	0	5	0	10	0	13	1	18
		Math	0.126	0.179	0.111	0.178	4	7	19	30	26	38	35	53
		Lang	0.189	0.248	0.166	0.244	2	8	20	24	32	35	45	47
		Ave/Sum	**0.088**	**0.146**	**0.078**	**0.155**	**7**	**29**	**47**	**82**	**70**	**106**	**99**	**142**
	Union	Chart	0.159	0.226	0.146	0.254	1	4	6	9	10	11	12	13
		Time	0.052	0.195	0.046	0.257	0	5	2	9	2	9	4	12
		Closure	0.003	0.045	0.002	0.056	0	5	0	10	0	13	1	18
		Math	0.131	0.182	0.119	0.179	4	6	22	32	29	39	39	56
		Lang	0.202	0.256	0.188	0.260	3	9	23	26	34	37	45	48
		Ave/Sum	**0.092**	**0.149**	**0.084**	**0.159**	**8**	**29**	**53**	**86**	**75**	**109**	**101**	**147**
	Intersection	Chart	0.147	0.218	0.136	0.249	1	4	6	9	8	11	10	12
		Time	0.049	0.189	0.042	0.237	0	4	2	9	2	9	4	12
		Closure	0.003	0.045	0.002	0.056	0	5	0	10	0	13	1	18
		Math	0.117	0.166	0.107	0.161	4	6	18	27	25	36	35	53
		Lang	0.178	0.234	0.159	0.232	2	8	20	23	29	32	40	44
		Ave/Sum	**0.083**	**0.139**	**0.074**	**0.147**	**7**	**27**	**46**	**78**	**64**	**101**	**90**	**139**
Stack Trace	StackTrace	Chart	0.147	0.227	0.165	0.263	2	3	6	9	8	13	8	15
		Time	0.033	0.157	0.043	0.227	0	3	4	11	4	11	5	12
		Closure	0.023	0.065	0.028	0.080	3	8	4	14	4	15	5	20
		Math	0.099	0.153	0.105	0.169	6	8	17	27	21	37	30	53
		Lang	0.208	0.274	0.238	0.326	12	18	15	22	22	32	33	39
		Ave/Sum	**0.089**	**0.148**	**0.100**	**0.176**	**23**	**40**	**46**	**83**	**59**	**108**	**81**	**139**
Predicte Switching	PredicteSwitching	Chart	0.058	0.130	0.059	0.162	0	2	2	5	4	10	4	12
		Time	0.025	0.146	0.048	0.214	1	4	2	8	2	8	2	8
		Closure	0.024	0.064	0.045	0.087	5	8	7	15	8	18	11	25
		Math	0.062	0.130	0.078	0.146	4	7	12	24	15	33	26	49
		Lang	0.112	0.196	0.113	0.219	3	10	7	17	15	27	32	38
		Ave/Sum	**0.054**	**0.118**	**0.069**	**0.143**	**13**	**31**	**30**	**69**	**44**	**96**	**75**	**132**
Hybrid	MCBFL	Chart	0.208	0.311	0.206	0.350	2	5	9	13	14	18	15	18
		Time	0.229	0.315	0.308	0.409	3	6	14	17	16	19	18	19
		Closure	0.135	0.194	0.170	0.234	15	24	28	37	36	43	48	48
		Math	0.286	0.330	0.325	0.388	24	28	45	56	54	64	65	71
		Lang	0.324	0.362	0.372	0.406	15	17	34	39	44	47	52	55
		Ave/Sum	**0.227**	**0.283**	**0.266**	**0.333**	**59**	**80**	**130**	**162**	**164**	**191**	**198**	**211**

are the subsequent commits. For a small part of bugs (i.e., 13.7%), HSFL can only identify a set of likely-inducing commits. For project Time, the ratio of likely-inducing commits is extremely higher than the other projects. It is because that most of Time's test cases require extra classes (e.g., `DateTime`) to create time objects for testing. Those tests can not be successfully run on old versions since the required extra classes have not been checked in to the system. In this case, we cannot precisely identify the bug-inducing commit and thus use likely-inducing commits instead.

We further examined those bugs whose bug-inducing commits are identified as Type 1 or Type 2, we found that 95.8% of those bugs' root causes (i.e., fixing statements) have overlap with the statements modified by the bug-inducing commits identified by this strategy. This demonstrates that the precision of such strategy is high. Such strategy might not be able to retrieve all the bug-inducing commits for a bug if there are multiple since the bug-revealing tests might not be complete. However, HSFL's goal is not to retrieve all of them. As long as the identified bug-inducing commits are precise, HSFL is able to improve the performance of fault localization. The improvements shown in our evaluation confirmed this point.

6.2 Effects of Different Kinds of Bug-Inducing Commits

Different types of bug-inducing commits will affect the performance of HSFL. Figure 19 shows HSFL's performance based on different types of the identified bug-inducing commits. For subsequent commits, HSFL achieves the optimum performance with an average of MAP of 0.316 and MRR of 0.377. Compared with SBFL, it achieves the improvements of 88.8% and 117.4% for MAP and MRR respectively. For the type of initial commits, HSFL is able to achieve an average improvements of 11.5% and 21.0% for MAP and MRR respectively. However, there are no significant improvements for the type of likely inducing commits. Such differences are caused by the different sizes (i.e., modified number of statements) of the bug-inducing commits. Histrum is built on those modified statements of the bug-inducing commits. Therefore, the larger size of the bug-inducing commit, the more noises (i.e., those non-buggy statements irrelevant to the bug) it contains, and thus more difficult for HSFL to eliminate the noises and distill the root causes. Figure 20 shows the sizes of these three types of bug-inducing commits in terms of the number of modified statements. The median number of modified statements is 102 for subsequent commits of the five subjects, and the number for initial commits is $1,575$. This number is even larger for likely-inducing commits, which is around 10^5. As a result, HSFL performs the best for subsequent commits and worst for likely-inducing commits. Note that, If the identified bug-inducing commits are a set of likely-inducing commits, we aggregate all the modified statements over all these likely-inducing commits. It is common that the likely-inducing commits include the initial commit of the code base with lots of source files checked in to the project. Therefore, it is not surprising that the likely-inducing commits contain the largest number of modified statements.

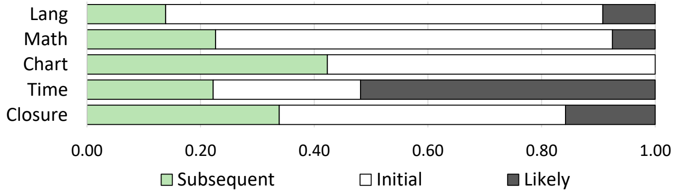
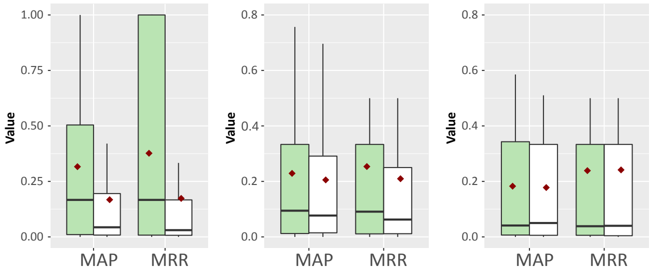


Fig. 18: Ratios of Different Types of Inducing Commits

Actually, *delta-debugging* [68] can be leveraged to help us identify the minimum set of changes that cause the bug in those large initial commits. However, it is extremely time-consuming. We leave the exploration of leveraging delta-debugging in HSFL as our future work.



(a) Subsequent (b) Initial (c) Likely

Fig. 19: Performance of HSFL Evaluated on Different Types of Inducing Commits. The Green Bar 🟩 Denotes HSFL and The White Bar Denotes SBFL

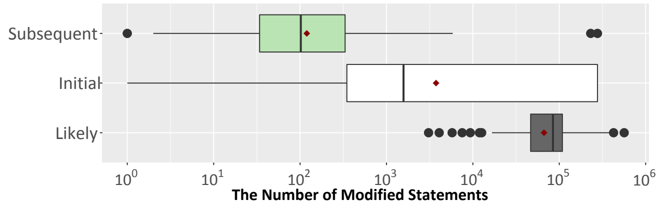


Fig. 20: The Sizes of Different Inducing Commits. The Size is Measured by the Number of Modified Statements

These results indicate that HSFL is suggested to be applied to locate buggy statements for regression bugs whose bug-inducing commits can be precisely identified as "subsequent" commits. However, it is not suggested to be applied when only a range of likely-inducing commits can be identified.

6.3 Effects of Multiple Faults

Large real-world programs, such as those subjects in the Defects4J dataset, always contain multiple faults at the same time, some of which might influence with each other. Although it is often the case, a recent study [11] indicate that there is no need to correct for this when performing fault localization, as long as the failing tests only reveal one defect at a time. Despite that, we are still curious about whether the performance of HSFL will be affected, when multiple faults coexist in the same program. Therefore, in this section, we investigate the effects of multiple faults.

We have extended the subject programs in the Defects4J dataset to extract programs with multiple faults following the practice of an existing study [35]. Be noted that we only investigate the effects of two faults at the same time in this study, and the investigation on a larger cardinality of faults is left as our future work. Actually, different faulty programs in Defects4J of a same project are different versions of the project. Therefore, to create a subject with two faults, we check whether any of two faults in the Defects4J dataset coexist in the same version of a same project (both of the faults have been introduced but have not been fixed). If so, we include it in our experimental subject. For instance, in the version before commit `#a92450e` of project Time, both bug `Time-19` and `Time-20` coexist in the system. This process was conducted manually. In total, we extracted 309 versions that contain two observable distinct faults. To evaluate the performance of HSFL under such scenarios, we combined the bug-revealing tests of both of these two bugs (suppose we cannot distinguish between the bug-revealing tests of these bugs). Then, we run our tool HSFL to see how well can it rank all the buggy statements. Finally, we compare the performance of HSFL under such scenarios with that obtained via running HSFL for each of the bug individually, in terms of MAP and MRR.

Figure 21 shows the results of the extracted 309 versions. When locating a single fault at a time, HSFL achieves an average MAP of 0.244 and MRR of 0.281. When locating multiple faults (two faults in our experiment) at a time, HSFL achieves an average MAP of 0.239 and MRR of 0.268. On average, the MAP has been decreased by 1.19% and that value is 4.40% for MRR when working on a scenario containing multiple faults. However, no significant differences have been observed (*p-value* ≥ 0.05) for the performance of HSFL under these two experimental settings as shown in Figure 21. Such results indicate that our proposed technique is able to handle multiple faults effectively and it does not lead to significant deterioration in its performance.

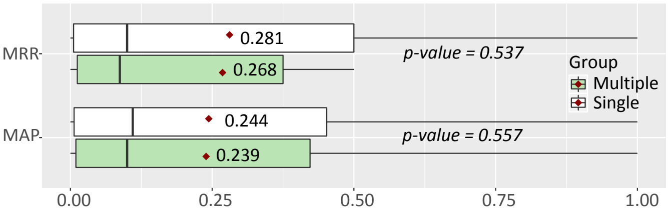


Fig. 21: The Performance of HSFL in Terms of MAP and MRR in the Presence of Multiple Faults

6.4 Overhead of HSFL

Table 8 shows the overheads of HSFL in processing a bug. In the table, each column represents different subjects while each row represents the different steps in HSFL. Specifically, step 1 refers to identifying bug-inducing commits; step 2 refers to constructing historical spectrum and step 3 refers to ranking suspicious elements. On average, it takes HSFL less than three minutes in total to obtain the final rankings of all the suspicious statements. This indicates that HSFL is practical in locating faults for real-world projects. The major overhead comes from the first step to identify bug-inducing commits. Recompiling the project's previous versions and rerunning the bug-revealing tests on those versions contribute to overall costs of this step. Therefore, the size of the project (i.e., number of source files) and the number of historical commits (since we need to search among the whole history using binary search) affect the time cost of this step significantly. The five projects from the DEFECTS4J benchmark are not large in their scales. A version of these projects can be compiled successfully in around 10 seconds. Therefore, the overhead of HSFL is not high for these projects. It will take HSFL longer time to identify the bug-inducing commits for projects with larger sizes. However, identifying bug-inducing commits is a common procedure for debugging in practice as confirmed by our observations in Section 2. Therefore, bug-inducing commits can be directly leveraged by HSFL once it has been identified by developers. Actually, similar to static analysis, which usually takes quite a long time, a natural fit of our approach to the development cycles is the nightly build cycle [69], [70]. A typical nightly build and test cycle takes 5-10 hours. Therefore, it is applicable to deploy our approach.

TABLE 8: HSFL Overheads in Seconds

	Lang	Math	Chart	Time	Closure	Average
Step 1	45.46	76.56	39.23	157.96	147.84	93.41
Step 2	3.57	31.91	4.53	32.20	182.80	51.00
Step 3	0.01	0.01	0.01	0.01	0.01	0.01

Step 1 refers to identifying bug-inducing commits;
Step 2 refers to constructing the historical spectrum
Step 3 refers to ranking suspicious elements.

6.5 Threats to Validity

One potential threat to validity is the generality of the projects used in our evaluation, which means that our experimental results may not be generalized to other dataset. Specifically, the distributions of different types of bug-inducing commits for different projects might be divergent as shown in Figure 18, and this will affect the final performance of HSFL. However, real-world open-source projects with different characteristics used in our evaluation, provided by benchmark DEFECTS4J, may at least partially mitigate the risk of over-generalization. Besides, for those projects with different distributions of inducing commits (e.g., likely-inducing commits ranging from 0.0% to 51.9%), HSFL can achieve significantly better results compared with SBFL on average for all of them (as shown in Sections 5.1 and 6.2). This reflects the generality of HSFL in improving the effectiveness of existing FL techniques. Evaluating HSFL on more subjects (e.g., projects from industry) and other languages is left for our future work.

Another threat is that our survey presented in Section 2.1 was only conducted at Microsoft, the findings of which might not be generalized to other companies or the open source communities. However, the empirical survey is not the major contribution of this study. The results of the empirical survey demonstrated the usefulness of version histories, especially the bug-inducing commits, in debugging activities. This inspired us to leverage the version history in fault localization. Such findings have also been echoed by the observations from the open source communities as discussion in Section 2.1. Specifically, developers from the open source community also often look for the information of bug-inducing commits when debugging. Such observations reflect the generality of our empirical survey conducted at Microsoft.

7 Related Work

7.1 Automated Fault Localization

Various automated fault localization techniques have been proposed [2], such as spectrum-based techniques (e.g., [12], [14], [15], [71]), mutation-based techniques (e.g., [3], [57], [72]), slice-based techniques (e.g., [73], [74]), machine-learning based techniques (e.g., [75]), program-state based techniques (e.g., [60]), data-augmented (e.g., [76]), feedback-based (e.g., [77]), and qualitative reasoning-based techniques (e.g., [78]) , which are the most related techniques with our proposed technique. Spectrum-based techniques leverage the test coverage information obtained via executing the associated test suite to rank suspicious code elements. Substantial existing SBFL techniques focus on refining fault locating formulae to improve their effectiveness [30], [32], [33], [34], [35]. Mutation-based techniques (e.g., [3], [57], [72]) utilize the test results obtained via mutating the program to improve the effectiveness of fault localization. Slice-based techniques (e.g., [73], [74]) leverage static or dynamic program dependencies to rank suspicious code elements. Machine-learning based techniques combine different FL techniques with machine learning techniques (e.g., [75]). Data-augmented techniques utilize defect prediction, which is built based on version histories, to point out those code elements that are more likely to be buggy in fault localization [76]. Feedback-based approach is able to leverage past diagnosis experience with the aim to improve fault localization performance [77]. Qualitative reasoning-based techniques leverage qualitative reasoning to augment the spectrum information obtained in SBFL to boost the final performance [78]. Specifically, it qualitatively partitions system components, and treats each qualitative state as a new spectrum component to be used in locating faults [78]. We have compared HSFL with most of the above-mentioned techniques as presented in Section 5.5. Data-augmented technique, feedback-based techniques and qualitative reasoning-based technique are not included in our evaluation in Section 5.5, since they are not applicable to our experimental setting, which targets at locating bugs automatically at the fine granularity of the statement level. Data-augmented techniques work at the source file level [76]. This is because, those features, which are extracted (e.g., number of public methods, number of other classes referenced) to augment the spectrum information, are not applicable to measure a single statement in our setting. The feedback-back technique needs to select modeling variables, which is an essential step in the their approach but requires to be done manually [76]. Besides, the authors evaluated the feedback-based technique on artificial faults (i.e., synthesized or manually injected) [76], while our evaluations were performed on real faults extracted from real large-scale projects. Qualitative reasoning-based technique works at the method level [78], and it leverages the parameter and return values of a method to partition spectrum components. Unfortunately, such an approach (i.e., leveraging parameters and return values) cannot be applied to components such as statements. Although these approaches cannot be directly applied to compare with HSFL, we can still incorporate the insights of these approaches to advance the performance of HSFL. Therefore, we left it as our important future work.

There are also other work focused on preprocessing test cases to improve the effectiveness of SBFL [12], [21], [79], [80], [81], [82]. For instance, Xuan *et al.* [79] proposed test case purification to reduce failing test cases for better performance. It first produced a set of single-assertion failing test cases, and then removed the irrelevant statements through dynamic slicing in them. Finally, it applied traditional SBFL to rank suspicious statements. Other researches are proposed with the aim to improve the performance of fault localization in terms of other aspects. For instance, Zhang et al. proposed to differentiate the contributions of different test cases using the PageRank algorithm [12], which is used to recompute the spectrum information to improving the effectiveness of SBFL. Furthermore, other works integrated more information to SBFL besides test coverage to improve its effectiveness [13], [80], [83], [84], [85], [86], [87], [88]. Sohn and Yoo [13] introduced code and change metrics (e.g., size and age) that have been widely-used in defect prediction to SBFL to improve its effectiveness. More specifically, based on the suspiciousness values from traditional SBFL and these source-code metrics, they utilized learning-to-rank techniques to rank suspicious source methods. Lucia *et al.* also proposed *Fusion Localizer*, which leverage the diversity of existing different SBFL techniques to better localize bugs using data fusion techniques [71]. However, all these techniques focused on the target buggy version to improve FL effectiveness (i.e., building coverage based spectrum on this single version). Different from them, HSFL is the first one to leverage the information of version histories for better fault localization. In particular, HSFL builds spectrum along the version histories, which is orthogonal to the traditional execution-based spectrum on the buggy version. Our experimental results demonstrated that the Histrum and the coverage based spectrum complement well to each other.

7.2 Mining Version Histories

Identifying Bug-Inducing Commits: SZZ-algorithm [89], [90] identifies bug-inducing commits by *blaming* the changed lines in the bug-fixing commits. Essentially, it assumes that the lines changed the bug-fixing commits contain the fault statements, and leverages `git blame` to identify which commits changed these lines previously. However, the SZZ algorithm is not applicable in our application scenario since it assumes the information of bug-fixing commits (i.e., the buggy statements) is available. On the contrary, the goal of our approach HSFL is to identify those buggy statements. If the bug-fixing commits are available, in which case, the buggy locations of the bug are known, then there is no need to launch our approach for fault localization. As a result, we resort to software testing to identify the bug-inducing commits. Besides, it has been reported that the bug-inducing commits identified by SZZ is not precise recently [25], [91], [92]. Specifically, it is reported that the SZZ algorithm can only achieve an average recall of 68.7%. LOCUS is later pro-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2948158, IEEE Transactions on Software Engineering

posed to identify the bug-inducing commits based on bug-reports using information retrieval techniques [23]. Change-Locator identifies bug-inducing commits based a bucket of crash reports [24]. These techniques work in the case where the bug-revealing tests are not available or the efforts of running test suite is extremely high. Delta debugging is another related work that aims at identifying the minimum set of changes inducing a failure [68]. HSFL identifies bug-inducing commits precisely via running the bug-revealing tests on the complete version history using binary search. This strategy is also adopted by a recent work to identify regression bugs [25].

Tracing Source Code Evolution: Understanding the evolution of source code is important for developers. Girba *et al.*proposed a meta-model to represent the history of code entities at different levels of granularity (e.g., class, method) [93]. Zimmermann *et al.* proposed annotation graph to identify line changes across several versions. However, code differences between two versions are maintained by version control systems at the granularity of hunks (i.e., a block of code elements) instead of lines. Therefore, precisely tracking the mappings at the granularity of statement level is challenging. History slicing techniques are proposed to tackle this issue [29], [94], [95]. Specifically, it approaches the line mapping problem as the problem of finding the minimum matching of a weighted bipartite graph, and then leverages the Kuhn-Munkres algorithm [46] to find the optimum matching. The historical evolution of source code has also been leveraged to enhance the performance of automated program repair techniques [96]. HSFL leverages history slicing to construct the historical spectrum based on the identified bug-inducing commits. To the best of our knowledge, historical spectrum is novel to FL. It is another dimension information of spectrum, which complements well to different families of fault localization techniques.

8 Conclusion

We present a novel FL technique, HSFL, in this paper, which leverages the information of version histories in fault localization. The key novelty of HSFL, which allows it to locate more bugs compared with SBFL, is the historical spectrum (i.e., Histrum). Histrum is constructed by tracing the evolution of bug-inducing commits along version histories and is another dimension of spectrum orthogonal to the conventional coverage based spectrum used in SBFL. It reflects the root causes of bugs directly and breaks the tie issues of conventional SBFL significantly. We evaluate HSFL on the benchmark Defects4J, and the results show that HSFL outperforms SBFL significantly. Specifically, it locates and ranks the buggy statement at Top-1 for 77.8% more bugs compared with SBFL, and 33.9% more bugs for Top-5. Besides SBFL, our evaluation results also show that our proposed approach can outperform and boost the performance of other six families of FL techniques. In the future, we plan to design better techniques specific for our proposed novel model, Histrum, to further improve the effectiveness of HSFL.

Acknowledgment

Rongxin Wu is the corresponding author. We thank anonymous reviewers for their constructive comments. This work is supported by the Hong Kong RGC/GRF Grant 16202917, the Hong Kong PhD Fellowship Scheme and the National Natural Science Foundation of China (Grant No. 61902329).

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[2] W. E. Wong and V. Debroy, "A survey of software fault localization," *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, vol. 9, 2009.

[3] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 464–475.

[4] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 49–60.

[5] X. Xia, L. Bao, D. Lo, and S. Li, "" 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," 2016.

[6] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 165–176.

[7] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair." ICSE, 2018.

[8] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software quality journal*, vol. 21, no. 3, pp. 421–443, 2013.

[9] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 3–13.

[10] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 191–201.

[11] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 609–620.

[12] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 261–272.

[13] J. Sohn and S. Yoo, "Fluccs: using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

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

[15] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 378–381.

[16] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 224–238.

[17] A. Riboira, R. Abreu, and R. Rodrigues, "An opengl-based eclipse plug-in for visual debugging," in *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*. ACM, 2011, pp. 60–60.

[18] A. Riboira, R. Rodrigues, R. Abreu, and J. Campos, "Integrating interactive visualizations of automatic debugging techniques on an integrated development environment," *International Journal of Creative Interfaces and Computer Graphics (IJCICG)*, vol. 3, no. 2, pp. 42–59, 2012.

[19] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[20] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie, "How do assertions impact coverage-based test-suite reduction?" in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 418–423.

[21] Y. Yu, J. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 201–210.

[22] X. Xu, V. Debroy, W. Eric Wong, and D. Guo, "Ties within fault localization rankings: exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 06, pp. 803–827, 2011.

[23] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 262–273.

[24] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang, "Changelocator: locate crash-inducing changes based on crash reports," *Empirical Software Engineering*, pp. 1–35, 2017.

[25] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 105–115.

[26] "Gcc-79286," https://gcc.gnu.org/bugzilla/show_bug.cgi?id=79286, 2018, accessed: 2018-01-12.

[27] "Solr-2606," https://issues.apache.org/jira/browse/SOLR-2606, 2018, accessed: 2018-01-12.

[28] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 121–130.

[29] F. Servant and J. A. Jones, "History slicing: assisting code-evolution tasks," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 43.

[30] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[31] R. Just, D. Jalali, and M. D. Ernst, "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*. ACM, 2014, pp. 437–440.

[32] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 273–282.

[33] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[34] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, p. 11, 2011.

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

[36] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 51.

[37] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 118–121.

[38] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 308–318.

[39] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: improving cooperation between developers and users," in *Proceedings of the 2010 ACM conference on Computer supported cooperative work*. ACM, 2010, pp. 301–310.

[40] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 117–128.

[41] "Solr-2606," https://issues.apache.org/jira/browse/SOLR-8026, 2018, accessed: 2018-01-12.

[42] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 65–74.

[43] C. Couder, "Fighting regressions with git bisect," vol. 4, no. 5, 2008.

[44] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.

[45] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[46] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics (NRL)*, vol. 2, no. 1-2, pp. 83–97, 1955.

[47] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of ISSTA, Demonstration Track*, 2016.

[48] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," 2016.

[49] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 201–211.

[50] E. M. Voorhees *et al.*, "The trec-8 question answering track report." in *Trec*, vol. 99, 1999, pp. 77–82.

[51] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press, 2008, vol. 39.

[52] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT Press, 1999, vol. 999.

[53] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 314–324.

[54] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th international conference on Software engineering*. ACM, 1995, pp. 41–50.

[55] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[56] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "An empirical analysis of the influence of fault space on search-based automated program repair," *arXiv preprint arXiv:1707.05172*, 2017.

[57] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 153–162.

[58] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[59] H. Pan and E. H. Spafford, "Heuristics for automatic localization of software faults," *World Wide Web*, 1992.

[60] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 272–281.

[61] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[62] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bug-cache for inspections: hit or miss?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 322–331.

[63] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 191–200.

[64] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[65] C. Manning, P. Raghavan, and H. Schütze, "Introduction to information retrieval," *Natural Language Engineering*, vol. 16, no. 1, pp. 100–103, 2010.

[66] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 524–535.

[67] T.-M. Kuo, C.-P. Lee, and C.-J. Lin, "Large-scale kernel ranksvm," in *Proceedings of the 2014 SIAM international conference on data mining*. SIAM, 2014, pp. 812–820.

[68] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *ACM SIGSOFT Software engineering notes*, vol. 24, no. 6. Springer-Verlag, 1999, pp. 253–267.

[69] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 241–252.

[70] S. McPeak, C.-H. Gros, and M. K. Ramanathan, "Scalable and incremental software bug detection," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 554–564.

[71] D. Lo, X. Xia *et al.*, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 127–138.

[72] M. Delahaye, L. C. Briand, A. Gotlieb, and M. Petit, "μtil: Mutation-based statistical test inputs generation for automatic fault localization," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 197–206.

[73] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software: Practice and Experience*, vol. 23, no. 6, pp. 589–616, 1993.

[74] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 33–42.

[75] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *ISSRE*, 2007, pp. 137–146.

[76] A. Elmishali, R. Stern, and M. Kalech, "Data-augmented software diagnosis," in *Twenty-Eighth IAAI Conference*, 2016.

[77] N. Cardoso and R. Abreu, "A kernel density estimate-based approach to component goodness modeling," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[78] A. Perez, R. Abreu, and I. HASLab, "Leveraging qualitative reasoning to improve sfl."

[79] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 52–63.

[80] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei, "Test input reduction for result inspection to facilitate fault localization," *Automated software engineering*, vol. 17, no. 1, p. 5, 2010.

[81] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *TOSEM*, vol. 22, no. 3, p. 19, 2013.

[82] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 82–91.

[83] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 520–523.

[84] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 177–188.

[85] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 56–66.

[86] W. Masri, R. A. Assi, F. Zaraket, and N. Fatairi, "Enhancing fault localization via multivariate visualization," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 737–741.

[87] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM transactions on software engineering and methodology (TOSEM)*, vol. 23, no. 1, p. 8, 2014.

[88] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 181–190.

[89] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 81–90.

[90] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[91] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

[92] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 326–337.

[93] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *Journal of Software: Evolution and Process*, vol. 18, no. 3, pp. 207–236, 2006.

[94] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Precise semantic history slicing through dynamic delta refinement," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 495–506.

[95] F. Servant and J. A. Jones, "Fuzzy fine-grained code-history analysis," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 746–757.

[96] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.