Exploring and Exploiting the Correlations between Bug-Inducing and Bug-Fixing Commits

Ming Wen
The Hong Kong University of Science and Technology (HKUST)*
Hong Kong, China
mwenaa@cse.ust.hk

Rongxin Wu
Department of Cyber Space Security, Xiamen University
Xiamen, China
wurongxin@xmu.edu.cn

Yepang Liu
Shenzhen Key Laboratory of Computational Intelligence, SUSTech
Shenzhen, China
liuyp1@sustech.edu.cn

Yongqiang Tian
HKUST, Hong Kong, China
ytianas@cse.ust.hk

Xuan Xie
Sun Yat-sen University, China
xiex27@mail2.sysu.edu.cn

Shing-Chi Cheung
HKUST, Hong Kong, China
scc@cse.ust.hk

Zhendong Su
ETH Zurich, Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

Bug-inducing commits provide important information to understand when and how bugs were introduced. Therefore, they have been extensively investigated by existing studies and frequently leveraged to facilitate bug fixings in industrial practices.

Due to the importance of bug-inducing commits in software debugging, we are motivated to conduct the first systematic empirical study to explore the correlations between bug-inducing and bug-fixing commits in terms of code elements and modifications. To facilitate the study, we collected the inducing and fixing commits for 333 bugs from seven large open-source projects. The empirical findings reveal important and significant correlations between a bug's inducing and fixing commits. We further exploit the usefulness of such correlation findings from two aspects. First, they explain why the SZZ algorithm, the most widely-adopted approach to collecting bug-inducing commits, is imprecise. In view of SZZ's imprecision, we revisited the findings of previous studies based on SZZ, and found that 8 out of 10 previous findings are significantly affected by SZZ's imprecision. Second, they shed lights on the design of automated debugging techniques. For demonstration, we designed approaches that exploit the correlations with respect to statements and change actions. Our experiments on Defects4J show that our approaches can boost the performance of fault localization significantly and also advance existing APR techniques.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Empirical software validation*; *Software evolution.*

KEYWORDS

Empirical study, bug-inducing commits, fault localization and repair

ACM Reference Format:
Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung and Zhendong Su. 2019. Exploring and Exploiting the Correlations between Bug-Inducing and Bug-Fixing Commits. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338962

1 INTRODUCTION

Software evolves by means of code changes committed to fix bugs, introduce new features, refactor existing code and so on. Such committed code changes, however, can also induce new bugs [28, 31, 54, 69]. Commits that induce software bugs are known as *bug-inducing commits*, which embed important information about when and how bugs are introduced. Due to the importance of bug-inducing commits, they have been extensively studied by researchers. For example, various studies have been conducted to understand the characteristics of bug-inducing commits (e.g., [9, 12, 13, 17, 20, 21, 30, 46, 53]), predict buggy-prone commits during software evolutions (e.g., [10, 23, 28, 31, 33, 43, 52, 67]), and locate bug-inducing commits (e.g., [32, 54, 62, 64]), and so on. Bug-inducing commits are also leveraged to ease debugging in industrial practices. Particularly, we observe that developers from open-source projects often look for the information of bug-inducing commits when devising the fixing patches of a bug (see Section 2 for more details).

Exploring the correlations between bug-inducing commits and bug-fixing commits is important. The reasons are two-fold. First, existing studies [9, 12, 13, 17, 20, 21, 28, 30–32, 46, 53, 54, 57, 62, 64, 69] mostly leveraged the SZZ algorithm [54] and its variants [18, 19, 32] to infer bug-inducing commits based on bug-fixing commits, with an implicit assumption that bug-inducing and bug-fixing commits

* This work was conducted when the first author was a visiting student at SUSTech (Southern University of Science and Technology).

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338962

are highly correlated in the code hunks (e.g., modifying the same code elements). To ease presentation, we refer to these approaches [18, 19, 32, 54] as "SZZ" and the implicit assumption as "SZZ assumption". While the SZZ assumption plays a critical role in previous studies, it has not been systematically validated in practice. Unexamined assumptions can result in questionable research claims and wasted research effort [47]. Second, although bug-inducing commits provide contextual clues for debugging in practice (see Section 2), the way to leverage them effectively for debugging is not well studied. Understanding the correlations between bug-inducing and bug-fixing commits is a fundamental step towards this target, enabling us to design better automatic debugging techniques.

Although several existing studies [15, 62] have explored certain correlations between a bug's inducing commits and fixing commits, their findings mainly focused on the metadata (e.g., the authorship of a commit [62]) or the metrics of commits (e.g., the complexity of the committed code hunks [15]). Unfortunately, these findings are insufficient to answer the questions on the validity of the SZZ assumption and provide few guidance on the applications of bug-inducing commits in bug fixing. Therefore, in this work, we propose to study the correlations between bug-inducing and bug-fixing commits in terms of code elements and modifications. Specifically, we investigate the following research question:

RQ1: Do bug-inducing commits and bug-fixing commits modify the same code elements (e.g., source files, statements), and do they involve similar change actions?

Here, change actions refer to the code modifications made to a program such as *inserting an* `if` *statement*. Inferring change actions (a.k.a., *mutation operators*) is an important step in designing *automated program repair* (APR) techniques [26, 34, 41, 49, 61].

To answer RQ1, we created a benchmark dataset containing the information of bug-inducing commits and the associated bug-fixing commits. Specifically, we analyzed substantial bug reports from large open-source projects and manually recovered the links between the inducing and fixing commits for hundreds of bugs. Studying RQ1 offers benefits in two aspects.

First, as above-mentioned, many existing studies (e.g., [21, 62, 69]) leveraged SZZ to collect bug-inducing commits. Unfortunately, recent studies have pointed out that SZZ suffers from a low precision [15, 18]. However, neither of these studies explained why SZZ is imprecise nor investigated the effect of such imprecision on the findings of existing studies, because SZZ's underlying assumption has not been systematically validated. The answers to RQ1 together with our benchmark enable us to validate the SZZ assumption and further answer the following question.

RQ2: What are the major causes for the imprecision in SZZ? Does such imprecision result in a significant bias in the findings made by previous studies?

Second, the answers to RQ1 can shed lights on the design of better automated debugging techniques. For instance, if the statements modified by a bug's fixing commits and inducing commits are correlated, we can leverage a bug's inducing commits to better infer the program locations that need to be modified to fix faults (*fault localization*, or FL in abbreviation). If the change actions performed to fix a bug are correlated with those actions that introduced the bug, we can better infer the required mutation operators when devising automated program repair tools. To investigate the feasibility of these ideas, we further study the third research question:

RQ3: Can bug-inducing commits and our findings be leveraged to advance existing automated debugging techniques?

To answer RQ3, we recovered the bug-inducing commits of 91 bugs in Defects4J [27] via bisect testing on version histories following the strategy adopted by an existing study [15]. Driven by the answers of RQ1, we devise novel debugging approaches incorporating the information of bug-inducing commits and compare them with the state-of-the-art techniques.

Via studying the three RQs, we obtained several major findings:

(1) Software bugs can be fixed in a source file but introduced in another due to different reasons such as faulty configurations, incomplete changes and so on (Section 3.2.1).
(2) Around 64.6% of the statements modified by the bug-fixing commits were also modified by the associated bug-inducing commits, and the ratio can be enhanced to 71.1% if data-flow and control-flow analyses are performed (Section 3.2.1).
(3) The majority of the change actions that are required to fix bugs can be inferred via reverting those change actions made in the corresponding bug-inducing commits (Section 3.2.2).
(4) Previous findings have been significantly affected by the biased bug-inducing commits identified by SZZ. Specifically, significant differences and non-negligible effect sizes have been observed for 8 out of 10 previous findings (Section 4).
(5) The information of bug-inducing commits can boost the performance of automated debugging techniques significantly. Based on the evaluations on Defects4J, we found that our FL approach can significantly outperform existing ones (around 100% improvement in terms of MAP and MRR) (Section 5).

This paper's contributions and organization are as follows:

(1) We created a benchmark dataset containing the bug-inducing and bug-fixing commits for 333 bugs from seven large open-source projects. The dataset is publicly available and can facilitate future research (Section 2).
(2) We conducted empirical studies based on the collected dataset. The results reveal significant correlations between a bug's inducing and fixing commits (Section 3).
(3) We analyzed SZZ and revisited previous findings based on bug-inducing commits identified by SZZ (Section 4).
(4) We devised approaches exploiting the correlations revealed by our studies and evaluated them on Defects4J. Results show that these approaches are promising to advance existing automated debugging techniques (Section 5).

2 DATASET CONSTRUCTION

2.1 Data Collection

Characterizing and understanding the correlations between bug-inducing commits and bug-fixing commits require a vast collection of these commits for real bugs. However, the information of a bug's inducing commits is rarely available, and there is no large dataset in public. Böhme *et al.* [15] identified the bug-inducing commits via software testing for only 70 bugs from four subsystems (`make`, `grep`, `findutils` and `coreutils`) of GNU project. Due to the requirement of bug-revealing tests, their approach to collecting bug-inducing commits is difficult to generalize since bug-revealing

Table 1: The Benchmark Dataset Collected in This Study

Project	Description	#Stars	#Files	KLOC	#Changes	Major	Critical	Blocker	Minor	Trivial	#Bugs	#BIC	#BFC
Accumulo	Data Storage and Retrieval	431	2,078	661	9,741	10	4	14	4	1	33	53	36
Ambari	System Administrators for Hadoop	788	3,267	132	22,800	11	8	12	0	0	31	36	38
Lucene	Text Search Engine Library	1,477	7,776	375	29,608	42	6	9	12	0	69	144	71
Hadoop	Distributed System	6,185	10,892	555	18,032	29	11	10	1	2	53	57	53
JackRabbit	Content Repository Management	202	3,158	132	8,571	24	0	2	7	0	33	66	42
Oozie	Web Services for Hadoop	444	1,373	278	2,224	24	6	11	3	0	44	49	45
CoreBench	Projects from GNU Operating System	~	2,317	146	33,433	~	~	~	~	~	70	70	70
Summary		9,527	30,861	2,279	124,409	140	35	58	27	3	333	475	355

#BFC denotes the number of bug-fixing commits; #BIC denotes the number of bug-inducing commits; ~ means the data is not available

tests are rarely provided in bug reports. An alternative is to collect bug-inducing commits that are flagged by domain experts (i.e., the corresponding developers) as pointed out by a recent study [18]. Fortunately, we found that many developers of large open-source projects (e.g., those from the Apache ecosystem) often leave the messages of their debugging activities in bug reports, and those messages might contain the information of bug-inducing commits. We show two examples of such messages as follows:

"*I'm fairly certain this is caused by the enhancements made in SOLR-1297 to add sorting functions" [6]*

"*This is an unintended bug introduced by commit #bc56c03" [2]*

We also observed that developers often conduct software testing to find the bug-inducing commits.

"*The test passes before this commit (LUCENE-6758) and fails after"[7]*

Such information provided by the domain experts enables us to identify the bug-inducing commits precisely for software bugs, and thereby to create a ground-truth dataset. In this study, we adopted the following three steps to build our dataset.

In the first step, we analyzed bugs' associated reports, and collected those candidates of bugs that potentially contain the information of bug-inducing commits. Specifically, we collected a bug as a candidate if it satisfies one of the following two conditions. First, we checked whether a bug's report has the specific attribute which indicates that other issues introduced this bug. For example, JIRA records such information in the field of "*is broken by*" under the category of "*Issue Links*". Second, we examined the description and comments of the bug reports via keyword searching. Specifically, we used the following regular expression:

```
(started with|caused by|introduced by|commit).*?(r*(\w){6,41}|PROJECT-(\d){3,7})
```

By manually checking a portion of bug reports that are sampled randomly, we observed that developers from Apache often used keywords *"started with", "caused by", "commits"* and *"introduced by"* to deliver the information of bug-inducing commits. Expression `(r*(\w){6,41})` matches the hash ID of a commit directly. Expression `(PROJECT-(\d){3,7})` matches an issue ID where `PROJECT` refers to the concerned project. For example, `LUCENE-6758` is an issue ID for the project Lucene. A bug is collected as a candidate if its descriptions or comments match the regular expression.

In the second step, we manually examined all the candidates to ensure the data quality. We found that the candidates such identified in the first step might contain noises. For instance, a matched comment "*The problem is caused by stack overflow in SOLR-6213* [5]" might be a noise and thus has been filtered out. This is because we were unsure whether the bug was caused by the commits to fix `SOLR-6213` or it was just caused by the same stack overflow problem similar to issue `SOLR-6213`. Similar candidates were filtered out. Two PhD students were involved in the manual checking. A candidate bug was discarded, if any one of them was unsure that the matched texts indeed deliver the information of the commits that introduced the bug.

In the third step, we built the links between bugs and the associated bug-fixing commits following the practices adopted by existing studies [38, 60, 65]. This step served two purposes. First, we used these links to identify the bug-fixing commits for all the collected bugs. Second, since some bug reports may contain indirect information of the bug-inducing commits, i.e., a pointer to another issue whose fixing commits introduced this bug, we resolved such indirect information using the built links. Therefore, we built the links between issues and commits by matching the issue ID in the commits' log messages. A candidate was discarded if such links cannot be recovered via referring to the unified format `PROJECT-ISSUE` as mentioned before. Even if the link can be successfully recovered, the identified bug-fixing commits might still contain noises since the tangling issue of commits is prevalent as revealed by existing studies [24, 25]. For such cases, the code changes made in the fixing commits might not all be related to the bug, and such noises will prevent us from understanding the correlations between the inducing and fixing commits of a bug correctly. Therefore, we further manually checked each fixing commit via investigating its log messages to see if it serves for multiple purposes (a.k.a., *non-atomic*) besides fixing the bug. For instance, commit `#4f9c470` of Lucene serves to fix multiple issues (i.e., `LUCENE-4796` and `LUCENE-4373`). Commit `#5e0f6a4` of Accumulo serves seven agendas as indicated by its log messages. Candidates with non-atomic bug-fixing commits have been discarded to control the data quality of our empirical study.

Rigorous strategies have been adopted in these three steps to collect the information of bug-inducing and bug-fixing commits. The reason is that our goal in this study is to collect a bug's inducing and fixing commits with high confidence to avoid bias in our empirical studies. We did not intend to collect the bug-inducing commits for all bugs in this study. In total, we analyzed all the bug reports from six popular large open-source projects (i.e., Hadoop, Lucene, Ambari, JackRabbit, Accumulo and OOZIE) and successfully identified the bug-inducing commits for 263 bugs. We also analyzed the bug reports for other large-scale open-source projects such as Spark, Struts and Tomcat. However, the number of bug-inducing commits successfully identified after manually checking was less than 20 for each project. This falls short of the minimum of 20 instances for the statistically significance tests [16] in our subsequent empirical studies. Therefore, we did not include those projects in this study.

Note that our collected bugs contain but are not limited to ***regression*** bugs. For example, bug `SOLR-2606` is not a regression

bug, which is introduced by code changes made to add new features [6]. We also included the benchmark dataset collected by Böhme *et al.*[15], which contains the information of bug-fixing and bug-inducing commits for 70 bugs from the GNU project.

The descriptions of the selected seven projects, including 333 bugs in total, are listed in Table 1. These projects are popular (i.e., with over 200 stars in GitHub) and large-scale (i.e., over 1,000 source files and over 2,000 commits). Our collected dataset is publicly available[1]. Note that there might be multiple bug-inducing commits for a bug. For example, there are in total 57 bug-inducing commits for the 53 bugs of project Hadoop. Previous studies have also revealed that a bug might be jointly introduced by multiple changes [14].

2.2 Threats to Validity

Collection of bug-inducing commits: The major threat of this study comes from the dataset used in our empirical analysis. That is, whether the bug-inducing commits identified as oracles truly introduced the corresponding bugs and whether all the bug-inducing commits have been identified for a bug. We addressed this threat from two aspects. First, we collected the data of bug-inducing commits based on the knowledge from domain experts (i.e., the developers assigned to the bug). This data collection methodology has been suggested by a recent study [18]. Besides, each candidate was manually checked involving two graduate students and potential noises have been filtered out following strict rules. Second, we also included a benchmark dataset (i.e., CoreBench [15]) in our empirical study. The bug-inducing commits in CoreBench were collected via software testing, and thus were validated to be real bug-inducing commits. More importantly, the empirical analysis results derived from our collected dataset are consistent with those derived from CoreBench (see Section 3 and 4). The consistency confirms the high quality of our collected dataset.

Missing links of bug fixes: The links between bugs and the corresponding fixing commits need to be recovered in this study. The bias of such links have been pointed out and systematically evaluated by an earlier study [11]. To mitigate such threat, we chose those projects that are well-maintained and whose bug-tracking systems are maintained by JIRA. For these projects, developers often leave the issue ID following a unified format (e.g., `PROJECT-ISSUE`) in the log messages of a commit for the issues that it has resolved [18]. Those bugs without such links are excluded from our study.

3 EMPIRICAL INVESTIGATION

In this section, we investigate the correlations between bug-fixing commits and the bug-inducing commits at the code level. Specifically, we aim at answering the following two research questions:

RQ1#A: Do bug-fixing commits modify the source files and statements that are modified by bug-inducing commits?

Inferring the code elements (e.g., source files and statements) that are required to be modified in order to fix a bug is critical in debugging, especially for *fault localization* and *automated program repair* [26, 35, 45, 49, 61, 62, 66]. Existing studies often infer such code elements based on the information of a program's current version [61], such as the *test coverage* information collected via executing the test suite against the current program. As mentioned earlier in Section 2, developers often identify the information of bug-inducing commits during debugging. Such practice provides a new direction for debugging in terms of version histories other than a program's current version. Therefore, in this RQ, we investigate the correlation between bug-fixing commits and bug-inducing commits, in terms of the modified source files and statements.

RQ1#B: Do bug-fixing commits perform similar change actions compared with those by the bug-inducing commits?

Inferring the change actions (i.e., mutation operators) that are needed to fix a bug is critical in designing *automated program repair* techniques [26, 34, 41, 49, 61]. A change action is represented as a pair of ⟨Operation, Node⟩ [41, 61]. There are four kinds of operation, which are Change, Add, Delete and Move. The Node represents the type of the changed AST node. We keep all but those types referring to non-source code changes (e.g., modification of JavaDoc). In total, 84 out of 96 distinct node types defined by JDT are kept. The strategy widely adopted by existing studies to infer frequent change actions is to learn from substantial real bug fixes collected from open-source projects [41, 61]. In this RQ, we investigate whether such change actions can be directly inferred from a bug's inducing commits.

To ease our explanation, let b denote a bug collected in our dataset, F_b denote the set of the corresponding bug-fixing commits, and I_b denote the set of the corresponding bug-inducing commits.

3.1 Methodology and Measurement

Measuring File Coverage: To answer RQ1#A in terms of source files, we investigate how many source files that are modified by the bug-fixing commits are modified by the associated bug-inducing commits. Specifically, suppose S_{F_b} denotes all the source files modified by the bug b's fixing commits F_b, and S_{I_b} denotes those by b's inducing commits. We measure the file coverage as follows:

$$File\ Coverage(b) = |S_{I_b} \cap S_{F_b}| / |S_{F_b}| \quad (1)$$

Measuring Statement Coverage: To answer RQ1#A in terms of statements, we investigate how many statements that are modified by the bug-fixing commits are introduced by or evolved from the associated bug-inducing commits. However, the real bug-inducing commits I_b might be checked in a long time before the bug-fixing commits F_b, and thus the lines of source code might be misaligned between I_b and F_b. For example, a statement at line 90 modified by I_b might be evolved to line 98 in F_b. Therefore, directly checking the overlap of changed statements between I_b and F_b is inappropriate. We adopt the following strategy to resolve this challenge, which maps the code modified in the bug-inducing commits to the one in the version before the bug-fixing commits are submitted.

For each bug-inducing commit $i_b \in I_b$, suppose L_{i_b} is the set of all its modified statements, we track the evolution history of these statements from i_b to each bug-fixing commit $f_b \in F_b$ to see if these statements can be mapped to the statements modified by any of the bug-fixing commits. Let us further suppose the version history is $\langle v_1, ..., v_j, v_{j+1}, ..., v_n \rangle$, where v_1 is the version after bug-inducing commit i_b and v_n is the version before the fixing commit f_b. 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 retrieve the statement in v_{j+1} that is mapped from the statement s in v_j. To find the optimum mappings of $\mathcal{M}_{j \mapsto j+1}(s)$, we follow an existing work of history slicing [50] and approach it as the problem of finding the minimum

[1]Data available at: **https://github.com/justinwm/InduceBenchmark**

matching of a weighted bipartite graph. The weight between any two statements is computed as their *Levenshtein Edit Distance* [36]. Our ultimate goal is to obtain $\mathcal{M}_{1\mapsto N}(s)$, which finds the statement in v_n that is traced 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 $\mathcal{M}_{1\mapsto N}(s) = \mathcal{M}_{N-1\mapsto N} \circ \mathcal{M}_{N-2\mapsto N-1} \circ \ldots \circ \mathcal{M}_{1\mapsto 2}(s)$. Note that not all statements in v_1 can be mapped to v_n 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 modified statements by the bug-inducing commit to statements in the version before the bug-fixing commit is made. We then examine whether the mapped statements in v_n have been further modified by the subsequent bug-fixing commits.

For a bug b, we use L_{F_b} to denote all the modified statements by F_b, and L_{I_b} to denote all the statements introduced by I_b. We use L_{IF_b} to denote those statements mapped from the bug-inducing commits to the bug-fixing commits. Specifically, $L_{IF_b} = \{\mathcal{M}_{1\mapsto N}(s), \forall s \in L_{I_b}\}$. We then compare L_{F_b} with L_{IF_b} to see how much they overlap, which is denoted and computed as follows.

$$Statement\ Coverage(b) = |L_{IF_b} \cap L_{F_b}|/|L_{F_b}| \quad (2)$$

For those overlapped statements, we further examine whether these statements have been modified by any other commits. The reason is that we want to understand whether the bug-fixing statements are directly modified by those statements that introduced the bug. Specifically, we compute the direct coverage ratio, which is the ratio of the statements that are modified by the bug-fixing commits and can be ***directly*** traced from the statements introduced in bug-inducing commits, as follows:

$$Statement\ Direct\ Coverage(b) = |L_{DC_b}|/|L_{F_b}| \quad (3)$$

where $L_{DC_b} = \{s : s \in L_{IF_b} \cap L_{F_b}$ and s has not been modified by other commits between I_b and $F_b\}$.

Measuring Change Action Coverage: To answer RQ1#B, we leverage GumTree [22] to conduct change analysis for all commits in I_b and F_b. For each commit, we can obtain a set of change actions, each of which is denoted as a pair ⟨Operation, Node⟩ as mentioned before. We use A_{F_b} to denote all the change actions extracted from the bug b's fixing commits F_b, and A_{I_b} to denote the change actions extracted from b's inducing commits. We then measure the change action coverage as follows:

$$Action\ Coverage(b) = |A_{I_b} \cap A_{F_b}|/|A_{F_b}| \quad (4)$$

We also observe that a bug can be repaired via reverting the actions performed in the bug-inducing commits [3]. For instance, if the inducing commits inserted a statement, the action performed to fix the bug might be deleting the statement. Therefore, it motivates us to investigate whether the change actions performed in the bug-fixing commits can be covered by the ***inverse change actions*** applied in the inducing commits. In particular, the inverse of ⟨Insert, Node⟩ is ⟨Delete, Node⟩ and the inverse of ⟨Delete, Node⟩ is ⟨Insert, Node⟩ accordingly. As for operations of Update and Move, the inverse actions are themselves. Here, suppose the inverse actions of A_{I_b} is denoted as IA_{I_b}. We then measure the inverse coverage of change actions as follows:

$$Action\ Inverse\ Coverage(b) = |IA_{I_b} \cap A_{F_b}|/|A_{F_b}| \quad (5)$$

3.2 Empirical Results

3.2.1 RQ1#A: File and Statement Coverage. Figure 1 shows the results of file coverage for the seven subjects. On average, 73.2% of the source files modified to fix bugs have also been modified by the associated bug-inducing commits (ranging from 65.6% to 81.5% across different projects). This result suggests that the majority of bugs are fixed in the same source files where they are introduced. However, a non-negligible proportion of bugs still can be fixed in a source file but introduced in another. We further investigated those 26.8% of the cases to see why bugs are fixed and induced in different files, and made the following observations.

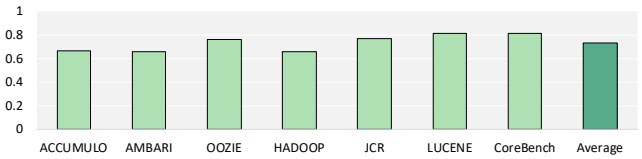

Figure 1: File Coverage between Bug-Fixing and Bug-Inducing Commits

Configuration/Log Related (17.9%): One reason that a bug's inducing and fixing commits modify different source files is that this bug relates to configurations or logging statements, an example of which is shown in Figure 2. In this example, the bug's inducing commit modified the functionalities in source file `NativeMap.java`. In the bug-fixing commits, the configuration related source file `ConfigurableMacBase.java` was modified to change settings related to `NativeMap`. Since the configuration related source files usually configure the settings of functionalities implemented in other source files, the inducing and fixing commits of a configuration related bug might modify different source files. This accounts for 17.9% of the cases.

Incomplete Changes (14.2%): Another important cause is related to *incomplete changes*, in which case developers made required changes (with respect to fix bugs, add features and etc.) but forgot to perform similar necessary changes elsewhere. Figure 3 shows an example of such case. The bug-inducing commit modified source file `AuthUrlClient.java`, which changed the encoding format to `UTF-8`. However, it forgot to perform similar changes in source file `DummyLogStreamingServlet.java`. The bug-fixing commit fixed this issue via changing the decoding format to `UTF-8`. The case of incomplete changes will be more prevalent if a system contains multiple modules since there might be similar functionalities implemented in different modules. An update in one module often requires an update accordingly to similar functionalities in other modules (e.g., see example of `ACCUMULO-2967` [1]).

For the rest of the cases (i.e., 67.9%), no unified patterns can be observed. However, for the majority of them (i.e., 73.6%, accounting for 50.0% with respect to the total number), the source files modified by a bug's fixing and inducing commit are ***directly*** dependent, even though they are different. For such cases, the classes where the bug was fixed have directly used those classes where the bug reside (e.g., superclass and subclass, imported classes and etc.).

Based on the above results, we can distill the following finding:

659a33e8e .../tserver/**NativeMap**.java *// Bug-Inducing Commit*

\+ String accumuloNativeLibDirs = System.getProperty("native.lib.path");

4cffe0290 .../functional/ConfigurableMacBase.java *// Bug-Fixing Commit*

\- cfg.setProperty(Property.TSERV_**NATIVEMAP**_ENABLED, TRUE.toString());

Figure 2: Bug-Inducing and Fixing Commit of ACCUMULO-4674

```
10549ef2  .../util/AuthUrlClient.java                    // Bug-Inducing Commit
-   stringBuilder.append(value);
+   stringBuilder.append(URLEncoder.encode(value,"UTF-8"));

b5524c90  .../service/DummyLogStreamingServlet.java      // Bug-Fixing Commit
-   lastQueryString = request.getQueryString();
+   lastQueryString = URLDecoder.decode(request.getQueryString(), "UTF-8");
```

Figure 3: Bug-Inducing and Fixing Commit of OOZIE-2320

> ***Finding #1.*** The majority of bugs (73.2%) are fixed in the same source files as where they were introduced. However, bugs can be fixed in a source file but introduced in another due to faulty configurations, incomplete changes and so on. The majority of source files modified by a bug's fixing and inducing commit have ***directly*** class dependency even though they are different.

Figure 4 shows the results in terms of statement coverage. For 64.6% of the statements modified by the bug-fixing commits on average (ranging from 58.7% to 68.8% across different projects), they were also modified by the associated bug-inducing commits. The direct coverage is 55.1% on average (ranging from 46.5% to 59.6% for different projects). These results indicate that, for 14.7% ((64.6%-55.1%)/64.6%) of the statements modified by both the bug-inducing and bug-fixing commits, they were also modified by other irrelevant commits in between. We further investigate whether the coverage can be enhanced if those statements affected by bug-inducing statements are considered. Specifically, we conducted data-flow and control-flow analysis to slice those dependent statements based on L_{IF_b}, and then augment L_{IF_b} with those sliced statements. Statement coverage is then recomputed based on Equation 2, and the results are shown in Figure 4 (displayed as *coverage+flow*). The coverage considering data-flow and control-flow reaches 71.1% on average, which has been improved by 10.2% compared with the original coverage. Two major reasons attribute to the incomplete coverage of statements. First, the source files are not covered for a portion of bugs. As a result, the modified statements cannot be covered. Second, we only conducted intra-procedural analysis to identify data and control dependent statements. However, the statements modified to fix the bugs might reside outside of the function where those bug-inducing statements are modified (e.g., in `JavaDoc`, other dependent functions or source files). For example, bug `HADOOP-13118` [4] was introduced by modifications to function `skipFully()`. The bug was fixed via updating statements of the associated `JavaDoc`. Such statements cannot be retrieved by intra-procedural data-flow or control-flow analysis.

Based on the these results, we can distill the following finding:

> ***Finding #2.*** For the statements modified by bug-fixing commits, 64.6% of them can be traced from the statements introduced by the corresponding bug-inducing commits on average. Such coverage can be improved to 71.1%, if we consider those statements affected by (i.e., control or data dependent on) bug-inducing statements.

Finding #1 and Finding #2 shed lights on understanding why the SZZ algorithm is imprecise [15, 18] (see Section 4). These findings also provide valuable guidance to advance existing automated FL techniques. For example, for FL at the source file level, we might need to refer to those source files that implement similar functionalities, relate to configurations, or those have direct class dependencies. For FL at the statement level, we can infer the fixing statements

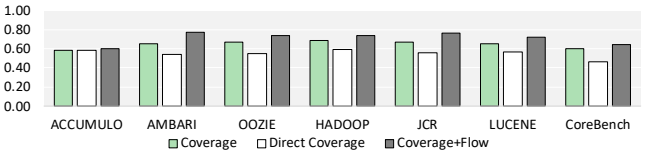

Figure 4: Statement Coverage (with Flow Analysis) and Direct Coverage between Bug-Inducing and Fixing Commits

largely from bug-inducing commits, and more statements can be inferred if we consider data or control dependencies.

3.2.2 RQ2#B: Change Action Coverage. Figure 5 shows the coverage and inverse coverage of change actions between a bug's inducing and fixing commits. The coverage ranges from 28.2% to 70.7% over different projects, with an average value of 43.1%. The inverse coverage of change actions ranges from 42.4% to 73.2% with an average value of 52.6%. These results indicate that around 40.0% of the change actions performed in the fixing commits can be inferred from the associated inducing commits, and that value is higher if we infer from bug-inducing commits' inverse change actions. We further investigated the coverage and inverse coverage for each type of change action, and Figure 6 shows the results of the top 100 frequently observed change actions averaged over the seven projects. We can see that for certain change actions (i.e., those displayed at the leftmost in Figure 6), the inverse coverages are higher than the direct coverage. This indicates that these change actions are more likely to be inferred by reverting the change actions observed in the bug-inducing commits. Examples of such change actions are ⟨Insert, Prefix_Expression⟩ and ⟨Insert, Array_Type⟩. On the contrary, for some other change actions (i.e., those displayed at the rightmost in Figure 6), the direct coverage is higher than the inverse coverage. Examples of such change actions are ⟨Insert, Initializer⟩ and ⟨Insert, Character_Literal⟩. Based on the above results, we can distill the following finding:

> ***Finding #3.*** Around 43.1% of the change actions used to fix bugs can be inferred from change actions that introduced the bug, and 52.6% of them can be inferred via reverting the change actions performed in bug-inducing commits. Besides, some change actions performed in bug-inducing commits are more likely to be directly applied to fix bugs while some others are more likely to be reverted.

Figure 5: Change Action Coverage Between Bug-Fixing and Bug-Inducing Commits. The Green Bar Denotes Coverage and the White Bar Denotes Inverse Coverage

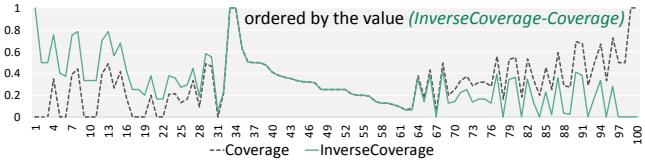

Figure 6: Coverage and Inverse Coverage of Change Actions

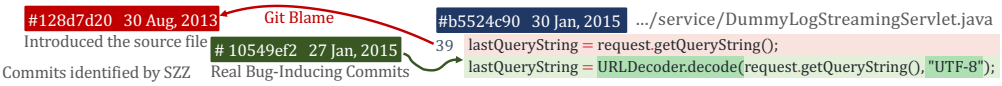

Figure 7: Identifying the Bug-Inducing Commits of OOZIE-2320 by SZZ

4 IMPLICATIONS ON EARLIER FINDINGS

In this section, we discuss the implications of our empirical findings on existing studies. Plenty of research studies have been proposed to characterize, and learn useful knowledge based on bug-inducing commits (e.g., [23, 62, 64]), since it helps understand how bugs were introduced and thus subsequently how to repair them. Collecting bug-inducing commits is the foundation of the aforementioned studies. The current widely adopted approach to collecting bug-inducing commits is SZZ [54] or its variants [18, 19, 32]. Unfortunately, recent studies [15, 18] have pointed out that SZZ suffers from the limitation of being imprecise. However, these two studies have neither explained why SZZ is imprecise nor investigated the effect of such imprecision on previous studies. Motivated by this, we aim at answering the following research questions:

RQ2#A: Can our findings explain why SZZ is imprecise?

RQ2#B: Will our findings affect previous studies which characterize bugs based on bug-inducing commits?

4.1 Implications on the SZZ Algorithm

The SZZ algorithm generally blames the statements modified by the bug-fixing commits, and assumes the latest commit that modified those statements introduced the bug. For the previous example displayed in Figure 3, SZZ will blame statement 39 modified in `#b5524c90` and assume `#128d7d20` is bug-inducing, since it last modified this statement as shown in Figure 7. However, the commit `#128d7d20` is the one that introduced source file `DummyLogStreamingServlet.java` 2 years ago, which is not the one that introduced the bug. The real bug-inducing commit `#10549ef2` is made only several days before the bug was fixed. This example demonstrates the limitation of SZZ, which motivates us the systematically evaluate the performance of SZZ. Specifically, we apply SZZ based on F_b for each bug as shown in Table 3, and denote the results as I_{SZZ_b}. We then compared I_{SZZ_b} with I_b to evaluate the performance of SZZ, and found that SZZ can retrieve 63.7% (ranging from 57.1% to 78.2% for different projects) of the bug-inducing commits (***recall***) while 68.7% (ranging from 62.4% to 72.0%) of the identified bug-inducing commits are not the real ones for the corresponding bug (***precision***). Note that there are five variants of SZZ as summarized by a recent study [18], and our results are based on AG-SZZ [32] since it achieves the optimum performance in terms of ***F-measure***. Besides, it was used by most of the earlier studies (e.g., [62, 67]). To explain such imprecision of SZZ, we investigated its working mechanisms, and found that SZZ is actually subject to the following two implicit assumptions:

Assumption 1. *The lines of code, which are modified by the commits with respect to fixing a bug, are the same as or evolved from the lines of code that are modified by the bug-inducing commits.*

Assumption 2. *For each line that is modified by the commits with respect to fixing a bug, the commit that last modified the line introduced this bug.*

If any of the two assumptions is invalid, SZZ cannot identify the bug-inducing commits correctly via blaming the statements modified by the bug-fixing commits as shown by the example displayed in Figure 7. Indicated by *file coverage* and *statement coverage*, we can see that 26.8% of the source files modified by the bug-fixing commits have not been modified by the associated bug-inducing commits on average, and that ratio is 35.4% at the statement level. Therefore, the first assumption is not hold for a non-negligible portion of cases. For such cases, the bug-inducing commits cannot be correctly identified if we only blame the modified statements in the bug-fixing commits. This explains the low recall of SZZ. As suggested by Finding #1, we can refer to those statements that are *data* or *control* dependent and those source files that are directly dependent to improve the recall. Indicated by *direct statement coverage*, we can see that, for those statements modified by the bug-fixing commits and also modified by the bug-inducing commits previously, a non-negligible portion of them (i.e., 14.7%) were last modified by other irrelevant commits instead of bug-inducing commits. Therefore, the validity of the second assumption is seriously affected. As a result, if we pick the last commit that modified the fixing statements as bug-inducing commits, as adopted by the SZZ algorithm, irrelevant commits might be identified as bug-inducing commits. This explains the low precision of SZZ. Based on these results, we answer RQ2#A via distilling the following finding:

Finding #4. Current SZZ implementations can only identify around 68.7% of the real bug-inducing commits (*Recall*), and 63.7% of the commits identified by them are not the real ones (*Precision*). The reason is that the implicit assumptions of the SZZ algorithm are violated by the insufficient *file coverage* and *statement direct coverage* between bug-inducing and bug-fixing commits.

4.2 Effects on Previous Findings

Plenty of researches have been conducted to understand the characteristics of software bugs based on bug-inducing commits. For example, *life span* of software bugs has been investigated [17, 30], which is measured from the time when the bugs were introduced to the time when they were fixed. Existing studies rely on the SZZ algorithm to identify the bug-inducing commits [17, 30]. However, such measurements can be biased if the bug-inducing commits identified by SZZ are imprecise. For instance, in the example as shown in Figure 7, the life span is nearly two years if the commit `#128d7d20` is identified as the bug-inducing commit. However, the life span of this bug is only three days since it got fixed ($30th$ Jan) soon after it was introduced ($27th$ Jan). Therefore, we are motivated to propose RQ2#B. To answer this question, we revisited previous findings based on the real bug-inducing commits I_b (i.e., the *Oracle* dataset) and the ones identified by AG-SZZ (denoted as I_{SZZ_b}).

Characteristics Selection: We choose the 10 most representative findings involving bug-inducing commits investigated by recent studies [9, 13, 17, 20, 21, 30, 46, 53] as shown in Table 2. In particular, the developer's *experience* is measured by the number of prior commits submitted [11]. The *time scatter* is the gap of the time

Table 2: Bias and Effect Sizes of the Findings Based on Bug-Inducing Commits Revealed by Earlier Studies

ID	Findings	ACCUMULO		AMBARI		LUCENE		HADOOP		JCR		OOZIE		CoreBench	
		p-value	effect	p-value	effect	p-value	effect	p-value	effect	p-value	effect	p-value	effect	p-value	effect
1^{L}	The number of bug-inducing commits per bug [18, 19]	**0.000**	0.554	**0.000**	1.117	**0.000**	0.901	**0.000**	1.138	**0.000**	0.796	**0.000**	1.236	**0.000**	0.654
2^{L}	The number of inducing source files per bug [28, 31, 67, 68]	**0.000**	1.087	**0.000**	0.711	**0.000**	0.779	**0.000**	0.615	**0.000**	0.762	**0.000**	1.097	**0.000**	0.755
3^{U}	The time of the bug inducing changes (in terms of the 24 hours) [21]	0.288	0.105	0.252	0.259	0.618	0.042	0.592	0.158	0.860	0.045	0.383	0.014	**0.000**	0.092
4^{U}	The time of the bug inducing changes (in terms of the 7 weekdays) [54]	0.658	0.098	0.434	0.124	0.727	0.048	0.696	0.062	0.746	0.060	**0.025**	0.350	0.243	0.010
5^{G}	The experiences of the developers who submitted the buggy commit [21, 28, 67, 68]	**0.000**	0.661	0.140	0.170	**0.000**	0.369	**0.037**	0.292	0.104	0.176	**0.002**	0.470	**0.032**	0.069
6^{L}	The life span of software bugs [17, 30]	**0.000**	1.512	**0.000**	1.240	**0.000**	1.434	**0.000**	1.544	**0.000**	1.249	**0.000**	1.509	**0.000**	1.067
7^{L}	The time scatter among bug-inducing commits for a bug [18]	**0.000**	1.594	**0.000**	1.596	**0.000**	1.493	**0.000**	1.503	**0.000**	1.426	**0.000**	1.571	**0.000**	1.212
8^{L}	The code churn of added lines of bug-inducing commits [28, 31, 67, 68]	**0.000**	0.368	0.270	0.238	**0.000**	0.251	**0.000**	0.237	**0.000**	0.350	**0.000**	0.470	0.266	0.190
9^{L}	The code churn of deleted lines of bug-inducing commits [28, 31, 67, 68]	**0.006**	0.343	0.252	0.243	**0.000**	0.272	**0.002**	0.391	**0.000**	0.276	**0.001**	0.301	**0.011**	0.244
10^{L}	The change entropy of the bug-inducing commits [28, 67, 68]	**0.000**	0.416	0.145	0.286	**0.000**	0.458	**0.002**	0.523	**0.000**	0.599	**0.000**	0.693	**0.022**	0.365

Value **0.000** indicates the *p-value* is less than 0.0001. Emphasis on Cohen's d indicates effect size: large ($|d| \geq 0.8$), medium ($0.5 \leq |d| < 0.8$), small ($0.2 \leq |d| < 0.5$) and negligible ($|d| < 0.2$)
Bold values indicate the difference is significant (*p-value*<0.05). Superscript **L** denotes the findings measured based on I_b are **Less** than S_b, **G** denotes **Greater**, and **U** denotes unequal distribution

between the earliest and latest bug-inducing commits for a same bug [18]. The *code churn* measures the number of added or deleted lines of codes of a commit [28, 31, 67, 68] and the *change entropy* measures the distributions of the modified code across each file in a commit [28, 67, 68].

Measurement: We investigated those selected findings based on I_b and I_{SZZ_b} separately, and then examined whether they are significantly different. Specifically, we used the Pearson's $\mathcal{X}^2$ Test [44] (applied to those characteristics with *categorical* values such as ID#3 and #4) and Mann-Whitney U-Test [39] (applied to characteristics other than ID#3 and #4 since they are *numerical* values) to test whether the differences are significant (*p-value*<0.05) as well as the Cohen's d *effect size* [29] to test whether the effects caused by such differences are non-negligible.

Result: Table 2 shows the statistical results. Among the 10 findings, no significant differences have been observed for two of them (ID#3 and ID#4), and the differences for these two characteristics are mostly negligible with respect to effect size. These two characteristics measure the time when bug-inducing commits are more frequently submitted (on Friday [54] and 0:00-4:00am [21]).

For the other eight findings, the differences are significant (*p-value*<0.05) for most of the cases, and most of the differences for the seven subjects are non-negligible with respect to effect size. Compared with the oracle data, AG-SZZ labels a significantly larger number of bug-inducing commits for each bug (ID#1). As a result, significantly more source files will be marked as bug-inducing (ID#2). This will hinder us correctly understand the percentage of buggy source files in a project [70], and further affect prioritizing software maintenance efforts. The experiences of developers who submitted the bug-inducing commits measured by I_b are significantly higher than those measured by S_{SZZ_b} (ID#5). As such, the earlier conclusion that junior developers are more likely to introduce bugs might be biased [21]. The life span of software bugs measured by S_{SZZ_b} is significantly longer than that measured by I_b (ID#6) and the effect sizes are all **large** for the seven projects. This can introduce bias in understanding the long-lived software bugs (i.e., those bugs whose life spans are longer than a year [48]). The practical impact of long-lived bugs is significant since experiencing the same failure repetitively over a year can be particularly frustrating to end-users. However, we observed that the proportion of long-lived bugs measured in terms of the oracle data is 11.4%, while the one measured by AG-SZZ is 69.2%. In other words, using AG-SZZ, 57.8% of the bugs are mistakenly regarded as long-lived. Such biased dataset will further lead to the bias of long-lived bugs' characteristics investigated by the previous study [48]. For example,

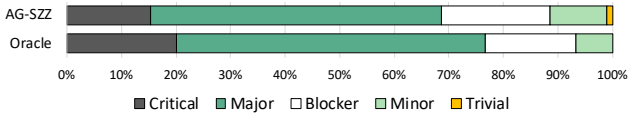

Figure 8: Comparison of the Severities of Long-Lived Bugs

as shown in Figure 8, over 76.7% of the long-lived bugs have high severity levels of "Major" or "Critical" measured based on I_b, while this ratio is only 68.7% based on the results of AG-SZZ. The characteristics in terms of *code churn* and *change entropy* are significantly affected with non-negligible effects (ID#8, #9 and #10), which will affect just-in-time quality assurance [28] since these are frequently used metrics to build prediction models.

We answer RQ2#B via distilling the following finding:

> ***Finding #5.*** The biased dataset of bug-inducing commits used in previous studies significantly affects their findings. Specifically, significant differences (*p-value*<0.05) and non-negligible effect sizes have been observed for 8 out of 10 previous findings.

5 IMPLICATIONS ON AUTOMATED DEBUGGING

As revealed in Section 2, developers in practice often look for bug-inducing commits to facilitate bug triaging and debugging. However, such information has never been leveraged in the design of automated debugging techniques. Therefore, we are motivated to study the following research question in this section:

RQ3: Can bug-inducing commits and our findings be leveraged to advance existing automated debugging techniques?

To facilitate this study, we selected the Defects4J [27] benchmark since it has been widely adopted in fault localization and automated program repair researches (e.g., [42, 45, 45, 51, 58, 59, 61, 62]). However, the information of bug-inducing commits for the bugs in Defects4J is not available and cannot be found in the associated bug reports neither. To identify bug-inducing commits, we conduct binary search, which executes the bug-revealing test cases provided by Defects4J, on the complete version history (automated by `git bisect`). The first commit on which the bug-revealing test cases start to fail is considered as the bug-inducing commit. This heuristic follows an existing study [15] and the debugging practice observed in open-source projects (e.g., see bug report of `SOLR-8026` [7]). In total, we have collected the bug-inducing commits for 91 out of the 357 bugs in Defects4J as shown in Table 3.

5.1 Implications on Fault Localization

Our Finding #2 indicates that the majority (i.e., 71.1%) of the bug-fixing statements are evolved from or dependent on the bug-inducing

Table 3: Subjects from the Defects4J Dataset

Subject	#Recovered	#Total	KLOC	Test KLOC	#Test Cases
Commons **Lang**	7	65	22	6	2,245
JFree**Chart**	9	26	96	50	2,205
Commons **Math**	24	106	85	19	3,602
Joda-**Time**	6	27	28	53	4,130
Closure Compiler	45	133	147	104	7,929
Total	**91**	357	378	232	20,111

statements. This observation suggests that it is possible to leverage such knowledge to advance existing spectrum-based fault localization (SBFL) techniques [45]. Current SBFL techniques typically leverage test coverage collected via executing test cases on the current version. On the contrary, bug-inducing commits provide the information from another dimension in terms of version histories. To investigate whether such information can boost the performance of SBFL, we devise the following approach inspired by Finding #2.

Methodology: The same as Section 3.1, for a bug b, we use L_{I_b} to denote the statements introduced by the bug-inducing commits, and L_{IF_b} to denote those statements on the buggy version that are evolved from L_{I_b}. We further conduct data-flow and control-flow analyses based on L_{IF_b}, and obtain a set of statements affected by them (denoted as AF_{IF_b}). Suppose $\mathcal{A}$ denotes the set of statements identified by SBFL with the suspiciousness score of $SBFL(s)$ for each statement s in $\mathcal{A}$. We adopt the approach specified in Equation 6 to compute the suspiciousness score of statements via exploiting the information of bug-inducing commits. Our insight is intuitive. If a statement identified by SBFL is evolved directly from the associated bug-inducing commits, we raise its suspiciousness score by 1.0. If it is affected by those statements (i.e., control or data-flow dependent) evolved from the bug-inducing commits, we raise its suspiciousness score by 0.5. Otherwise, we keep the original score as identified by SBFL. Be noted that the original scores denoted by $SBFL(s)$ have been normalized to the range of $[0, 1]$ before applying our strategy.

$$BOOST(s) = \begin{cases} SBFL(s) + 1.0 & s \in \mathcal{A} \wedge s \in L_{IF_b} \\ SBFL(s) + 0.5 & s \in \mathcal{A} \wedge s \notin L_{IF_b} \wedge s \in AL_{IF_b} \\ SBFL(s) & otherwise \end{cases} \quad (6)$$

We then use the boost model to rank all suspicious statements.

Results: Pearson *et al.* have evaluated multiple SBFL techniques on Defects4J recently [45]. They provided the oracle (i.e., the buggy statements) and the coverage spectrum for each bug. Therefore, we leverage these publicly available data to generate the results of SBFL. Specifically, we leverage the formula of Ochiai [8] since it has been reported to be the best formula for SBFL [45, 63]. We leverage two well-known metrics, namely MAP and MRR [40, 55, 62], to evaluate the performance of our boost model. Figure 9 shows the results of the 91 recovered bugs. Our boost model can achieve an average MAP of 0.324 and MRR of 0.379 (these results are weighted averages over the 95 bugs from five projects). The improvement of

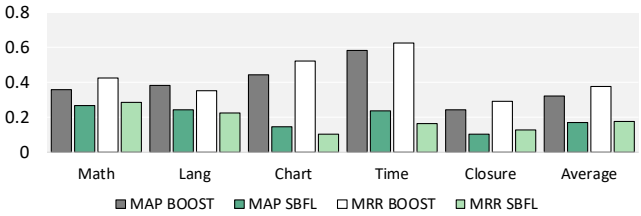

Figure 9: Comparing our Boost Model and SBFL

```
a4c526da  .../ jscomp/Compiler.java                    // Bug Inducing Commit
- 482  CompilerInput previous = inputsById.put(id, input);
+ 482  CompilerInput previous = putCompilerInput(id, input);
- 1272 if (options.dependencyOptions.needsManagement()) {
+ 1272 if (options.dependencyOptions.needsManagement() &&
+ 1273         !options.skipAllPasses &&
+ 1274         options.closurePass) {
0a670cb5  .../ jscomp/Compiler.java                    // Bug Fixing Commit
  1284 if (options.dependencyOptions.needsManagement() &&
- 1285         !options.skipAllPasses &&
  1286         options.closurePass) {
```

Figure 10: Bug-Inducing and Fixing Commit of Closure 31

MAP over the Ochiai technique is 90.8% on average (ranging from 33.0% to 210.9%). The improvement for MRR is 112.7% on average (ranging from 47.7% to 390.1%). These results are very promising, which indicates that we can identify the root causes of bugs more precisely via leveraging the information of bug-inducing commits than merely using SBFL. However, the statement coverage of bug-fixing statements is still only around 70.0% as indicated by Finding #2. In future, we plan to design more sophisticated approaches, such as considering dependent functions and classes as indicated by Finding #1 and Finding #2, to better locate bugs.

Based on the above results, we can distill the following finding:

Finding #6. The information of bug-inducing commits can boost the performance of automated FL significantly. Specifically, the MAP can be improved by 90.8% and MRR by 112.7%.

5.2 Implications on Automated Program Repair

Selecting appropriate change actions (a.k.a., mutation operators) is critical in designing effective APR techniques [41, 61]. Our Finding #3 indicates that the majority (i.e., 52.6%) of the change actions that are performed to fix bugs can be inferred via reverting those change actions performed in the corresponding bug-inducing commits. This finding can guide the search of mutation operators in designing APR. Figure 10 shows the bug fixing patch and the identified bug-inducing commit of bug Closure-31 from Defects4J. As we can see, code element "`!options.skipAllPasses &&`" at line 1285 is deleted to fix this bug, and this code element was inserted at line 1273 by the corresponding bug-inducing commit. Therefore, we can simply revert partial of the bug-inducing commit to fix this bug if the information of the bug-inducing commit can be identified. On the contrary, this bug cannot be repaired by the state-of-the-art APR techniques (i.e., CapGen[61], SimFix [26], ELIXIR [49], FixMiner [34] and ssFix [66] as reported by [37]). This example also demonstrates the new challenges when leveraging bug-inducing commits. As shown in Figure 10, the bug-inducing commit also introduced other change actions, such as replacing the method name at line 482, which is not required to be changed to fix the bug. Therefore, new searching strategies are required to be devised to identify those change actions that are needed to be reverted among all the ones that are performed by the bug-inducing commit. To achieve such a goal, we plan to further investigate the correlations between the change actions performed by the bug-inducing commit and those performed by the bug-fixing commit. For instance, we observe that change action ⟨Delete, Expression_Statement⟩ in bug-inducing commits is significantly correlated with change action ⟨Insert, Field_Declaration⟩ in bug-fixing commits. This suggests

that if a bug is introduced via deleting an *expression statement*, it is more likely to be fixed via inserting a *field declaration*. We plan to leverage such correlations to design more sophisticated strategies to search appropriate mutation operators for APR in the future. Based on the above example, we distill the following finding:

Finding #7. It is promising to leverage bug-inducing commits in designing APR techniques.

As revealed by Finding #6 and Finding #7, it is highly recommended to consider the information of bug-inducing commits, which can be obtained automatically via testing on version histories, when devising new automated FL and APR techniques.

6 RELATED WORK

Many researches have been conducted involving bug-inducing commits, which can be broadly classified into the following categories:

Characterizing bug-inducing commits. Śliwerski *et al.* [54] studied on which weekdays developers are more likely to submit bug-inducing commits. Bernardi *et al.* [13] investigated over 9,000 bugs from Eclipse and Mozilla, and found that developers who are more likely to introduce bugs seldom communicate with other developers through analyzing developers' social networks. Researchers are also interested in whether the experience of developers will affect their possibility of introducing software bugs [21, 46]. Eyolfson *et al.* [21] found that developers who make commits on a daily basis are less likely to introduce bugs, which indicates that daily-committing is a good developing practice. The life span of software bugs has also been investigated [17, 30]. For instance, Kim *et al.* [30] analyzed the duration between the check-in time of the bug-inducing commit and that of the bug-fixing commit, and found that the life span of bugs is usually 100 to 200 days. Based on this, Saha *et al.* [48] later investigated the characteristics of long-lived bugs (i.e., those bugs whose life spans are over a year). Bavota *et al.* [12] investigated when a refactoring code change will induce a bug. Recently, Asaduzzaman *et al.* [9] also investigated the characteristics of bug-inducing commits for Android systems.

Just-in-time quality assurance. Based on the characteristics of bug-inducing commits, many existing studies propose to predict whether a commit will introduce bugs at check-in time. Such a process is known as *just-in-time quality assurance* [28]. It enables developers to timely identify the introduced bugs if any, and prevents these bugs from spreading. Aversano *et al.* [10] proposed to learn from bug-inducing commits to prevent error-prone code. Specifically, they used a weighted term vector to represent a commit, where each term is extracted by considering a sequence of alphanumeric characters separated by non-alphanumeric characters. Machine learning algorithms (e.g., KNN and SVM) are then leveraged to predict whether a commit is error-prone. Kim *et al.* also proposed to predict whether a commit is bug-inducing [31]. Instead of extracting features from the tokens of a commit, they proposed to extract features from the log message, file names and the complexities of the changes made in the commit. Kamei *et al.* [28] later conducted a large-scale empirical study of just-in-time quality assurance. They extracted the following five categories of features from each commit: the diffusion of the commit, the size of the commit, the intention of the commit, the history of the files modified in the commit, and the experience of the developer who committed the changes. More recently, Yang *et al.* [67] leveraged deep learning techniques to predict whether a commit is error-prone. In addition to the techniques for single projects, cross-project just-in-time quality assurance has also been investigated by a recent study [23].

Debugging involving bug-inducing commits. Wen *et al.* [62] proposed a fault localization technique, which leverages information retrieval methods, to locate bug-inducing commits based on bug reports. Wu *et al.* [64] conducted an empirical study to understand the characteristics of the inducing commits for crashes, and then proposed an approach to locating crash-inducing commits automatically based on crash reports. Tan *et al.* [56] derived a set of code transformations obtained from the bug-inducing commits for 73 real regressions via manual inspection, and proposed *relifix* to repair regression bugs. The SZZ algorithm [54] and its variants [18, 19, 32] have been proposed to locate bug-inducing commits based on the code changes made by the bug-fixing commits. However, recent studies have pointed out that SZZ suffers from the problem of being imprecise [15, 18]. For instance, Böhme *et al.* [15] found that, for nearly one third of their studied bugs, SZZ cannot identify any real bug-inducing commits via "blaming" the statements modified by the bug-fixing commits. Costa *et al.* [18] later proposed a framework to evaluate the results of SZZ. They found that for 46.0% of their studied bugs, the bug-inducing commits identified by SZZ are years apart from one another while it is unlikely that code changes committed years apart will induce the same bug [15]. Unfortunately, these two studies have neither explained why SZZ is imprecise nor investigated the effect of such imprecision. Our study bridges this gap.

7 CONCLUSION

In this study, we collected the bug-fixing commits and the associated bug-inducing commits for 333 bugs from seven large open-source projects. Based on this dataset, we conducted empirical studies to understand the correlations, in terms of code elements and modifications, between a bug's inducing and fixing commits. The empirical findings explain why the SZZ algorithm, the most widely-adopted approach to collecting bug-inducing commits, is imprecise. We also observed that most of the findings revealed by previous studies that leveraged SZZ are significantly affected by SZZ's imprecision. Furthermore, by conducting experiments on Defects4J [27], we observed that leveraging the information of bug-inducing commits can significantly boost the performance of existing automated fault localization and program repair techniques.

In this study, the design of using bug-inducing commits in fault localization and program repair shows its effectiveness, but is still preliminary. In the future, we plan to devise more sophisticated approaches based on our empirical findings to further improve the performance of fault localization and automated program repair.

ACKNOWLEDGMENTS

Rongxin Wu is the corresponding author. This work is supported by RGC/GRF 16202917, MSRA Research Collaborative Grant 2018, the Hong Kong PhD Fellowship Scheme, the National Natural Science Foundation of China (Grant No. 61802164), the Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. ZDSYS201703031748284), and the Program for University Key Laboratory of Guangdong Province (Grant No. 2017KSYS008).

REFERENCES

[1] 2018. ACCUMULO-2967. https://issues.apache.org/jira/browse/ACCUMULO-2967. Accessed: 2018-02-28.

[2] 2018. AMBARI-3505. https://issues.apache.org/jira/browse/AMBARI-3505. Accessed: 2018-02-28.

[3] 2018. BUG-262975. https://bugs.eclipse.org/bugs/show_bug.cgi?id=262975. Accessed: 2018-02-28.

[4] 2018. HADOOP-13118. https://issues.apache.org/jira/browse/HADOOP-13118. Accessed: 2018-02-28.

[5] 2018. LUCENE-5786. https://issues.apache.org/jira/browse/LUCENE-5786. Accessed: 2018-02-28.

[6] 2018. SOLR-2606. https://issues.apache.org/jira/browse/SOLR-2606. Accessed: 2018-02-28.

[7] 2018. SOLR-8026. https://issues.apache.org/jira/browse/SOLR-8026. Accessed: 2018-02-28.

[8] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[9] Muhammad Asaduzzaman, Michael C Bullock, Chanchal K Roy, and Kevin A Schneider. 2012. Bug introducing changes: A case study with Android. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 116–119.

[10] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. 2007. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 19–26.

[11] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 97–106.

[12] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 104–113.

[13] Mario Luca Bernardi, Gerardo Canfora, Giuseppe A Di Lucca, Massimiliano Di Penta, and Damiano Distante. 2012. Do developers introduce bugs when they do not communicate? the case of eclipse and mozilla. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 139–148.

[14] Marcel Böhme, Bruno C d S Oliveira, and Abhik Roychoudhury. 2013. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 334–344.

[15] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 105–115.

[16] Douglas G Bonett and Thomas A Wright. 2000. Sample size requirements for estimating Pearson, Kendall and Spearman correlations. *Psychometrika* 65, 1 (2000), 23–28.

[17] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. 2011. How long does a bug survive? an empirical study. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 191–200.

[18] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2017. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2017), 641–657.

[19] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process* 26, 1 (2014), 107–139.

[20] Jordan Ell. 2013. Identifying failure inducing developer pairs within developer networks. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1471–1473.

[21] Jon Eyolfson, Lin Tan, and Patrick Lam. 2011. Do time of day and developer experience affect commit bugginess?. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 153–162.

[22] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 313–324.

[23] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 172–181.

[24] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016), 303–336.

[25] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 121–130.

[26] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. (2018).

[27] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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, 437–440.

[28] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773.

[29] Ken Kelley and Kristopher J Preacher. 2012. On effect size. *Psychological methods* 17, 2 (2012), 137.

[30] Sunghun Kim and E James Whitehead Jr. 2006. How long did it take to fix bugs?. In *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 173–174.

[31] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[32] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. 2006. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 81–90.

[33] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 489–498.

[34] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791* (2018).

[35] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software quality journal* 21, 3 (2013), 421–443.

[36] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[37] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. (2019).

[38] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. 2016. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 396–409.

[39] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[40] Christopher D Manning and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. Vol. 999. MIT Press.

[41] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.

[42] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA, Demonstration Track*.

[43] Osamu Mizuno and Hideaki Hata. 2010. Prediction of fault-prone modules using a text filtering based metric. *International Journal of Software Engineering and Its Applications* 4, 1 (2010), 43–52.

[44] Bernard Ostle et al. 1963. Statistics in research. *Statistics in research*. 2nd Ed (1963).

[45] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 609–620.

[46] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 491–500.

[47] Eric F Rizzi, Sebastian Elbaum, and Matthew B Dwyer. 2016. On the techniques we create, the tools we build, and their misalignments: a study of KLEE. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 132–143.

[48] Ripon K Saha, Sarfraz Khurshid, and Dewayne E Perry. 2014. An empirical study of long lived bugs. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 144–153.

[49] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 648–659.

[50] Francisco Servant and James A Jones. 2012. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 43.

[51] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. 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, 201–211.

[52] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. 2013. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering* 39, 4 (2013), 552–569.

[53] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. Hatari: raising risk awareness. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 107–110.

[54] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *ACM sigsoft software engineering notes*, Vol. 30. ACM, 1–5.

[55] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 273–283.

[56] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 471–482.

[57] Michele Tufano, Gabriele Bavota, Denys Poshyvanyk, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2017. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process* 29, 1 (2017), e1797.

[58] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2019. How Different Is It Between Machine-Generated and Developer-Provided Patches? An Empirical Study on The Correct Patches Generated by Automated Program Repair Techniques. (2019).

[59] Shangwen Wang, Ming Wen, Xiaoguang Mao, and Deheng Yang. 2019. Attention please: Consider Mockito when evaluating newly proposed automated program repair techniques. In *Proceedings of the Evaluation and Assessment on Software Engineering*. ACM, 260–266.

[60] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 319–330.

[61] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. ICSE.

[62] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 262–273.

[63] W Eric Wong and Vidroha Debroy. 2009. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45* 9 (2009).

[64] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2017. ChangeLocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* (2017), 1–35.

[65] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 15–25.

[66] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 660–670.

[67] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS)*. IEEE, 17–26.

[68] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 157–168.

[69] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 26–36.

[70] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering*. IEEE Computer Society, 9.