How Useful is Code Change Information for Fault Localization in Continuous Integration?

An Ran Chen*
Concordia University
Montreal, Quebec, Canada
anr_chen@encs.concordia.ca

Tse-Hsun (Peter) Chen†
Concordia University
Montreal, Quebec, Canada
peterc@encs.concordia.ca

Junjie Chen†
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

ABSTRACT

Continuous integration (CI) is the process in which code changes are automatically integrated, built, and tested in a shared repository. In CI, developers frequently merge and test code under development, which helps isolate faults with finer-grained change information. To identify faulty code, prior research has widely studied and evaluated the performance of spectrum-based fault localization (SBFL) techniques. While the continuous nature of CI requires the code changes to be atomic and presents fine-grained information on what part of the system is being changed, traditional SBFL techniques do not benefit from it. To overcome the limitation, we propose to integrate the code and coverage change information in fault localization under CI settings. First, code changes show how faults are introduced into the system, and provide developers with better understanding on the root cause. Second, coverage changes show how the code coverage is impacted when faults are introduced. This change information can help limit the search space of code coverage, which offers more opportunities for improving fault localization techniques. Based on the above observations, we propose three new change-based fault localization techniques, and compare them with Ochiai, a commonly used SBFL technique. We evaluate these techniques on 192 real faults from seven software systems. Our results show that all three change-based techniques outperform Ochiai on the Defects4J dataset. In particular, the improvement varies from 7% to 23% and 17% to 24% for average MAP and MRR, respectively. Moreover, we find that our change-based fault localization techniques can be integrated with Ochiai, and boost its performance by up to 53% and 52% for average MAP and MRR, respectively.

ACM Reference Format:
An Ran Chen, Tse-Hsun (Peter) Chen, and Junjie Chen. 2022. How Useful is Code Change Information for Fault Localization in Continuous Integration?. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556931

*This work was done when An Ran Chen was visiting Tianjin University.
†Corresponding author

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556931

1 INTRODUCTION

Continuous integration (CI) is a software practice by which developers frequently merge and test code under development. Every commit in CI consists of a smaller set of code changes that are incrementally tested in the system. The continuous practices ensure the stability of the code base and allow developers to detect test failures as early as possible. Occasionally, new code changes may introduce faults that cause the previously passing regression tests to fail [13].

Prior studies [6, 8, 10, 18, 26, 63] have proposed spectrum-based fault localization (SBFL) techniques to locate faulty locations at either statement or method level that may be the cause of test failures. In most studies, researchers evaluate SBFL techniques in traditional software development settings, where they only consider a single snapshot of the system. In contract, in modern software development, and especially in continuous integration, developers make continuous and finer-grained changes to the system. Hence, when a new test failure occurs, such fine-grained information may provide additional information to locate the fault. The atomic nature of code changes also limits the unintended consequences in scope. In that sense, fault isolation can be realized with more accessible metrics that are less expensive in diagnosis cost.

In this paper, we conduct an empirical study on the effectiveness of using and integrating change information for fault localization in CI settings. In particular, we study two types of change information: code changes and coverage changes. These two types of change information capture the changes in both the static and dynamic aspects of a system. We characterize the change information from a real world fault, and motivate the design insights of leveraging such information in fault localization. The key idea is to consider code changes and coverage changes as important debugging knowledge to improve fault localization, as such change information is readily available for systems following CI practices.

For further evaluation, we conduct our experiments on seven open source Java systems, with a total of 193 real world faults. We first study the overlaps between the change information and the faulty methods in the code. Our results suggest that, while both change information covers a reduced search space compared to code coverage, the percentages of faulty methods in the search space are 7 and 14 times higher for code changes and coverage changes, respectively. Then, we propose three change-based fault localization techniques and compare them with *Ochiai*, a commonly used SBFL technique. Our evaluation results show that all three change-based techniques outperform *Ochiai*'s performance. For example, in terms of average MAP and MRR, the improvement varies from 7% to 23% and 17% to 24% over *Ochiai*. Finally, we study how the change

information may be integrated with traditional SBFL techniques to further improve fault localization. We study different combinations of proposed change-based techniques and *Ochiai*. We find that combining both change information with *Ochiai* achieves the best performance, achieving up to 53% and 52% improvement over *Ochiai* for average MAP and MRR respectively, and locating 41 more faults at Top-1.

Our extensive evaluation shows that code changes can further help with effort reduction, and are important to consider in fault localization. To the best of our knowledge, this is the first study that integrates the change information available in CI to improve SBFL techniques. Our findings show that both change information provide valuable information on fault localization. Future studies may consider leveraging change information to complement fault localization techniques.

In summary, our contributions are:

- We created and released a fault localization dataset that contains 193 real world faults. Our dataset provides detailed code and coverage change information between the fault-inducing commit and its prior commit. The dataset may be used for benchmarking future fault localization techniques in the CI context.
- We found that both code and coverage changes have a reduced search space compared to code coverage (i.e., cover fewer methods). However, the covered methods have a higher faulty rate, which implies that change information may be used to improve the ranking of existing SBFL techniques.
- We proposed three change-based fault localization techniques, which all outperform *Ochiai* by a considerable margin (up to 23%, 23%, and 24% improvements over Top-1, MAP, and MRR).
- We found that by combining traditional SBFL technique (i.e., *Ochiai*) with change-based techniques, we can achieve an even larger improvement.

In summary, our study sheds light on the importance of change information. Future studies should consider leveraging such change information when designing fault localization techniques in CI settings. To ease the replication of our study, we made the replication package publicly available online [5].

Paper Organization. In Section 2, we present the motivation for leveraging code change coverage changes and related work. In Section 3, we present our experimental setup which includes the studied systems, fault dataset and our data collection process. In Section 4, we present our results and findings for each research question. In Section 5, we further elaborate on the results, and discuss the overheads. In Section 6, we discuss the threats to validity. In Section 7, we conclude this work.

2 MOTIVATION AND RELATED WORK

In this section, we first present the motivation for leveraging code and coverage changes to help improve fault localization in CI. Then, we discuss related work.

2.1 Motivation

Continuous integration is a software development practice by which developers regularly deliver into a central repository. CI has been widely established in modern software systems [21, 24, 41, 48]. With continuous delivery, developers automatically build and test new code changes incrementally. This practice helps to identify faults early in the development cycle, making them less expensive to fix.

Prior studies [7, 9, 16, 17, 32, 58, 60] focused on studying the use of code coverage in locating faults under traditional software development settings. However, under CI settings, the changes are incremental. Those finer-grained change information may provide helpful hints on the faulty locations. Typically, there are two types of change information: **code changes**, which track the modified code statements in a commit; and **coverage changes**, which record the changes in code coverage before and after the commit.

As pointed out by prior studies [14, 54, 55], changes to the system may help reveal hints on the faults. In this paper, we examine the two aforementioned change information and study whether they can provide additional information compared to code coverage, which is widely used in traditional spectrum-based fault localization (SBFL) techniques. We examine code and coverage change information in the CI context where changes are continuous and finer-grained. We consider these two types of change information since they are less expensive to obtain and may be readily available for systems following CI practices. Studying such change information may open new directions to improve fault localization. Below, we discuss the two types of changes in detail.

Coverage Changes. Coverage changes are the effect of the code changes from the coverage aspect. Coverage changes include two types of changed statements: statically and dynamically changed statements. While both types lead to changes in coverage execution, they are different in how the statements are changed. For the statically changed statements, changes happen because the original statements are modified based on the code changes. For instance, when developers modify a statement that is part of the code coverage, the coverage execution naturally changes from the original statement to the recently updated statement. For the dynamically changed statements, changes happen because the dynamic execution (e.g., control flow) in the system is different. The coverage changes as a result of the system taking an alternative execution path. Together, statically and dynamically changed statements provide different in-depth information on the changes performed on the system.

There are two benefits to leveraging such change information. First, the coverage changes can help characterize the source code from a new perspective — the perspective of the statements statically and dynamically influenced by the code changes. This perspective may gain new insights into existing problems, such as tie issue in fault localization. Tie issue is a well-investigated problem in traditional spectrum-based fault localization techniques [6, 8, 10, 18, 26, 63], which is caused by the exceeding number of statements within the code coverage. The coverage changes may help prioritize faulty locations within the coverage based on the statically and dynamically changed statements. Another advantage of using the coverage changes is that it helps limit the search space (i.e., coverage change is a subset of code coverage), which offers more opportunities for improving the precision in fault localization.

Let us illustrate the coverage changes through an example adapted from fault Time 2 in the Defects4J benchmark. In Figure 1, we show

```
// Commit: 3ba9ba7
// Partial.java
189: Partial(DateTimeFieldType[] types, int[] values, Chronology
     chronology){
212:   DurationField lastUnitField = null;
217:   int compare = lastUnitField.compareTo(loopUnitField);
218:   if (compare < 0 || (compare != 0 && ..)) {
219:     throw new IllegalArgumentException(..);
...
426: public Partial with(DateTimeFieldType fieldType, int value) {
463: - Partial newPartial = new Partial(iChronology, newTypes, newValues);
463: + // use public constructor to ensure full validation
464: + Partial newPartial = new Partial(newTypes, newValues, iChronology);
465:   iChronology.validate(newPartial, newValues);
466:   return newPartial;
474: }
// UnsupportedDurationField.java
226: public int compareTo(DurationField field){
227:   return 0;
228: }
```

s	Prior	Faulty
Partial.java		
189		
212		•
217		•
218		•
219		•
426		
463	•	
464		•
465	•	•
466	•	•
474		
UnsupportedDurationField.java		
226		
227		•
228		

Figure 1: Coverage of the fault-triggering test from Time-2, where *Faulty* denotes the fault-inducing commit and *Prior* denotes the commit prior to the fault-inducing commit.

the source code, and the coverage of the test failing in the fault inducing commit (denoted as Faulty) and passing in the prior commit (denoted as Prior). First, when updating from the prior commit to the fault-inducing commit, we observe that the statements are modified at line 463 and 464. Those modifications change the code coverage by covering a different *Partial* constructor, as well as introducing new coverage based on the statements within that new constructor. We highlight (in red) those coverage changes in Figure 1. While the code changes reveal what is being done to the system, the aforementioned coverage changes show the effect of the code changes on the system. In this fault, the faulty statements locate at line 218 in the `Partial.java` file, and at line 227 in the `UnsupportedDurationField.java` file where the coverage changes. A prior study [11] suggests there exist correlations between the statements dynamically affected by code changes and the faulty locations. This study finds that by inspecting only the dynamically changed statements, developers may reduce the inspection cost and find faults faster. The above observations motivate us to study the usefulness of coverage changes in fault localization.

Code Changes. In CI, code changes are one of the mostly used information by developers. They show the modified methods/code statements that cause the fault-triggering test to fail. Such change information is important to developers in practice, as it can provide developers with better understanding on the root cause of the faults. For instance, in the example illustrated in Figure 1, the code changes at line 463 and 464 provide a new perspective on the static change of the system, orthogonal to the coverage changes. Prior studies [14, 54] analyze the change metric (e.g., the complexity of the introduced code hunks), and suggest that code changes can be closely correlated to the faulty locations. However, the usefulness of code changes remains unknown in the context of CI and fault localization, where the changes are continuous and finer-grained. Based on the above observations, we further investigate how the code changes might contribute to enhancing existing fault localization techniques in CI.

2.2 Related Work

Debugging on Fault-Inducing Commits. Prior studies [14, 54, 55] show that the most recent commit that introduces the fault (i.e., fault-inducing commit) is highly correlated to the faulty locations. Even though a number of studies [14, 50, 53, 61] tackled this challenge by leveraging the new code change on the fault-inducing commit, they only analyze the changes applied to the source code. Yet, the dynamic changes derived from the code coverage contain the test execution information, which may further help improve the performance of fault localization techniques.

Testing Practices in CI. Prior studies identify test failures as one of the main challenges in CI practices [13, 43]. Beller et al. [13] investigate the impact of test failure in CI practices, and they find that test failures are responsible for most of the broken builds in CI. Shahin et al. [43] perform a systematic review of the challenges in adopting CI practices. One of the challenges inherent in adopting CI is the low test quality, which is characterized by frequent test failures, low test coverage and long running tests. Other existing studies investigate how test failures affect CI [20, 31]. Labuschagne et al. [31] evaluate the costs and benefits of testing in CI. They suggest that the prevalence of test failures caused by faults represents a benefit because it provides a positive return on CI maintenance cost. Elbaum et al. [20] present techniques for improving regression testing in CI. They propose algorithms to make CI processes more cost-effective. Hilton et al. [23] study the impact of code changes on coverage evolution in CI. They highlight the reasons why coverage can increase or decrease when code changes. Kochhar et al. [30] survey practitioners on their expectations of fault localization, and one of the respondents proposes the integration of fault localization in CI. In this study, we leverage the code coverage to compute coverage changes, and apply them in fault localization techniques in CI.

Table 1: An overview of studied systems.

System	# Faults	KLOC	# Test cases
Fastjson	88	183	4,736
Lang	6	22	2,245
Math	22	85	3,602
Closure	41	147	7,927
JacksonCore	12	45	664
Time	5	28	4,130
Chart	18	96	2,205
Total	192	606	25,509

3 EXPERIMENTAL SETUP

In this section, we first present the studied systems and fault dataset. Then, we discuss the data collection process, and the challenges that we encountered in test execution.

3.1 Studied Systems and Fault Dataset

Although there are several open source fault datasets such as Defects4J [29], none of them includes the code evolution details (e.g., the test result and code coverage information prior to the fault). Therefore, we collect the dataset using five studied systems from the Defects4J benchmark (version 1.0) and two additional systems (i.e., Fastjson and Jackson-core). In total, we collected 192 faults and the corresponding test failures across seven studied systems.

We choose the Defects4J benchmark because it has been widely used in prior fault localization studies [34, 42, 44, 53, 63]. The benchmark contains a clean dataset that allows researchers to reproduce the faults easily. For each fault, it provides the faulty commit, the fix commit, and fault-triggering tests. However, Defects4J is not applicable for studies in CI context due to the following reasons. First, the *faulty commit* identified by Defects4J does not fit the CI setting. The faulty commit is defined as one of the commits where fault happens, but not the first commit. However, testing begins as soon as the commits are submitted into CI, and if some tests fail, developers will investigate the issue at the failing commit, which differs from the faulty commit identified by Defects4J. To simulate a CI setting, we need to conduct the study on the the fault-inducing commit (i.e., where test failure occurs). Following prior studies [14, 53], we identify the matching fault-inducing commits by using "git bisect" to run the fault-triggering tests on previous commits of the system..

Second, the fault-triggering test at the fault-inducing commit may have different points of failure and reason of failing compared to the fault-fixing commit provided by Defects4J. As these provided fault-fixing commits serve as the ground truth for evaluating the effectiveness of fault localization techniques, they indicate the location where developers should change to fix the faults. However, there might have been code changes between the fault-fixing commit provided by Defects4J and the fault-inducing commit. Therefore, we need to make sure that the test is failing due to the same reason on both commits. To address this challenge, we extract the fault-triggering test from the fault-fixing commit and execute it on the fault-inducing commits by following a prior study [55]. Specifically, we first execute the fault-triggering test on the commit prior to the fault-fixing commit (i.e., where the fault still occurs) to obtain the point of failure (e.g., assertion statement). We then execute the same test on the fault-inducing commit and exclude the fault if the point of failure is different. We use this approach on all the 357 faults from the Defects4J 1.0 benchmark. At the end of this process, we collect 83 faults from the Defects4J 1.0 benchmark.

To further increase the size of the fault dataset, we added two additional systems (i.e., Fastjson and Jackson-core, which follow the CI practices) and increase the fault data in three Defects4J systems (i.e., Chart, Lang, and Time) that have the least number of faults after our previous data validation step. Fastjson is a popular open source Java system used for JSON object conversion (with 24k stars on GitHub). Fastjson has been used in prior research [19, 45] to study code evolution in the CI context. Jackson Project is a well-known Java JSON library, and its fundamental component, Jackson-core, is frequently used in prior fault localization studies [27, 39].

To collect the fault dataset in these five systems, we automatically compile and execute the tests for the 1,000 latest commits at the time when we conduct our case study in May 2021. Our goal is to find the fault-inducing commit where a fault is first introduced. We sort the commits by their creation time and find the first occurrence for each test failure. Because a test failure may continuously occur in sequential commits until the fault is fixed, we isolate and identify the first occurrence of the test failure as the fault-inducing commit. At the end of this process, we collect 109 additional faults (with a total of 192 faults, as shown in Table 1) and their corresponding fault-inducing commit.

3.2 Data Collection Process

In the previous subsection, we discuss the dataset that we collected and used in our study. In this subsection, we provide a detailed explanation of our data collection process.

3.2.1 Identifying the Commit Prior to Fault-Inducing Commit. To obtain the code and coverage changes that resulted in test failure, we need to identify the commit prior to the fault-inducing commit. Namely, the commit where the fault has not yet been introduced or triggered by the tests. We identify the prior passing commit using the Git command "`git rev-parse commit^`" where `commit` refers to the fault-inducing commit. In the case where there are multiple parent commits, the caret annotation (^) helps to locate the first immediate parent.

3.2.2 Collecting Code Changes. As mentioned in Section 2.1, we want to analyze the code changes to study the benefits of leveraging such change information in fault localization. To collect code change information, we first use the "`git diff`" command to capture the change information between the fault-inducing commit and the prior commit. This change information includes the modified files, the modified code statements, and their corresponding line numbers. To perform a more comprehensive analysis, we also trace higher granularity information (i.e., method in which the modified line belongs to). We use a static analysis tool (i.e., JavaParser [25]) to derive the per-method abstract syntax tree (AST) for each modified file. Since the generated ASTs contain the starting and ending line number for each method, we check whether the line number of the

modified statement is within the range of the starting and ending line numbers of the ASTs to determine its corresponding method.

3.2.3 Collecting Code Coverage. Our goal is to compare the change information to the conventional code coverage when leveraged in fault localization. Therefore, we collect the code coverage on the fault-inducing commit, and also on the prior commit, to identify the changes in code coverage. To automate this process, we integrate GZoltar [4] into each studied system as a Maven plug-in. GZoltar is a Java framework for automatic debugging and coverage generation. On every test execution, GZoltar instruments the source files to obtain a coverage matrix. The coverage matrix provides information on which statements were executed and by which tests. Thus, we collect the coverage matrix to compute the code coverage for each test.

3.2.4 Identifying Coverage Changes. In addition to code changes, we also want to study whether changes in code coverage help improving fault localization techniques. As our goal is to identify the code that is likely to be affected by faults, we compute the changes based on the coverage of the fault-triggering test, between the fault-inducing commit and the prior commit.

We first represent each covered statement using the code statements (e.g., `Reducer r = new Reducer(..)`), rather than the conventional location information (e.g., Reducer.java, line 33). When comparing the code coverage between two different commits, the location information is not reflective on what is the exact code statements been covered, which might introduce bias. For instance, if the code statement at line 33 changes, then there is a coverage change at line 33, despite the location information remaining the same (i.e., line 33 is covered) on the fault-inducing commit and the prior commit. Therefore, we map the code statements to code coverage and denote the code coverage as $Cov = \{s_1, s_2, ..., s_n\}$, where s represents the code statement covered. Then, we compare the statements covered on the fault-inducing commit (i.e., Cov_{fail}) with the prior commit where the test passed (i.e., Cov_{pass}). We locate the changes by identifying the newly added and changed code statements on the Cov_{fail}. We do not consider the deleted statements, because when we localize faults on the fault-inducing commit, only the existent statements are helpful for analysis. For instance, given $Cov_{fail} = \{s_1, s_2, s_3\}$ and $Cov_{pass} = \{s_2, s_3, s_4\}$, the change is $Cov_{change} = \{s_1\}$. Once we locate the changes, we describe them with the location representation.

3.3 Resolving Challenges in Test Execution

Building the systems and running the tests require non-trivial effort [46, 47]. As neither the fault-inducing commit nor the prior commit is readily available on Defects4J, we first need to find the two commits, build and compile the systems, and execute the tests multiple times. In total, we spent hundreds of hours of manual effort compiling the code, executing the tests, and collecting code coverage. To encourage future studies in the area and ease the replication of our study, we made the replication package publicly available [5]. It should be noted that while the data collection has been challenging, gathering the code coverage information requires lower overheads in practice. In Section 5.2, we further discuss about the time costs associated with using the change information in fault localization.

Below, we share how we resolve the challenges that we encountered, which may help future studies create benchmarks in CI settings.

Automatically compiling evolving code. The project structure may change as the system evolves. As a result, we need to update the location of the build file in our automation scripts accordingly. For instance, in the earlier versions of the system Time, the build file and the source files were placed inside a nested directory rather than at the root. In order to compile the system on the earlier versions, we need to manually resolve the issue and update our automation script to include the new location of the build file.

Fixing test execution issues. Compiling fault-triggering tests on the fault-inducing commit is not always straightforward. For instance, the JUnit 3 framework is not able to evaluate test annotations with excepted exception (e.g., `@Test(expected = Exception)` that is featured in JUnit 4. Hence, running a fault-triggering test that is implemented with JUnit 4 syntax on the fault-inducing commit that still uses JUnit 3 will result in a test compilation error. To solve this error, we manually refactor the tests to ensure there is no compilation issue.

Handling JDK compilation. Some studied systems may depend on specific versions of the Java Development Kit (JDK). To address this challenge, we manually determine the required JDK version for each studied system and build an automated script to switch between versions when needed.

Handling flaky tests. To ensure the reliability of our results, we need to remove flaky tests from the fault-inducing commit and the prior passing commit. Flaky tests generate inconsistent code coverage because of their non-deterministic nature. We run Deflaker [3, 12], a state-of-the-art flaky tests detection tool, on both the fault-inducing commit and the prior commit to detect flaky tests, and exclude them from the suspiciousness score computation.

3.4 Evaluation Metrics

To measure the effectiveness of leveraging fine-grained change information for fault localization, we consider the following three evaluation metrics: top ranked N (Top-N), mean average precision (MAP), and mean reciprocal rank (MRR), as they have been widely used in fault localization [15, 49, 54, 56, 61, 65]. Below, we discuss each metric in detail.

Top-N: Given a number N, the Top-N metric defines the number of faults whose faulty program elements (i.e., methods in our experiment) are ranked in the top n ranking positions. Top-N evaluates the ability to find relevant methods among the top ranked n methods. When the suspiciousness score is the same, we randomly break the tie, and repeat the process three times to calculate the average result.

MAP: The MAP metric first computes the average precision for each fault, then calculates the mean of the average precision. We define the average precision (AP) as the average of precision values at all ranks where relevant methods are found. MAP assesses the ability in finding all relevant methods.

Table 2: The number of total methods covered by code coverage, code changes and coverage changes, and the number of faulty methods captured in each information. *Total* and *Faulty* denote the number of total and faulty methods. *Faulty ratio* denotes faulty method ratio, which is the percentage of faulty methods per the total methods covered.

System	Code Coverage			Code Changes			Coverage Changes		
	Total	Faulty	Faulty Ratio	Total	Faulty	Faulty Ratio	Total	Faulty	Faulty Ratio
Fastjson	7,538	135	1.8%	283	88	31.1%	443	82	18.5%
Lang	41	6	14.6%	7	6	85.7%	8	5	62.5%
Math	827	18	2.2%	264	15	5.7%	90	14	15.6%
Closure	22,572	27	0.1%	293	12	4.1%	2,169	12	0.6%
JacksonCore	1,022	40	3.9%	185	31	16.8%	95	31	32.6%
Time	1,872	7	0.4%	41	5	12.2%	90	7	7.8%
Chart	1,611	21	1.3%	541	16	3.0%	68	19	27.9%
Total	35,483	254	0.7%	1,614	173	10.7%	2,963	170	5.7%

$$AP = \frac{\sum_{i=1}^{m} i/Pos(i)}{m} \quad (1)$$

MRR: The MRR metric calculates the mean of the reciprocal position at which the first relevant method is found. MRR assesses the ability to find the first relevant method.

$$MRR = \frac{1}{K}\sum_{i=1}^{K}\frac{1}{rank_i} \quad (2)$$

4 EXPERIMENT RESULTS

In this section, we present our experiment results by answering three research questions (RQs). For each RQ, we present the motivation, approach, and results and discussion.

RQ1: What Are the Overlaps Between the Change Information and Faulty Locations

Motivation: Prior research has widely studied code coverage-based fault localization techniques (e.g., SBFL) [7, 57, 62, 63, 66]. Despite their popularity, these techniques suffer from precision issues due to the broad search space [8, 28, 44, 62]. Intuitively, coverage changes are subsets of code coverage which implies a smaller search space. The code changes can also help restrict the search space while providing new information (e.g., the changed code may not have corresponding tests to cover it). Hence, in this RQ, we investigate how the change information overlaps with the faults, and whether they are helpful in fault localization.

Approach: To understand how the change information contributes to finding faulty locations, we analyze the number of ***faulty methods*** covered by each type of change information, and code coverage. We define faulty methods as the methods that were modified by developers to fix the faults (i.e., faulty locations). For each fault, we first identify a set of faulty methods from the fault-resolving commits. Then, we study how many faulty methods have code changes or coverage changes, and compare them to code coverage. Furthermore, we use ***faulty method ratio*** to study the percentage of faulty methods among all the covered methods. A higher faulty method ratio means that the identified search space has more faulty methods, which may be leveraged to improve the precision of fault localization techniques.

Results: Although coverage changes cover only 67% of the faulty methods from the code coverage, its faulty method ratio is 7 times higher. As shown in Table 2, coverage changes have overlaps with 170 faulty methods in the reduced search space (since coverage changes are a subset of code coverage) and code coverage has overlaps with 254 faulty methods. Although coverage changes cover fewer faulty methods, the covered methods have a much higher faulty method ratio (i.e., 7 times higher, 5.7% compared to 0.7% from code coverage) and significantly fewer methods compared to code coverage (i.e., 2,963 methods v.s. 35,483 methods). The results show that the coverage changes, as a subset of the code coverage, cover 12 times fewer total methods than the code coverage, which may help in the ranking of faulty locations. Nevertheless, the coverage changes provide as much as 67% of the faulty methods within the reduced search space. This means that, when leveraging the coverage changes, we can perform the fault localization on a much smaller number of methods, while identifying a good percentage of faulty methods. The above findings suggest initial evidence for the potentials of leveraging coverage changes to improve the precision of fault localization.

Code changes cover additional faulty methods over code coverage, and provide faulty method ratio that is 14 times higher. In Table 2, code changes overlap with 173 faulty methods in the reduced search space while code coverage overlaps with 254 faulty methods. Even though code changes cover fewer faulty methods, its faulty method ratio is 14 times higher (i.e., 10.7%, compared to 0.7% from code coverage). Code changes' search space is also smaller, covering 22 times fewer methods compared to code coverage (i.e., 1,614 methods v.s. 35,483 methods). We also find that code changes have overlap with 14% *additional* faulty methods that code coverage is not able to cover. After some manual investigation, we find that the reason is these faulty methods do not have a corresponding test and code coverage (i.e., not tested in the system). Hence, these additional faulty methods that code changes have overlap with may further help coverage-based fault localization techniques identify more faults.

In short, we find that both types of change information have a higher percentage of faulty method ratio within the identified search space. Moreover, code changes overlap with faulty methods that code coverage fails to identify. These findings shed lights on the potentials of incorporating change information to improve coverage-based fault localization in CI settings.

> Both types of change information cover a higher percentage of faulty methods compared to code coverage in their reduced search space. Code changes also cover additional faulty methods that do not have code coverage.

RQ2: How Does Change Information Perform in Fault Localization?

Motivation: In RQ1, we found that change information achieves a higher faulty method ratio in the reduced search space. However, it is yet to explore whether both types of change information can be used for fault localization. Therefore, in this RQ, we propose three change-based techniques derived from change information and evaluate their effectiveness in fault localization techniques under CI settings.

Approach: Our goal is to systematically study the effectiveness of each type of change information in fault localization. We adopt three change-based techniques to characterize the change information with fault proneness. These change-based techniques are based on the size of code changes, the size of coverage changes, and the size of the statements affected by coverage changes. *CodeChange*, *CoverageChange*, and *CoverageExecution* denote these three change-based techniques respectively, and they each exclusively leverages one of the aforementioned change-based metrics. We want to investigate how each change-based technique performs in fault localization.

We conduct our analysis at the **method level**. For each method, we compute its suspiciousness score by computing and aggregating the suspiciousness scores across all the statements within the method. Prior studies [33, 35, 44, 63, 64] have also demonstrated that such method level aggregation helps better distinguish the non-faulty statements from the faulty ones. We compare the change-based techniques with *Ochiai*, a commonly used SBFL technique [22, 32, 64, 66]. We choose *Ochiai* since it outperforms other SBFL formulas in terms of fault localization performance [32, 36, 63]. For evaluating the results, we examine the Top-1, Top-5 and Top-10 accuracy, MAP, and MRR values (defined in Section 3.4).

We design *CodeChange* to rank methods with the most changes to be more suspicious. Namely, a method is ranked to be more suspicious if it contains more modified statements. For instance, we rank the method with the most changed statements at position 1, indicating it is the most suspicious method. For methods without any code changes, the technique considers them as non-suspicious, and removes them from the ranking to reduce noise. We design *CodeChange* this way since we want to study how vanilla code changes may be used for fault localization, and previous research [38, 40, 59] observe that the size of a change is a good indicator of fault proneness.

We design *CoverageChange* to rank methods with most coverage changes as more suspicious. Within a method, each code statement with coverage change receives a suspiciousness score of 1. Within a method, we count the number of code statements with coverage change, and rank the method with the most changed statements at position 1. Hence, a method is ranked more suspicious if more changes happen in code coverage. A previous research [11] found that the size of changes gives good indication on the fault proneness. Therefore, we apply the same concept to study the effect of cover changes on fault localization. For the methods that do not have any coverage change, the technique considers them as non-suspicious, and removes them from the ranking to reduce noise.

We design *CoverageExecution* to rank methods with the most statements affected by the coverage changes to be more suspicious. Previous studies [37, 52, 58] found that the size of the execution affected by the faults can provide additional guidance towards the faulty locations. Intuitively, if there is a coverage change (either dynamic or static change) at any statement within a method, the internal state (i.e., dynamic execution) of the subsequent statements is likely affected. Therefore, we design this technique to boost the methods that are "likely affected" by the change. We identify the methods with the most affected statements to be more suspicious. For instance, if the first occurrence of the coverage changes locates at line 33 of a given method, then starting from line 33, we count the number of statements that were executed by the code coverage. If the number of statements executed (affected) in that method is higher than other methods, then it is considered as the most suspicious method. The methods without any coverage change are considered as non-suspicious, and removed from the ranking to reduce noise.

Results: On average, *CodeChange*, *CoverageChange* and *CoverageExecution* achieve 13%, 7% and 23% improvement over *Ochiai* for MAP, respectively, and 17%, 17% and 24% improvement for MRR, respectively. Table 3 shows the fault localization results in terms of MAP, MRR, Top-1, Top-5, and Top-10. We observe that all three techniques derived from change information, on average, perform better than *Ochiai* in fault localization. In particular, *CoverageExecution* has the best overall Top-5 and Top-10 (i.e., locating 109 and 118 faults), and the highest average MAP and MRR (i.e., with an average MAP of 0.37 and MRR of 0.52). *CoverageExecution* achieves an improvement of 23% for average MAP and 24% for MRR.

CoverageChange achieves the second best overall performance, improving the average MAP and MRR by 13% and 17% respectively (i.e., with an average MAP of 0.37 and MRR of 0.52.). *CodeChange* achieves an improvement of 7% and 17% for average MAP and MRR (i.e., with an average MAP of and MRR of 0.49). The results show that even simple techniques that rank by the size of code changes or coverage changes tend to perform well.

All three techniques achieve improvements in the overall Top-N values. *CodeChange* locates the most faults at Top-1 (i.e., locating 81 faults at Top-1), followed by *CoverageExecution* (i.e., locating 80 faults at Top-1), and *CoverageChange* also achieves improvements over *Ochiai* (i.e., locating 77 faults at Top-1). In terms of the Top-5 and Top-10, *CoverageExecution* achieves the most faults (i.e., locating 109 at Top-5, and 118 at Top-10), followed by *CoverageChange*

Table 3: Effectiveness of Ochiai, CodeChange, CoverageChange and CoverageExecution in terms of Top-1, Top-5, Top-10, MAP and MRR. For each project, we show the best MAP and MRR in bold. The last rows of the table show the sum values for Top-N, and the weighted average for MAP and MRR across the studied systems. The last row of the table shows the sum values for Top-N, and the weighted average for MAP and MRR.

System	Approach	Top-N			MAP	MRR
		N=1	N=5	N=10		
Fastjson	Ochiai	31	45	45	0.20	0.36
	CodeChange	45	53	55	0.29 (+45%)	0.56 (+56%)
	CoverageChange	54	62	65	**0.38 (+90%)**	**0.65 (+81%)**
	CoverageExecution	46	60	65	0.35 (+75%)	0.61 (+69%)
Lang	Ochiai	4	4	4	0.69	0.71
	CodeChange	4	5	5	**0.78 (+13%)**	**0.90 (+27%)**
	CoverageChange	4	5	5	0.72 (+4%)	**0.90 (+27%)**
	CoverageExecution	4	5	5	0.72 (+4%)	**0.90 (+27%)**
Math	Ochiai	9	10	13	0.51	0.51
	CodeChange	10	12	12	**0.53 (+4%)**	**0.54 (+6%)**
	CoverageChange	8	13	13	0.47 (-8%)	0.49 (-4%)
	CoverageExecution	8	14	14	0.50 (-2%)	0.52 (+2%)
Closure	Ochiai	4	10	13	0.18	0.16
	CodeChange	7	10	11	**0.22 (+22%)**	**0.24 (+20%)**
	CoverageChange	2	3	5	0.07 (-61%)	0.09 (-55%)
	CoverageExecution	2	8	9	0.12 (-33%)	0.13 (-35%)
JacksonCore	Ochiai	0	1	1	0.02	0.02
	CodeChange	7	7	7	**0.21 (+950%)**	**0.58 (+2800%)**
	CoverageChange	1	5	6	0.20 (+900%)	0.26 (+1200%)
	CoverageExecution	4	5	6	0.19 (+850%)	0.39 (+1850%)
Time	Ochiai	3	4	4	**0.63**	**0.50**
	CodeChange	2	2	4	0.29 (-54%)	0.46 (-8%)
	CoverageChange	1	2	2	0.24 (-62%)	0.32 (-36%)
	CoverageExecution	2	2	3	0.43 (-32%)	0.43 (-14%)
Chart	Ochiai	14	16	16	**0.80**	**0.83**
	CodeChange	6	7	8	0.50 (-38%)	0.54 (-35%)
	CoverageChange	7	15	16	0.57 (-29%)	0.62 (-25%)
	CoverageExecution	14	15	16	0.76 (-5%)	0.81 (-2%)
Sum/Avg.	Ochiai	65	90	96	0.30	0.42
	CodeChange	81	96	102	0.32 (+7%)	0.49 (+17%)
	CoverageChange	77	105	112	0.34 (+13%)	0.49 (+17%)
	CoverageExecution	80	109	118	**0.37 (+23%)**	**0.52 (+24%)**

(i.e., locating 105 at Top-5, and 112 at Top-10), and *CodeChange* (i.e., locating 96 at Top-5, and 102 at Top-10).

In short, the change-based techniques have a better fault localization performance compared to Ochiai. Even though change information has a reduced search space, our findings show that change information may be better at ranking the faulty methods and reducing possible investigation effort from developers. Future fault localization studies should consider change information due to its effectiveness and availability in CI settings.

> The three change-based techniques achieve an improvement that varies from 7% to 23% and 17% to 24% over *Ochiai* for the average MAP and MRR, respectively. The results also indicate that all three change-based techniques outperform *Ochiai* in locating faults across all studied Top-N metrics.

RQ3: Can Change Information Complement Existing Fault Localization Techniques?

Motivation: In RQ2, our findings show that the change-based techniques achieve better fault localization results compared to the coverage-based baseline (*Ochiai*). However, as found in RQ1, code coverage still covers more faulty methods compared to coverage and code changes. Therefore, we hypothesize that the two types of information (i.e., coverage and change information) may complement coverage-based SBFL techniques when combined together. In this RQ, we experiment with different combinations of *Ochiai* and the three proposed change-based techniques, and then we discuss their fault localization results.

Approach: To answer this RQ, we study the effectiveness of adding five different combinations of the change-based techniques to *Ochiai*. These combinations includes *Ochiai + CC*, *Ochiai + CovC*, *Ochiai + CovE*, *Ochiai + CovC + CC*, and *Ochiai + CovE + CC*, where we denote the size of code changes and coverage changes as *CC* and *CovC* respectively, and *CovE* as the size of the execution affected by coverage changes. Similar to RQ1 and RQ2, we conduct the fault localization at the method level by aggregating the suspiciousness scores of the code statements within a method (by taking the highest score). We compare the results of Top-1, Top-5, Top-10, MAP and MRR. Below, we describe how we combine *Ochiai* with *CC*, *CovC* and *CovE*.

Ochiai + CC: We combine the size of code changes with *Ochiai* by following a similar equation (Equation 3 below) defined in prior studies [15, 51, 56] to calculate a boost score for each code statement.

$$BoostScore(s) = \begin{cases} \frac{1}{rank} & \text{if } s \in \text{ RankedStatements} \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

The intuition is that the methods with more changes are ranked higher, and thus the corresponding statements receive a higher boost score. If a method is ranked second, then the boost score is 0.5 (1/2). If a method is not part of the coverage changes (and therefore not ranked), then the boost score is 0. We calculate the suspiciousness score for each code statement by adding the boost score to the initial suspiciousness score computed by *Ochiai*. Finally, we aggregate the suspiciousness score for all code statements within a method, and calculate method-level ranking.

Ochiai + CovC: We combine the size of coverage changes with *Ochiai* by also following Equation 3. The methods with more coverage changes are ranked higher, and thus more likely to be faulty. We attribute a higher boost score to the code statements within that method. Similarly, we calculate the suspiciousness score for each code statement by adding the boost score to the initial suspiciousness score computed by *Ochiai*.

Ochiai + CovE: We combine the size of the execution affected by coverage changes with *Ochiai* by following Equation 3. The methods with more affected execution are ranked higher, and thus more likely to be faulty. Similarly, we add the boost score to the initial suspiciousness score computed by *Ochiai* to come up with a final suspiciousness score.

Ochiai + CovC + CC and ***Ochiai + CovE + CC:*** We combine the change-based techniques from different change information

Table 4: Effectiveness of *Ochiai* when applied different combination of change information. For each combination, we evaluate the performance in terms of MAP and MRR. *CC* denotes the code change information. *CovC* denotes the coverage change information. *CovE* denotes the coverage execution information. The best performing approach is marked in bold. The last row of the table shows the sum values for Top-N, and the weighted average for MAP and MRR.

System	Approach	Top-N			MAP	MRR
		N=1	N=5	N=10		
Fastjson	Ochiai	31	45	45	0.20	0.36
	Ochiai + CC	46	61	61	0.28 (+40%)	0.58 (+61%)
	Ochiai + CovC	48	67	74	0.35 (+75%)	0.63 (+75%)
	Ochiai + CovE	46	68	72	0.35 (+75%)	0.62 (+72%)
	Ochiai + CovC + CC	54	74	76	**0.39 (+95%)**	**0.70 (+94%)**
	Ochiai + CovE + CC	55	75	75	0.38 (+90%)	0.69 (+92%)
Lang	Ochiai	4	4	4	0.69	0.71
	Ochiai + CC	4	5	5	**0.85 (+23%)**	**0.90 (+27%)**
	Ochiai + CovC	4	5	5	0.83 (+20%)	**0.90 (+27%)**
	Ochiai + CovE	4	5	5	0.83 (+20%)	**0.90 (+27%)**
	Ochiai + CovC + CC	4	5	5	0.83 (+20%)	**0.90 (+27%)**
	Ochiai + CovE + CC	4	5	5	0.83 (+20%)	**0.90 (+27%)**
Math	Ochiai	9	10	13	0.51	0.51
	Ochiai + CC	14	15	16	**0.71 (+39%)**	**0.73 (+43%)**
	Ochiai + CovC	9	13	15	0.53 (+4%)	0.56 (+10%)
	Ochiai + CovE	9	13	15	0.53 (+4%)	0.55 (+8%)
	Ochiai + CovC + CC	12	15	16	0.65 (+27%)	0.67 (+31%)
	Ochiai + CovE + CC	13	15	16	0.68 (+33%)	0.71 (+39%)
Closure	Ochiai	4	10	13	0.18	0.16
	Ochiai + CC	8	12	16	0.27 (+50%)	0.29 (+81%)
	Ochiai + CovC	4	10	13	0.17 (-6%)	0.20 (+25%)
	Ochiai + CovE	5	11	16	0.22 (+22%)	0.24 (+50%)
	Ochiai + CovC + CC	6	11	16	0.24 (+33%)	0.27 (+69%)
	Ochiai + CovE + CC	9	13	16	**0.30 (+67%)**	**0.33 (+106%)**
JacksonCore	Ochiai	0	1	1	0.02	0.02
	Ochiai + CC	1	7	7	0.14 (+600%)	0.33 (+1550%)
	Ochiai + CovC	1	6	6	0.08 (+300%)	0.22 (+1000%)
	Ochiai + CovE	4	6	6	0.10 (+400%)	0.38 (+1800%)
	Ochiai + CovC + CC	6	7	7	0.19 (+850%)	**0.53 (+2550%)**
	Ochiai + CovE + CC	6	7	7	**0.22 (+1000%)**	0.53 (+2550%)
Time	Ochiai	3	4	4	**0.63**	0.50
	Ochiai + CC	2	4	4	0.51 (-19%)	**0.51 (+2%)**
	Ochiai + CovC	2	4	4	0.42 (-33%)	**0.51 (+2%)**
	Ochiai + CovE	2	3	4	0.48 (-31%)	0.48 (-4%)
	Ochiai + CovC + CC	2	4	4	0.44 (-30%)	0.49 (-2%)
	Ochiai + CovE + CC	2	3	4	0.48 (-24%)	0.48 (-4%)
Chart	Ochiai	14	16	16	0.80	0.83
	Ochiai + CC	17	18	18	0.90 (+13%)	**0.96 (+16%)**
	Ochiai + CovC	8	18	18	0.66 (-18%)	0.71 (-14%)
	Ochiai + CovE	16	18	18	0.88 (+10%)	0.93 (+8%)
	Ochiai + CovC + CC	9	18	18	0.69 (-14%)	0.74 (-11%)
	Ochiai + CovE + CC	17	18	18	**0.92 (+14%)**	**0.96 (+16%)**
Sum/Avg.	Ochiai	65	90	96	0.30	0.42
	Ochiai + CC	92	122	127	0.40 (+33%)	0.57 (+38%)
	Ochiai + CovC	76	123	135	0.36 (+20%)	0.53 (+26%)
	Ochiai + CovE	86	124	136	0.40 (+33%)	0.56 (+36%)
	Ochiai + CovC + CC	93	134	142	0.42 (+40%)	0.61 (+45%)
	Ochiai + CovE + CC	106	136	141	**0.46 (+53%)**	**0.64 (+52%)**

together to examine the effect of each change metric on the performance *Ochiai*. To combine both change information in *Ochiai*, for each code statement, we add the boost scores calculated from each of technique to the suspiciousness score computed by *Ochiai*. We base our recommendation of results on the resulting suspiciousness score.

Results: Overall, *Ochiai + CovE + CC* achieves the best performance, improving the MAP and MRR values from *Ochiai* by 53% and 52% respectively. Table 4 compares the performance of *Ochiai* with different change information considered. We calculate the evaluation metrics for each combination and compute its improvement over Ochiai. On average, all techniques outperform *Ochiai*. Specifically, adopting both the size of affected statements and the size of code changes in Ochiai (i.e., *Ochiai + CovE + CC*) achieves the best overall MAP and MRR, and the highest Top-N values. *Ochiai + CovE + CC* achieves the optimum improvement of 53% and 52% for MAP and MRR over Ochiai (i.e., with an average MAP of 0.46 and MRR of 0.64). *Ochiai + CovC + CC* achieves the second best improvement of 40% and 45% for MAP and MRR (i.e., with an average MAP of 0.42 and MRR of 0.61).

Adopting the individual change metric also yields better results for *Ochiai*. Specifically, for *Ochiai + CC*, *Ochiai + CovC* and *Ochiai + CovE*, the improvements for MAP are 32%, 20% and 33%, respectively, and the improvements for MRR are 38%, 26% and 36%, respectively. We observe similar results in terms of the Top-N values. Our finding shows that any type of the studied change information provides noticeable benefits when combined with *Ochiai*.

We find that adopting both change information in *Ochiai* achieves the best performance on average. Either of the two combinations (i.e., *Ochiai + CovC + CC* and *Ochiai + CovE + CC*) provides better results than adopting individual change information. In particular, *Ochiai + CovC + CC* improves the overall MAP and MRR by 20% and 19% when compared to *Ochiai + CovC*, and 8% and 7% when compared to *Ochiai + CC*. We observe that combining *CovE* with *CC* produces better results compared to combining *CovC* with *CC*. The results suggest that *CovE* with *CC* complement each other better and can help further improve the fault localization results.

Adopting the size of code changes alone (i.e., *Ochiai + CC*) can significantly improve *Ochiai*. We observe an improvement of 33% and 38% for average MAP and MRR respectively. Moreover, *Ochiai + CC* achieves 92 faults at Top-1, locating 27 more faults than *Ochiai*. This result suggests that, by only investigating the first position, the developers might locate 48% (i.e., 92 out of 192 faults) of the faults. The above findings illustrate the effort reduction for developers in practice, and show the usefulness of code changes metric in fault localization. Compared to the use of two other change-based techniques (i.e., *Ochiai + CovC* and *Ochiai + CovE*), *Ochiai + CC* achieves better performance in fault localization. Particularly, in three out of the seven studied systems (i.e., Math, Closure and Chart), *Ochiai + CC* locates more faults at Top-1 than the two other metrics. This is because, different from the two other change-based techniques that are both based on the code coverage, the code changes leverage a different search space. The additional information as discussed in RQ1 helps to cover more faults.

We observe that the system, Time, experiences a decrease in fault localization performance when leveraging some of the change-based techniques. For instance, all of the combinations locate one less fault at Top-1, and up to one less fault at Top-5. After our investigation, we find that one specific fault, Time 16, contributes to this result. As the code changes and coverage changes are large in size (due to potential refactoring that happens when fixing the fault), it introduces much noise (i.e., non-faulty statements), making it more challenging to locate the faulty locations. Nevertheless, in

Ochiai + CovE, *Ochiai + CovC + CC* and *Ochiai + CovE + CC*, the faulty statement is only ranked at a slightly lower position (i.e., from position 1 to 4). Note that considering there are only five faults in Time, this difference is further magnified when looking at the percentages of difference in MAP and MRR values, but having a small impact on the overall results. The results still demonstrate the usefulness of the change information in locating faults, considering the overall performance improvement. Future studies are needed to explore different techniques to leveraging change-based techniques in fault localization.

> The change information can complement *Ochiai* by providing up to 53% and 52% improvement in MAP and MRR respectively, and locating 41 more faults at Top-1. Future fault localization techniques may consider combining both change information to further improve the performance.

5 DISCUSSION

5.1 Effectiveness of Change Metrics

Although the combination of the change metrics and *Ochiai* achieves promising fault localization results, the change metrics do not contribute to locating some of the faults. In this section, we discuss the reasons and hope the finding can provide insights for future studies in CI and fault localization.

Conventional statement coverage may not be sufficient for capturing the behaviour changes in some code statements (20/32). While common code coverage tools (e.g., JaCoCo, Cobertura, and GZoltar) report coverage at various levels such as code statement or branch (aggregated per method), they do not report the condition coverage within each code statement. Therefore, coverage change that happens at the condition-level may be missed. For example, in fault `Closure-85`, based on the coverage change analysis, the covered code statements are identical between the fault inducing commit and the prior passing commit. However, based on our manual study, the condition coverage changes at line 199. The statement at line 199 was:

`if (n.isEmpty() || (n.isBlock() && !n.hasChildren()))`. Although both the fault inducing commit and prior passing commit cover this code statement, the condition coverage is different. In fact, the prior commit only covers the first condition `n.isEmpty()` while the fault inducing commit covers both conditions. This change in condition coverage is relevant to the root cause of the fault, but has not been captured because of the absence of condition coverage information. Note that the above-mentioned condition coverage refers to *per-statement condition coverage*, which is different from the term "condition coverage" used in Cobertura [1] (also called "branch coverage" in JaCoCo [2]) which shows the percentage of conditions covered throughout a method.

Our findings show that despite the advantages of leveraging coverage change information, there is a need for finer-granularity coverage information. Future studies are encouraged to study the usefulness of finer-granularity coverage information in fault localization.

Noise introduced when combining the change metrics (12/32). To better examine the effect of each change metric on the performance, we adopt the design of our approach to combine the change metrics following prior studies [15, 56]. We find that, in some faults, such combinations may introduce noises (i.e., prioritizing non-faulty statements). For instance, on fault Closure 120, we observe that neither the code changes, nor the coverage changes contain the faulty statements, thus the approach boosts the non-faulty statements to a higher position. This causes the rank of the faulty statements to be further pushed down in the ranking, which reduces the performance. While faults are affected by noises, the result demonstrates that the effect on the overall performance is trivial.

5.2 Overheads of Change-based Techniques

In this subsection, we discuss the overheads for integrating change-based fault localization techniques into CI.

To evaluate the overheads, we measure the processing time in seconds for locating a fault. The localization breaks down into four steps. In particular, step 1 refers to collecting and analyzing code change. Step 2 refers to collecting code coverage. Step 3 refers to performing code coverage change analysis. Finally, step 4 refers to ranking suspicious methods. On average, it takes less than 42 seconds in total to determine the final ranking of suspicious methods. In practice, this processing time only represents a small overhead considering a single build can take more than 12 minutes to run in some projects (i.e., Fastjson). The main source of overheads originates from the second step to collect code coverage. This step requires compilations from the fault-inducing commit and the prior passing commit. Therefore, depending on the size of the system, it takes 13 to 43 seconds to successfully compile both commits. Collecting the code coverage of failing tests can take up to 8 to 24 seconds. While this step contributes to significant time cost, the compilation of the system is a necessary procedure in CI to run the builds. And thus, in practice, the compiled source code can be directly used to collect the code coverage when test failures are identified. This can help to reduce the overheads associated with the compilation time. Eventually, as part of the continuous practice, change-based techniques can be automatically triggered when test failures happen during nightly builds. As some tests might take longer time to run, this procedure allows the collection of additional information on the test failures overnight (e.g., list of suspicious methods, change information), which can assist developers in debugging later.

6 THREATS TO VALIDITY

External Validity. One potential threat to external validity is the generalizability of our results based on the studied systems. To mitigate this threat, we conduct our experiments on seven real-world open source Java systems each with different characteristics and infrastructures. On top of the five studied systems provided by the Defects4J benchmark, we carefully select two additional systems that are actively maintained, widely used, and follow the CI practices. While we cannot confirm the generalization of our findings to fault localization approaches written in other programming languages, we design our approaches of leveraging the change metrics

as generic as possible. Future study can easily adopt our approaches to fit other programming languages.

Construct Validity. One potential threat is our design decision of combining the change information to *Ochiai* in RQ3. There may be better ways of combining the change information that further explore their benefits. We adopt the approaches following existing studies [15, 51, 56] to provide insights on the effects of combining the change information together. Such design helps better illustrate the improvement over the baseline (i.e., *Ochiai*), and can be easily integrated into the CI context. We encourage future studies to explore other ways of combining the change information, and release the data online [5] to facilitate replication. In addition, we did not consider mutation analysis in our approach. While the mutation analysis may provide more information, one challenge is that mutation is very costly, and thus does not fit the CI scenario. Moreover, in this study, we focus on two change information as they provide accurate information on the internal execution changes of the system. However, there are other change information that may be leveraged in fault localization. Future study is needed to investigate other change metrics.

7 CONCLUSION

We present finer-grained change information, code and coverage changes, in this paper, as a new direction to improve fault localization. Compared to the traditional code coverage metric used in SBFL techniques, change information presents limited search space and additional information from the version history. Besides, both change information are less costly to obtain and may be readily available for systems following CI practices. However, such information is not utilized by prior SBFL techniques. In this paper, we conduct an empirical study on the helpfulness of change information to perform fault localization in CI. Our findings show that, compared to code coverage, change information helps to limit the search space and covers a larger percentage of faulty method ratio. Inspired by these findings, we propose and evaluate three change-based techniques on seven open source systems. Our results show that adopting the change information achieves an improvement over *Ochiai* that varies from 7% to 23% and 17% to 24% for average MAP and MRR respectively. Moreover, our results also show that these change-based techniques can complement *Ochiai* by providing up to 53% and 52% improvement in MAP and MRR respectively, and locating 41 more faults at Top-1. In short, our study sheds light on the usage of change information in fault localization. We encourage future studies to consider leveraging such change information when designing fault localization techniques.

REFERENCES

[1] 2021. Cobertura. https://cobertura.github.io/cobertura/. Last accessed May 5 2021.

[2] 2021. JaCoCo. https://www.eclemma.org/jacoco/. Last accessed May 5 2021.

[3] 2022. Deflaker. https://www.deflaker.org/. Last accessed February 28 2022.

[4] 2022. GZoltar. https://gzoltar.com/. Last accessed February 28 2022.

[5] 2022. Leveraging-Change-Information repository. https://github.com/anonymized-datascientist/Leveraging-Change-Information. Last accessed March 7 2022.

[6] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. 88–99.

[7] 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.

[8] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A Practical Evaluation of Spectrum-based Fault Localization. *Journal of Systems and Software* 82, 11 (Nov. 2009), 1780–1792.

[9] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.

[10] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*. 89–98.

[11] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. 2011. Fault-localization using dynamic slicing and change impact analysis. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 520–523.

[12] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 433–444.

[13] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 356–367.

[14] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 international symposium on software testing and analysis*. 105–115.

[15] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. 2021. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering* (2021).

[16] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234.

[17] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89.

[18] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. 2018. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 184–191.

[19] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. 2020. A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering* (2020).

[20] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235–245.

[21] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. 2013. Development and deployment at facebook. *IEEE Internet Computing* 17, 4 (2013), 8–17.

[22] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[23] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 53–63.

[24] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 426–437.

[25] JavaParser. 2019. https://javaparser.org/. Last accessed July 1 2020.

[26] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 502–514.

[27] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting concise bug-fixing patches from human-written patches in version control systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 686–698.

[28] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 467–477.

[29] 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*. 437–440.

[30] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[31] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous

integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 821–830.

[32] Tien-Duy B Le, Ferdian Thung, and David Lo. 2013. Theory and practice, do they match? a case with spectrum-based fault localization. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 380–383.

[33] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 169–180.

[34] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673.

[35] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.

[36] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. 2014. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process* 26, 2 (2014), 172–219.

[37] Wes Masri. 2010. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability* 20, 2 (2010), 121–147.

[38] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*. 181–190.

[39] Manish Motwani and Yuriy Brun. 2020. Automatically repairing programs using both tests and bug reports. *arXiv preprint arXiv:2011.08340* (2020).

[40] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering* (St. Louis, MO, USA) *(ICSE '05)*. ACM, New York, NY, USA, 284–292.

[41] Steve Neely and Steve Stolt. 2013. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *2013 Agile Conference*. IEEE, 121–128.

[42] 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 *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[43] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.

[44] 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*. 273–283.

[45] Xuezhi Song, Yun Lin, Siang Hwee Ng, Ping Yu, Xin Peng, and Jin Song Dong. 2021. Constructing Regression Dataset from Code Evolution History. *arXiv preprint arXiv:2109.12389* (2021).

[46] Matúš Sulír and Jaroslav Porubän. 2016. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. 17–25.

[47] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017), e1838.

[48] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 805–816.

[49] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*. 53–63.

[50] Shaowei Wang and David Lo. 2016. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process* 28, 10 (2016), 921–942.

[51] Shaowei Wang and David Lo. 2016. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process* 28, 10 (2016), 921–942.

[52] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. 2009. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 45–55.

[53] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* (2019).

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

[55] 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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 326–337.

[56] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*. 181–190.

[57] W Eric Wong, Vidroha Debroy, and Byoungju Choi. 2010. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software* 83, 2 (2010), 188–208.

[58] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[59] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* 23, 5 (2018), 2866–2900.

[60] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4 (2013), 1–40.

[61] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. 2015. Bug localization based on code change histories and bug reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 190–197.

[62] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology* 124 (2020), 106312.

[63] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–272.

[64] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. 2019. An empirical study of boosting spectrum-based fault localization via pagerank. *IEEE Transactions on Software Engineering* 47, 6 (2019), 1089–1113.

[65] Jian Zhou, Hongyu Zhang, and David Lo. 2012. 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, 14–24.

[66] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347.