Automatically Learning a Precise Measurement for Fault Diagnosis Capability of Test Cases

YIFAN ZHAO, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China

ZEYU SUN, National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences, China

GUOQING WANG, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China

QINGYUAN LIANG, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China

YAKUN ZHANG, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China

YILING LOU, Fudan University, China

DAN HAO[*], Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China

LU ZHANG, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China

Prevalent Fault Localization (FL) techniques rely on tests to localize buggy program elements. Tests could be treated as fuel to further boost FL by providing more debugging information. Therefore, it is highly valuable to measure the Fault Diagnosis Capability (FDC) of a test for diagnosing faults, so as to select or generate tests to better help FL (i.e., FL-oriented test selection or FL-oriented test generation). To this end, researchers have proposed many FDC metrics, which serve as the selection criterion in FL-oriented test selection or the fitness function in FL-oriented test generation. Existing FDC metrics can be classified into result-agnostic and result-aware metrics

[*]Dan Hao is the corresponding author.

Authors' addresses: Yifan Zhao, zhaoyifan@stu.pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Zeyu Sun, zeyu.zys@gmail.com, National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences, Beijing, China; Guoqing Wang, guoqingwang@stu.pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Qingyuan Liang, liangqy@stu.pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Yakun Zhang, zhangyakun@stu.pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Yiling Lou, yilinglou@fudan.edu.cn, Fudan University, Shanghai, China; Dan Hao, haodan@pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Lu Zhang, zhanglucs@pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

depending on whether they take test results (i.e., passing or failing) as input. Although result-aware metrics perform better in test selection, they have restricted applications due to the input of test results, e.g., they cannot be applied to guide test generation. Moreover, all the existing FDC metrics are designed based on some predefined heuristics and have achieved limited FL performance due to their inaccuracy. To address these issues, in this paper, we reconsider result-agnostic metrics (i.e., metrics that do not take test results as input), and propose a novel result-agnostic metric RLFDC which predicts FDC values of tests through reinforcement learning. In particular, we treat FL results as reward signals, and train an FDC prediction model with the direct FL feedback to automatically learn a more accurate measurement rather than design one based on predefined heuristics. Finally, we evaluate the proposed RLFDC on Defects4J by applying the studied metrics to test selection and generation. According to the experimental results, the proposed RLFDC outperforms all the result-agnostic metrics in both test selection and generation, e.g., when applied to selecting human-written tests, RLFDC achieves 28.2% and 21.6% higher acc@1 and mAP values compared to the state-of-the-art result-agnostic metric TfD. Besides, RLFDC even achieves competitive performance compared to the state-of-the-art result-aware metric FDG in test selection.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Fault localization, Fault diagnosability, Reinforcement learning

ACM Reference Format:

Yifan Zhao, Zeyu Sun, Guoqing Wang, Qingyuan Liang, Yakun Zhang, Yiling Lou, Dan Hao, and Lu Zhang. 2024. Automatically Learning a Precise Measurement for Fault Diagnosis Capability of Test Cases. 1, 1 (December 2024), 28 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software debugging is a critical but painstaking process that costs tremendous resources every year [45, 47]. To alleviate this problem, Fault Localization (FL) techniques aim to automatically localize the buggy elements [47] and thus reduce the cost. Prevalent FL techniques [2, 18, 23, 28, 30, 34–36, 38, 39, 46, 56] rely on the coverage information and execution results of tests as input to predict a suspicious value for each code element, indicating their possibility of harboring faults. Consequently, the effectiveness of FL techniques heavily relies on the quality of the tests, as testing information provides critical clues for fault diagnosis. In other words, the fault diagnosis capability (FDC) of a test suite might potentially become the bottleneck of the FL effectiveness [11, 40]. Unfortunately, in practice, tests with high FDC are not always available, especially considering the heavy expense for manually writing tests [12].

To mitigate this issue, researchers propose FDC metrics to measure the diagnostic capability of a test or a test suite, and apply these FDC metrics to test generation [9, 11, 40] or selection [5, 16] for improving FL. Fig. 1 presents the workflow of these applications, where FDC metrics play a central role. Given a fault-triggering test (which composes an initial test suite), both of these two tasks aim to augment the initial test suite by generating or selecting more additional tests to more accurately pinpoint the fault. In particular, test generation for FL aims to automatically generate more high-quality tests. It utilizes FDC metric as a fitness function to guide the evolutionary algorithm of test generation tools and generate more tests with high FDC. These tests are then used to construct an augmented test suite to boost the performance of FL techniques. Test selection for FL aims to select valuable tests with high FDC from a given test pool. The selected tests are then used to construct an augmented test suite to boost FL. Selection is performed here because the existing test generation tools cannot generate accurate test oracles and thus the oracles are usually manually labelled [11]. To alleviate human efforts on oracle labeling, FDC metrics are applied to select a subset of valuable tests from the whole set of tests generated by test generation tools. The selected tests are then labeled with oracles and provides additional debugging information for FL techniques. In summary, FDC metric serves as a fitness function in test generation and a selection criterion in test selection. Generally, test-based FL techniques, especially

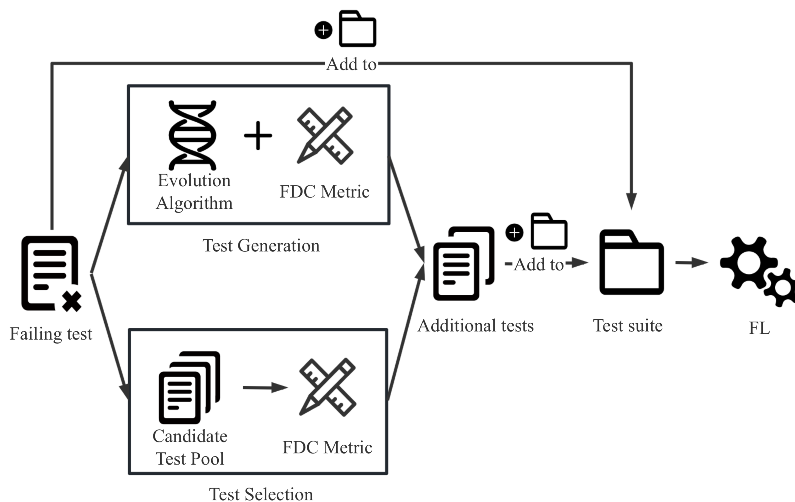

Fig. 1. The overall workflow of FDC metric

spectrum-based FL [2, 3], predict suspicious values of code elements based on the diagnosis capability of tests, and FDC metric is proposed to quantify this capability.

Existing FDC metrics can be classified into result-agnostic [9, 11, 15, 40] and result-aware metrics [5, 8, 16, 54]. Result-agnostic metrics calculate the FDC value based on test coverage, while result-aware metrics calculate the FDC value based on test coverage and previous test execution results. The extra input of execution results makes result-aware metrics perform better but limits their application scenarios. For example, they cannot be applied to test generation for FL, because during the evolution process in test generation, accurate oracles cannot be automatically generated [11, 40]. Moreover, all existing metrics are based on hand-crafted formulas and therefore use fixed heuristics on all subjects. However, the prior knowledge may be inaccurate and lacks general applicability for different subjects, making the proposed metrics inherently ineffective. For example, our experimental results on Defects4J show that the result-agnostic metric TfD [9] performs well on *Chart* and *Lang*, but performs badly on *Time* and *Math*.

To solve this problem, we propose a Reinforcement Learning (RL) based approach to build a more accurate result-agnostic FDC metric, RLFDC. Note that, differing slightly from previous work [5][1], we classify metrics as “result-aware” or “result-agnostic” based on whether they require execution results from tests other than the initial failing test. Our focus is on localizing the fault for a given failing test, thus the information from this initial test is known and can be leveraged. Importantly, utilizing this initial information does not impede the application of RLFDC to test generation; therefore, we categorize it as a result-agnostic metric. Different from previous metrics built with ingenious formulas, an RL model is built through repetitive training to automatically learn a measuring strategy, and the finalized RL model serves as our FDC metric RLFDC. Concretely, during the training stage, our approach uses the coverage information of tests to calculate state and action in RL, and uses the actual FL results to calculate the reward to motivate the RL model. Following Fig. 1, we investigate the performance of RLFDC in test selection and generation for FL, which can be regarded as the evaluating stage of the RL model. In test generation for FL, we use an FDC metric as the fitness function

[1]Prior work [5] classify metrics as “result-aware” or “result-agnostic” based on whether they require test execution results.

of an evolutionary algorithm and thus integrate it with a popular test generation tool EVOSUITE [13]. In test selection for FL, we use an FDC metric as the selection criterion and apply it to a test pool.

We evaluate RLFDC on the widely-used benchmark Defects4J [26], by comparing it against three representative result-agnostic metrics (i.e., EntBug [11], DDU [40], and TfD [9]) and the state-of-the-art result-aware metric FDG [5]. We evaluate the FDC metrics in three FL-oriented scenarios: test selection in human-written tests, test selection in automatically generated tests, and automated test generation. In each scenario, FDC metrics are used to guide test selection or generation, and their performance is evaluated via feeding the selected or generated tests to an FL technique. According to the results, in the test selection scenario, RLFDC outperforms all the existing result-agnostic metrics, while achieving competitive performance with the state-of-the-art result-aware metric FDG. With human-written tests, RLFDC achieves 28.2%, 36.2% and 21.6% higher acc@1, acc@10, and mAP (Mean Average Precision) values compared to the existing best result-agnostic metric TfD [5], respectively. With tests automatically generated by EVOSUITE, RLFDC achieves 3.0%, 5.6% and 4.4% higher acc@1, acc@10, and mAP values compared to TfD, respectively. With tests automatically generated by Randoop [37], RLFDC achieves 12.0%, 8.1% and 3.4% higher acc@1, acc@10, and mAP values compared to TfD, respectively. In the test generation scenario where result-aware metrics cannot be applied, EVOSUITE under the guidance of RLFDC successfully generates tests with high FDC values and helps improve FL performance. In particular, RLFDC achieves 4.9% and 3.1% higher acc@1 and mAP values compared to the state-of-the-art FL-oriented test generation technique DDU, respectively. We also evaluate RLFDC in the cross-project scenario and the results show that RLFDC steadily outperforms state-of-the-art result-agnostic metrics, indicating its general effectiveness. Finally, we perform an ablation study and the results show that each component of RLFDC positively contributes to its effectiveness.

The contribution of this paper can be summarised as follows:

- A new result-agnostic metric RLFDC which is constructed by our RL-based approach. To the best of our knowledge, we are the first to apply RL to this task.
- An extensive evaluation of RLFDC, including three scenarios: test selection (based on both human-written and automatically generated tests) and test generation.
- A prototype implementation of a RLFDC-guided test generation tool on top of EVOSUITE [13]. To our best knowledge, we are the first to integrate EVOSUITE with a machine learning model to guide test generation.

2 MOTIVATING EXAMPLE

We use the following motivating example to illustrate how an FDC metric facilitates test selection for FL. This example, derived from the *Math* project within Defects4J [26], represents a simplified real-world example. There is an initial failing test $t0$ which reveals a fault in the project. Our aim is to select additional tests with FDC metrics to help developers pinpoint the buggy method. Table 1 shows the coverage information and test outcomes for the ten tests selected by RLFDC and TfD, respectively (1/0 demonstrates cover/uncover and F/P demonstrates Fail/Pass). For simplicity, we only show the coverage matrix for the 26 statements, spanning six methods, that are covered by the failing test $t0$. The buggy statements and methods are highlighted in red. Table 2 and Table 3 show the change of "suspicious score/method ranking" of each suspicious method when additional tests are selected by RLFDC and TfD, respectively. The suspicious scores are computed with a widely-studied FL technique Ochiai [3] using the max-tie-breaker [5]. We aggregate the line-level Ochiai scores at the method level with the highest score of all the lines in the method [43].

From Table 2 and Table 3, we find that with ten tests selected by RLFDC, the buggy method $m4$ is ranked at the 1st, while with ten tests selected by TfD, it is ranked at the 3rd, which demonstrates the superiority of RLFDC. We further

Table 1. Coverage information for tests selected by RLFDC and TfD

		Init. Test	Tests selected by RLFDC										Tests selected by TfD									
Method	Statement	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15	t16	t17	t18	t19	t20
m1	s1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	1	0	0	1
m2	s2	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
m3	s3	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	s4	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	s5	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	s6	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	s7	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	s8	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
m4	s9	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	s10	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	s11	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	s12	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	s13	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	s14	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	s15	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	s16	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	s17	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
m5	s18	1	1	1	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1
m6	s19	1	1	1	1	0	1	1	0	1	1	0	0	1	1	1	0	0	1	0	0	1
	s20	1	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0	0	1
	s21	1	1	1	1	0	1	1	0	1	1	0	0	1	1	1	0	0	1	0	0	1
	s22	1	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0
	s23	1	1	1	1	0	1	1	0	1	1	0	0	1	1	1	0	0	1	0	0	1
	s24	1	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0
	s25	1	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0
	s26	1	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0
	s27	1	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0
	s28	1	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0
Test Outcome		F	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P

Table 2. Suspicious values for the suspicious methods with ten tests selected by RLFDC

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
m1	1.000/6	0.707/6	0.577/6	0.500/6	0.447/6	0.408/6	0.378/6	0.354/6	0.333/6	0.316/6	0.316/6
m2	1.000/6	0.707/6	0.707/3	0.707/3	0.577/4	0.577/3	0.577/3	0.577/2	0.577/2	0.577/2	0.577/2
m3	1.000/6	0.707/6	0.707/3	0.707/3	0.577/4	0.577/3	0.577/3	0.500/3	0.500/3	0.500/3	0.500/3
m4	1.000/6	0.707/6	0.707/3	0.707/3	0.707/1	0.707/1	0.707/1	0.707/1	0.707/1	0.707/1	0.707/1
m5	1.000/6	0.707/6	0.577/6	0.500/6	0.500/5	0.447/5	0.408/5	0.408/5	0.408/5	0.408/4	0.408/4
m6	1.000/6	0.707/6	0.577/6	0.577/4	0.577/4	0.500/4	0.447/4	0.447/4	0.408/5	0.378/5	0.378/5

Table 3. Suspicious values for the suspicious methods with ten tests selected by TfD

	t0	t11	t12	t13	t14	t15	t16	t17	t18	t19	t20
m1	1.000/6	0.707/6	0.577/6	0.500/6	0.447/6	0.447/6	0.447/6	0.408/6	0.408/6	0.408/6	0.378/6
m2	1.000/6	1.000/5	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3
m3	1.000/6	1.000/5	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3
m4	1.000/6	1.000/5	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3	1.000/3
m5	1.000/6	1.000/5	0.707/5	0.707/4	0.707/4	0.707/4	0.707/4	0.707/4	0.707/4	0.707/4	0.577/4
m6	1.000/6	1.000/5	0.707/5	0.577/5	0.577/5	0.577/5	0.577/5	0.500/5	0.500/5	0.500/5	0.500/5

explain how this result comes. Initially, all the suspicious statements are covered by the initial failing test $t0$. Therefore, all the suspicious methods have the same suspicious score and cannot be discriminated from one another. With the max-tie-breaker strategy, they are all ranked at the 6th. For RLFDC, it considers two features to value each test, *cover* and *split* (detailed in Section 3.1), and can adaptively balance the weight between the two features based on the current state thanks to reinforcement learning. For the *cover* feature, it measures how the additional test covers the suspicious statements since a passing test covering more statements can help reduce the suspiciousness of more non-buggy statements and help pinpoint the fault faster. For the *split* feature, it measures how an additional test distinguishes statements that were previously indistinguishable. Specifically, statements covered by the same set of tests are referred to as an "ambiguity group" in Section 3.1. Within an ambiguity group, all statements share the same program spectrum since they are executed by the same set of tests. Consequently, these statements are assigned the same suspicious value when spectrum-based fault localization techniques are applied, resulting in an indistinguishability problem. To address this, RLFDC incorporates the *split* feature, aiming to break the ambiguity groups formed by existing tests. In the example, RLFDC first selects tests that cover suspicious statements as many as possible. Therefore, it selects $t1$ and $t2$ which cover 28 and 18 statements respectively, and the buggy method $m4$ is ranked 3rd with the two tests. Note that $m4$ now has the same score with $m2$ and $m3$, because the buggy statement $s14$ now is in the same ambiguity group with $s2$, $s3 - s8$ and they cannot be discriminated from each other. Then RLFDC finds that with $t1$ and $t2$, there are many ambiguity groups formed (e.g., $s2 - s9$, $s10 - s13$). Based on the current state, RLFDC turns its attention to splitting those ambiguity groups to discriminate the statements and help more accurately pinpoint the fault. In particular, it selects $t3$ and $t4$ to break the groups. The test $t4$ covers $s2$, $s3 - s8$ but does not cover $s14$. Therefore, the group is broken and $s14$ has the highest suspicious score, making $m4$ ranked at the 1st. As for the state-of-the-art result-agnostic metric TfD, it only considers the number of ambiguity groups. Therefore, it aims at discriminating statements as much as possible by selecting $t11$ and $t12$. However, there are many ambiguity groups and it tries to split the trivial groups in those non-buggy methods $m1$, $m5$, and $m6$, failing to discriminate statements in $m2$, $m3$, and $m4$. Therefore, the buggy method $m4$ is finally ranked at the 3rd. Note that compared to TfD, RLFDC splits ambiguity groups in a more effective way since RLFDC also considers the *cover* feature. Therefore, RLFDC will select $t11$ with a smaller priority since $t11$ can only discriminate one statement $s1$.

Despite incorporating the two features, RLFDC utilizes reinforcement learning (RL) to automatically and adaptively determine their optimal balance based on the current state, including the number of tests and ambiguity groups. Specifically, RLFDC improves its accuracy by utilizing fault localization feedback as the RL reward, thereby achieving a more precise FDC measurement. This precision stems from the fact that the fault localization contribution directly reflects the FDC value of a test. We will provide a detailed explanation of our RL framework in the following section.

3 METHODOLOGY

We propose a RL-based approach to automatically construct an FDC metric, i.e., **R**eforcement **L**earning based **F**ault **D**iagnosis **C**apability (RLFDC). Unlike existing FDC metrics designed based on pre-defined heuristics, RLFDC is constructed systematically by learning the measuring strategy based on direct FL feedback. The key advantage of RL is that RL uses reward to update the model while traditional supervised learning approaches need labels to train the model. However, in such a scenario, it is hard to get labels for the FDC values of tests because FDC is influenced not only by the tests themselves but also by the underlying test suite, as demonstrated in the motivating example. Therefore, supervised learning is inapplicable and thus we use direct FL feedback to calculate reward to train our model.

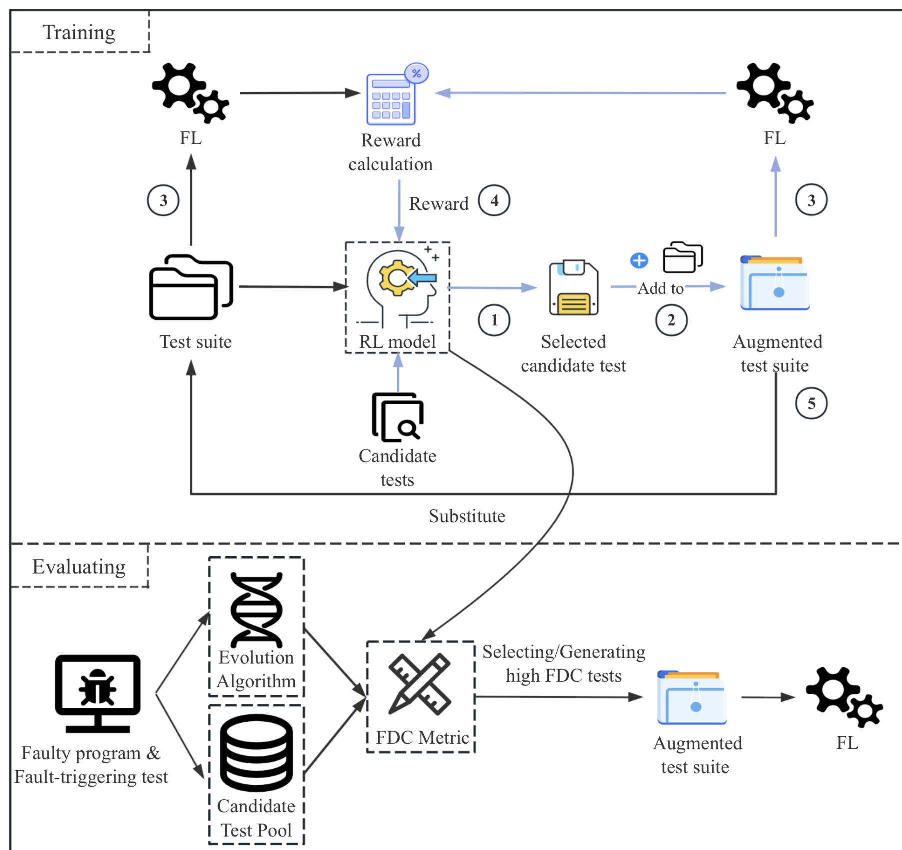

Fig. 2. Boosting FL with RLFDC

Our approach consists of two main stages, i.e., a training stage and an evaluating stage, as shown in Fig. 2. In the training stage, an RL model predicts an FDC value of each candidate test, based on which it sequentially selects individual tests and continuously improves based on the FL feedback with the selected tests. The training stage contains a number of iterations, each containing five steps (i.e., ①-⑤ in Fig. 2., detailed in Section 3.2). At the end of the training stage, a predictive FDC model is built. In the evaluating stage, the trained RL model is leveraged as an FDC metric (RLFDC) to predict the FDC values of candidate tests to boost FL with test selection or generation. In the following, Section 3.1 presents the RL design of our RL-based approach. Section 3.2 presents the training stage of it. Section 3.3 presents the evaluating stage of it.

3.1 RL Design

We first introduce the RL design of our approach. RL is a machine learning framework, which aims to learn a strategy to maximize the **reward**. It involves the interaction of an agent and an environment, through which the agent finds an optimal strategy to maximize the expected reward. The agent performs a series of **actions** based on a series of **states**

of the environment. The environment measures the influence of the series of actions and returns a series of **reward**, which helps the agent continuously adjust its strategy.

In this paper, we build a better FDC metric by modeling the test selection process for FL as an RL process. Initially, there is one test suite T containing only one failing test and a set of candidate tests to be selected to boost the FL performance. During the test selection process, the RL model plays the role of "agent". The coverage information of the whole test suite T (including the original failing test and selected tests) forms the "state". The selection of tests forms the "action", while the results produced by an FL technique establish the basis of "reward". To maximize the reward, the RL model predicts the value of each "action" (i.e., the FDC value of each test), and then selects the "action" with the highest value. Based on the reward (i.e., the influence of the selected tests on the FL results), the model updates its strategy on value prediction (i.e., improving the accuracy of RLFDC on FDC measurement).

3.1.1 RL Model Design. We utilize a deep neural network as our RL model, leveraging its strong representation power and outstanding effectiveness [28, 34]. The input of the network consists of state-action pairs, and the output of the network is the value of each action taken in the current state (i.e., the FDC value of each test considering the underlying test suite T). The input is extracted using coverage information of the test suite and the candidate tests. In particular, the state depends on the test suite T, while the action depends on the test suite and the candidate tests. More design details on the state, action, and reward of our RL-based approach are given in Section 3.1.2. The state is first embedded through an embedding layer consisting of two linear projection layers and two ReLU activation layers as follows.

$$\phi(state) = \text{ReLU}(\text{Linear}(\text{ReLU}(\text{Linear}(state)))) \tag{1}$$

The embedded state is then concatenated with the action vector and is fed into our FDC prediction network.

The FDC prediction network consists of three linear layers. Formally, the output of each layer is computed by

$$\boldsymbol{y}^{(l)} = W^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}, \quad l \in \{1, 2, 3\} \tag{2}$$

where l denotes the l-th layer, $\boldsymbol{y}^{(l)}$ denotes the output of the l-th layer, $\boldsymbol{y}^{(0)} = [\phi(state); action]$, $W^{(l)}$ and $\boldsymbol{b}^{(l)}$ denote trainable variables. In particular, we use the activation function ReLU for the first two layers. We use the output of the last layer, which is a singular value, as the predicted FDC value, i.e., FDC = $y^{(3)}$. The FDC value represents the potential gain of selecting or generating a test for FL, which is further used for test selection and generation in the evaluating stage.

3.1.2 State & Action & Reward Design. In this section, we introduce the state, action, and reward design of our RL-based approach. Let us assume there is an $m \times n$ coverage matrix C denoting the relation between a test suite T and code elements E, where $C[i, j] = 1$ denotes test t_i covers element e_j and $C[i, j] = 0$ otherwise. Given a test t_i, we have its coverage vector ct_i, where ct_i is the ith row of C. Given a code element e_j, we have its execution vector ce_j, where ce_j is the jth column of C. $AG(T)$ denotes the set of ambiguity groups under test suite T. An ambiguity group $ag \in AG(T)$ is a set of code elements that share the same execution vector and thus cannot be distinguished from each other.

(1) State. The state is defined based on the current test suite T. The current test suite is an indispensable factor that should be considered since the FDC value of a test depends not only on the test itself but also on the test suite where it is supposed to be added. We use two features to describe the state space: the number of tests in the test suite (denoted as *num_tests*) and the number of ambiguity groups in the coverage matrix of the test suite (denoted as *num_ag*), i.e., a state can be represented as:

$$state = (num_tests, num_ag) \tag{3}$$

The number of tests demonstrates the scale of the current test suite. The number of ambiguity groups demonstrates to what extent the current test suite distinguishes the code elements in FL. In other words, the two features describe how much effort has been made (i.e., the number of selected tests) and how many difficulties still exist (i.e., the number of ambiguity groups).

(2) Action. The action is defined based on candidate tests. We aim at selecting or generating tests to diagnose faults. From one aspect, tests that split suspicious ambiguity groups, i.e., tests executing different subsets of suspicious ambiguity groups, can help distinguish the elements and thus help diagnose faults. However, solely focusing on splitting ambiguity groups may fall into local optimums. A passing test simultaneously covering several ambiguity groups rather than splitting them may also help pinpoint the fault, because it reduces the suspiciousness of the non-buggy elements in these groups. Building on this intuition and the inspiration for combining the two aspects (i.e., respectively termed as "Cover" and "Split") introduced in previous work [5], we also use two features, **cover** and **split**, but adapt them in a result-agnostic way, to describe the potential value of a candidate test t when added to the underlying test suite T. Thus, the value of a test can be represented by its ability to **cover** suspicious elements and the ability to **split** ambiguity groups:

$$action = (cover, split) \tag{4}$$

For the *cover* feature, we adapt the prior result-aware metric Prox [5] by ignoring the execution results of all tests except the given failing test[2]. The *cover* feature is defined as follows.

$$cover(T, t) = \text{Jaccard}\left(ct_t, ct_{fail}\right) \tag{5}$$

where Jaccard denotes the Jaccard distance between two coverage vectors, ct_t and ct_{fail} denote the coverage vector of the candidate test t and the given failing test respectively. Therefore, *cover* assigns higher priority to tests that can cover more suspicious elements.

For the *split* feature, inspired by Hao et al. [16][3], we measure the ability of a test to distinguish suspicious elements based on: (1) whether this test splits a large ambiguity group containing many suspicious elements and (2) to what extent the split made by this test helps reduce the number of suspicious elements. To measure the former, for an ambiguity group *ag*, we define $priority(ag)$ as the number of code elements in *ag*. Intuitively, a large ambiguity group tends to have a higher probability of containing a buggy element and thus deserves more attention. To measure the latter, for an ambiguity group *ag*, we define $div(t, ag)$ as the impact of the test t on the ambiguity group *ag*. As evenly splitting an ambiguity group tends to result in better FL performance [16], we calculate $div(t, ag)$ based on the size of the smaller subset when t is used to divide *ag* to two subsets based on coverage, i.e., the minimum between $\left|\left\{e_j \in ag \mid ct_t[j] = 1\right\}\right|$ and $\left|\left\{e_j \in ag \mid ct_t[j] = 0\right\}\right|$, where ct_t denotes the coverage vector of the test t. Therefore, $div(t, ag)$ reaches the largest value when the group *ag* is split evenly. Based on $priority(ag)$ and $div(t, ag)$, the *split* feature is defined as follows.

$$split(T, t) = \sum_{ag \in AG(T \cup t)} priority(ag) \cdot div(t, ag) \tag{6}$$

where $AG(T \cup t)$ denotes the set of ambiguity groups formed if the candidate test t is added to the test suite T.

(3) Reward. We compute the reward based on the impact of the test t on the performance of Spectrum-Based Fault Localization (SBFL). An SBFL technique takes the coverage matrix C and test execution results as input and outputs a

[2]Prox calculates the average Jaccard distance between a candidate test and each failing test, as Prox considers the execution results of all tests as input.
[3]Although both works (i.e., prior work [16] and this work) are defined based on the same intuition, the prior work uses execution results of all tests and a different definition of ambiguity group, which makes the two works different.

ranked list of program elements based on the descending order of their suspicious values. The earlier the list ranks the buggy element, the better performance the corresponding SBFL technique achieves. Following this intuition, we use the rank improvement of the buggy element resulting from the selected test t as the reward, which measures the contribution of the test t to fault diagnosis as follows.

$$reward(T, t) = (rank(FL^{T_{init}}) - rank(FL^{T \cup t}))/rank(FL^{T_{init}}) \tag{7}$$

where $rank(FL^{T_{init}})$ denotes the highest rank of the buggy elements[4] using an SBFL technique with the initial test suite T_{init}, $rank(FL^{T \cup t})$ denotes the result after adding test t to the current test suite T. We divide the difference by $rank(FL^{T_{init}})$ to get the relative changed rank. Note that we do not use $rank(FL^{T}) - rank(FL^{T \cup t})$ to calculate the difference because some temporarily "bad" performance test (i.e., resulting in $rank(FL^{T}) < rank(FL^{T \cup t})$) may later benefit FL. The reward is only calculated in the training stage, but is not calculated in the evaluating stage.

3.2 Training Stage

In this section, we present the training stage of our approach. As shown by Fig. 2, the training stage begins with a given test suite T (which contains only a given failing test initially) and a pool of candidate tests T_c. The training stage aims to build an FDC metric RLFDC for later usage (e.g., for a project P). Thus, this process is conducted on some historical versions of P or on other projects.

The training stage contains a number of training iterations. Each training iteration is a test selection process for FL where the RL model plays the role of selection criterion and the FL feedback is used to improve the accuracy of the RL model. In particular, for each training iteration, we repeat five steps: ① The RL model predicts the FDC value of each candidate test in T_c and selects the candidate test with the largest FDC value. ② The selected candidate test t_{sel} is added to T, and removed from T_c. ③ An SBFL technique is applied to T_{init} and $T \cup t_{sel}$, and then the difference between the rank lists is calculated, which is utilized as the reward of RL. ④ The reward is forwarded to the RL model to notify the actual FDC value of the selected test t_{sel}. The RL model uses this FL feedback to adjust its prediction strategy. ⑤ The augmented T (i.e., $T \cup t_{sel}$) is utilized to substitute the previous T, which ends this iteration.

More specifically, we use double Q-learning with experience replay [17] to train our RL model (i.e., Q network in Algorithm 1). As Q-learning performs poorly due to large overestimation of action values [17], we adopt double Q-learning, which uses an extra target Q network to approximate the maximum expected value. We select double Q-learning as it's more suitable for high-accuracy regime tasks [17, 44, 49] and it is one of the most successful value-based, model-free RL algorithms, which suits our scenario where the network needs to predict FDC values (i.e., value-based) and has no prior-knowledge of the environment (i.e., model-free).

[4]Since there may exist multiple buggy elements in a program, we calculate the ranks of all buggy elements and choose the highest one.

Algorithm 1 Double Q-learning with experience replay

Input: initial failing test t_{fail}, candidate test set T_c, capacity N, step K, episode M, updating step C, learning step L, discount factor γ, random selection probability σ.

Output: A trained Q network.

1: Initialize replay memory D to capacity N
2: Initialize step counter $counter$ to 0
3: Initialize Q network with random weights θ
4: Initialize target $\hat{Q}$ network with weights $\theta^- = \theta$
5: **for** $episode = 1$ to M **do**
6:     Initialize test suite $T = \{t_{fail}\}$
7:     initBuggyRank = doFaultLocalization(T)
8:     **for** $k = 1$ to K **do**
9:         $s_k = \text{getState}(T)$
10:         **for** each candidate test $t_i \in T_c$ **do**
11:             $a_{ki} = \text{getAction}(T, t_i)$
12:             $fitness_{ki} = Q(\phi(s_k), a_{ki}; \theta)$
13:         **end for**
14:         Select a random test $t_{sel} \in T_c$ with a probability σ, otherwise select t_{sel} who has the highest $fitness_k$
15:         T_c=$T_c \setminus \{t_{sel}\}$ and $T = T \cup t_{sel}$
16:         curBuggyRank = doFaultLocalization(T)
17:         r_k = reward(initBuggyRank, curBuggyRank)
18:         $s_{k+1} = \text{getState}(T)$
19:         Store transition (s_k, a_k, r_k, s_{k+1}) in D
20:         $counter = counter + 1$
21:         **if** $(|D| = N)$ and $(counter \bmod L == 0)$ **then**
22:             Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
23:             **if** $j + 1 = K$ **then**
24:                 $y_j = r_j$
25:             **else**
26:                 $y_j = r_j + \gamma \max_{a'} \hat{Q}\left(\phi(s_{j+1}), a'; \theta^-\right)$
27:             **end if**
28:             Perform a gradient descent step on $\left(y_j - Q\left(\phi(s_j), a_j; \theta\right)\right)^2$ with respect to the network parameters θ
29:             Every C steps reset $\hat{Q} = Q$
30:         **end if**
31:     **end for**
32: **end for**

Algorithm 1 formally presents the double Q-learning algorithm used in our approach. It takes the initial failing test t_{fail}, the candidate test set T_c, a series of parameters as input, and returns the trained Q network model (i.e., the trained RL model). First, it initializes a memory pool D, step counter *counter*, network Q, and target network $\hat{Q}$ using the input parameters (Lines 1-4). Then, it utilizes M episodes to train the Q network (Lines 5-32). In each episode, it first

initializes the test suite T using t_{fail}, then calculates the initial buggy element rank with an FL technique (Lines 6-7). The algorithm uses K steps (i.e., iterations) to complete an episode, where in each step, a candidate test is selected, and the transition is stored (Lines 8-31). For each step, the state for the test suite and the action for each candidate test can be calculated using Eq. 3 and Eq. 4, then the fitness for each candidate test can be calculated using Eq. 2 (Lines 9-13). To ensure that the model does exploration rather than greedily falls into local optimums, the test t_{sel} is randomly selected with a probability of σ and is selected based on the fitness otherwise (Line 14). The selected test t_{sel} is removed from the candidate test set and added to the test suite T (Line 15). Then we apply the FL technique with the augmented test suite to get the current buggy element rank, and calculate the reward according to Eq. 7 (Lines 16-17). The transition for this step is stored in D, and a counter records the number of steps (Lines 18-20). When the replay memory reaches its capacity, for every L steps, we sample a random minibatch from D and update the parameters accordingly (Lines 21-30). The expected reward y is calculated using the target network $\hat{Q}$ (Lines 23-27). We use mean-squared error to calculate the difference between the predicted value and the expected reward y (Line 28). The parameters of $\hat{Q}$ are fixed and updated every C steps by being substituted with parameters from Q (Line 29).

3.3 Evaluating Stage

As shown in Fig. 2, the evaluating stage presents the usage of RLFDC, which is the trained RL model resulting from the training stage. In particular, in the evaluating stage, we apply RLFDC to two FL-oriented scenarios, i.e., test selection and generation for FL. **Note that in the evaluating stage, we do not feed any execution results of tests as input to the RL model** (FL results are only utilized in the reward calculation, which is only needed in the training stage). Thus, the trained RL model, i.e., RLFDC, is a result-agnostic metric.

Given a fault-triggering test and a faulty program, test selection for FL aims to select some tests from a given candidate test pool (without oracles) and use them to augment the given test suite for FL. In this scenario, we use RLFDC to guide test selection, i.e., iteratively selecting a test with the largest FDC value given by RLFDC and adding the selected test to augment the test suite, which initially contains only the given fault-triggering test. The process is repeated until some termination condition is satisfied, e.g., achieving some FL performance goals or reaching some resource limits.

Given a fault-triggering test and a faulty program, test generation for FL aims to generate more tests for FL automatically, and thus RLFDC can serve as a fitness function in search-based test generation. For example, EVOSUITE is a test generation tool based on evolutionary algorithm, and we integrate RLFDC with it to facilitate test generation for FL. In particular, in the evolutionary algorithm, tests with the largest FDC value predicted by RLFDC are selected, and are used as parents to produce offsprings (i.e., more tests).

4 EVALUATION SETUP

In this section, we evaluate RLFDC in FL-oriented test case selection and generation tasks, as prior work does [5, 9, 11, 40]. In particular, in the former task, we use RLFDC and other metrics as selection criteria to select tests and evaluate the FL performance with the corresponding selected test cases. In the latter task, we use RLFDC and other metrics as fitness functions to guide test generation and evaluate the FL performance with the corresponding generated tests. To sum up, we aim to answer five research questions (RQs).

RQ1. **How does RLFDC perform on human-written test selection?** To answer this RQ, we select tests based on human-written tests using different FDC metrics, and compare the FL performance of the various augmented test suites.

This serves as an ideal evaluation setting, featuring tests written by experienced developers with deep knowledge of the projects [5].

RQ2. **How does RLFDC perform on automatically generated test selection?** Different from RQ1, in this RQ we select tests from a test pool automatically generated by a search-based test generation tool EVOSUITE. Besides, to further investigate the generalizability of FDC metrics, we also evaluate the test selection task on a test pool automatically generated by Randoop [37].

RQ3. **How does RLFDC perform on automated test generation?** To answer this RQ, we utilize various metrics (including RLFDC) as fitness functions in EVOSUITE, and compare the FL performance of their generated tests.

RQ4. **How does RLFDC perform in cross-project scenario?** The proposed RLFDC is built based on a training process that simulates test selection for FL. This process can be conducted on cross-version scenario and cross-project scenario. The evaluation for RQ1-RQ3 is conducted in the former scenario, whereas in this RQ, we investigate the performance of RLFDC in the latter scenario.

RQ5. **How do different components of RLFDC contribute to its effectiveness?** We perform an ablation study to validate the design decisions of our approach. To answer this RQ, we investigate the impact of the designed features, the network structure, and the training algorithm to the effectiveness of RLFDC by conducting experiments on different variants of RLFDC.

Table 4. Subjects

Subject	# Faults	kLoC	Avg. # Tests		Avg. # Methods	
			Total	Failing	Total	Buggy
Commons-**lang**	62	22	1,819	2.0	2,180	1.5
Commons-**math**	106	85	2,524	1.7	4,544	1.7
JFree**Chart**	26	96	1,814	3.5	7,478	4.5
Joda-**Time**	26	28	3,918	2.8	3,804	2.0
Closure compiler	131	90	7,211	2.6	8,203	1.7
Total	351					

4.1 Subjects

In this study, following prior work [5], we use five projects from Defects4J (V2.0.0) [26], a Java benchmark widely used in FL [28, 33, 34, 57]. Each project has some buggy versions, while each buggy version contains one real fault in the program and at least one test revealing the existence of this fault. All the tests provided by Defects4J are manually constructed. We remove six deprecated faults from our study because they are irreproducible due to behavioral changes introduced under Java 8 [25]. Table 4 presents the statistics of the projects used in this study. Column "Subject" presents the full name of the subject. Column "# Faults" presents the number of buggy versions for each subject. Columns "Total" and "Failing" under "Avg. # Tests" present the average number of total tests and failing tests for each buggy version. Columns "Total" and "Buggy" under "Avg. # Methods" present the average number of total methods and buggy methods for each buggy version. We use the abbreviation of project names (stressed in bold font) in the remainder of this paper. Note that in RQ2 and RQ3, we remove another 103 faults because EVOSUITE cannot generate compilable tests for the corresponding buggy programs.

4.2 Baselines

In the literature, the metrics to measure FDC are classified into result-aware and result-agnostic metrics based on the utilization of test results [5]. To conduct a comprehensive comparison, we include three state-of-art result-agnostic metrics EntBug [11], DDU [40] and TfD [9], and one state-of-the-art result-aware metric FDG [5] as our baselines. Note that our RLFDC is a result-agnostic metric.

FDG [5] is a recently proposed result-aware metric that employs SBFL scores to learn which part of the program requires additional diagnostic information. It is shown to perform the best on the test selection task for FL [5]. FDG uses the SBFL scores to assign weights to ambiguity groups and program elements, where the execution result of the newly-added test is needed. $\mathrm{FDG}(T,t) = \alpha \cdot (1 - \frac{1}{n-1} \cdot \sum_{ag \in AG(T \cup \{t\})} p(ag) \cdot (|ag| - 1)) + (1-\alpha) \cdot \frac{\sum_{j=1}^{n} w_j \cdot ct_t[j]}{n}$, where w_j is the SBFL score of e_j, ct_t is the coverage vector of test t, n is the number of code elements, α is a hyperparameter that needs to be tuned, $p(ag) = \sum_{e_j \in ag} p\left(e_j\right)$, $p\left(e_j\right) = w_j / \sum_i w_i$.

EntBug [11] is a result-agnostic metric based on entropy. In particular, EntBug applies probability theory concepts to minimize the uncertainty in the diagnostic ranking, and is defined using the density of the coverage matrix. $\mathrm{EntBug}(T) = 1 - |1 - 2 \cdot \rho(T)|$, where $\rho(T) = \sum_{i=1}^{m} \sum_{j=1}^{n} C[i,j] / (m \times n)$. Recall that coverage matrix C has the shape of $m \times n$ and $C[i,j] = 1/0$ denotes whether test t_i covers/uncovers program element e_j.

TfD [9] is a result-agnostic metric based on the number of ambiguity groups in the test suite T, i.e., TfD = $|AG(T)|$. It is designed with the intuition that the more ambiguity groups a coverage matrix has, the less ambiguity the program spectrum has.

DDU [40] is a combined, result-agnostic metric. $\mathrm{DDU}(T) = density(T) \times diversity(T) \times uniqueness(T)$, where $density(T)$ is equal to $\rho(T)$, $uniqueness(T)$ is equal to $|AG(T)|/n$, $diversity(T)$ is the Gini-Simpson index [24] among the rows of the coverage matrix. DDU is proposed to combine these three key properties of a coverage matrix for achieving more accurate FL.

In summary, these metrics are all hand-crafted formulas designed based on some heuristics, which leaves room for harnessing the power of machine learning to learn a more accurate metric. Note that result-aware metrics (e.g., FDG) measure FDC values with test results, which limits their application for test generation.

4.3 Implementation Details

4.3.1 Baselines. We implement TfD, EntBug, DDU, and FDG for test selection using the implementation from prior work [5]. We implement EntBug and DDU for test generation using their published EVOSUITE artifacts [11, 40]. We do not include TfD for test generation since TfD is implemented on a test optimization tool utilizing bacteriologic algorithm [9] rather than EVOSUITE, and prior work [40] has shown that DDU is more powerful than TfD in terms of test generation.

4.3.2 Hyperparameters. To answer RQ1, RQ2, RQ3, and RQ5, we use five-fold validation[5]. In particular, we split each project's buggy versions in Defects4J (V2.0.0) into five folds: each time, one fold is selected as the testing set, and the rest four folds are used as the training set. In the training stage, we train each model for 30 epoches based on the human-written tests of the corresponding versions, resulting in our FDC metric, RLFDC. To train our model, we use a learning rate of 0.001 globally, and a batch size of 32 for memory sampling. The layer widths are (16, 16) for the embedding layer and are (16, 32, 1) for the FDC prediction network. The other parameters are empirically

[5]Note that some deep learning-based FL techniques [28, 30, 34] are evaluated in leave-one-out validation, i.e., use one buggy version as testing data and the remaining versions as training data, which is reported to cause overfitting and reduce applicability [52]. To address this concern, we use five-fold validation in the evaluation.

set as $K = 10, N = 100, C = 20, L = 5, \gamma = 0.9, \sigma = 0.1$. In the evaluating stage, we evaluate the performance of the four compared metrics and RLFDC on the remaining buggy versions. Following prior work [5], we evaluate the performance of metrics in test selection and generation. We gradually select n tests for a given failing test and report the corresponding FL results in the selection scenario on human-written and automatically generated tests. We report the FL results by selecting $1, 2, \ldots, n$ tests. In the generation process, we integrate these metrics into EVOSUITE, generate tests for a given failing test, and report the corresponding FL results. In our evaluation, n is set to ten by following prior work [5] because prior work [5] has shown that on these subjects with only ten addition tests, FL [5] may achieve comparable performance as the full test suite (62% of acc@1 and 80% of acc@10).

The experiments for RQ2 and RQ3 are conducted on tests generated by EVOSUITE or Randoop, which contain randomness from the input seeds and the searching process. To ease reproducibility and control the influence of randomness, we use fixed input seeds, repeat these experiments three times, and report the average FL results. We use the RLFDC built in RQ1 for RQ2 (i.e., selecting automatically generated tests) and RQ3 (i.e., guiding EVOSUITE to generate tests).

4.3.3 Oracle Labelling. The human-written tests used in RQ1, RQ4, and RQ5 have test oracles, while the EVOSUITE-generated tests used in RQ2 or RQ3 do not. We utilize EVOSUITE in a regression way to handle the oracle problem [5, 42]. Following prior work [5], we use EVOSUITE to generate tests on the buggy versions of Defects4J and execute them on the fixed versions. The tests failing on the fixed versions are labeled as "failing tests".

4.3.4 EVOSUITE Configuration. We use the `generateTests` option for EVOSUITE following previous work [11]. For each failing test, we initially measure its coverage. Subsequently, we calculate the FDC value for each individual in the population (i.e., each generated test) using a fitness function considering their overlap with the failing test's coverage. Individuals with the highest fitness values are then selected as parents to produce offspring. In other words, for each failing test, we generate a corresponding test suite that can diagnose the fault it reveals. We set the overall search time budget as 120 seconds. Besides, we configure the `budget_for_each_generation` option to 2 seconds for each fitness function evaluation, allowing the algorithm to produce 60 generations within the allocated 120 seconds.

4.3.5 Time costs. Table 5 presents the average time costs for our approach on each version of the five projects. Row "Train" and Row "Test" present our proposed approach's training and testing time. In particular, the testing time for RLFDC refers to the total time RLFDC used to predict the FDC values of all tests. From the table, the training time ranges from 3 minutes to 39 minutes, which is tolerable since training is conducted offline. The testing process costs only several seconds on average, indicating that RLFDC can efficiently predicts FDC values of tests. Overall, our approach is a lightweight RL-based technique with a small overhead.

4.3.6 Environment. All the experiments are conducted on a workstation with 2 Intel Xeon Gold 5218R CPUs, 256GB RAM, and four 24G GPUs of GeForce RTX 3090, running Ubuntu 18.04 x64 OS. We build our experiment on PyTorch V1.8.1 [1].

4.4 Fault Localization Techniques

As the FDC metrics (including RLFDC) are proposed to boost FL, following previous work [5], we investigate the performance of studied FDC metrics through a widely-used Spectrum-based Fault Localization (SBFL) technique, Ochiai with aggregation [43]. SBFL is one of the most widely studied FL techniques because of its simple input and effectiveness [3, 47], and SBFL scores are often used as features for more complicated FL techniques [28, 57], which motivates us to include it as our FL technique.

Table 5. Efficiency of RLFDC (seconds)

Subject	Chart	Time	Lang	Math	Closure
Train	634.41	423.72	180.67	561.78	2,338.51
Test	0.51	0.28	0.08	0.27	1.35

In particular, Ochiai with aggregation (abbreviated as Ochiai-agg) first computes the line-level Ochiai scores based on the coverage information of tests and then aggregates them at the method level with the highest score of all the lines in the method [43]. To measure the performance of the compared FDC metrics, we feed the original given failing test along with the tests selected or generated based on an FDC metric to Ochiai-agg, and report the FL results.

4.5 Evaluation Metrics

Following prior work [5, 28, 30, 34], we use Mean Average Precision (mAP) and Top-n Accuracy (acc@n) to evaluate the FL accuracy.

mAP is widely used in the Information Retrieval (IR) task while also adopted for measuring FL accuracy [5]. In our FL context, "Average Precision" is calculated based on the rank of multiple buggy elements, while "Mean" stands for averaging all the produced rankings. The calculation of mAP can be formally written as:

$$mAP = \frac{1}{|F|} \sum_{f \in F} \left(\frac{1}{|f_B|} \sum_{b \in f_B} \frac{num_higher_buggy_rank_b}{buggy_rank_b} \right) \tag{8}$$

where F denotes the set of considered buggy programs, f_B denotes the set of buggy elements (i.e., buggy methods in our setting) in the buggy program f, $buggy_rank_b$ denotes the rank of the buggy element b in the FL produced ranking list, $num_higher_buggy_rank_b$ denotes the number of buggy elements ranked higher than the buggy element b (including b). mAP presents the overall FL accuracy considering the average performance on various buggy programs and the ranks of multiple buggy elements.

acc@n is a widely-adopted metric for measuring FL performance [28, 34]. acc@n counts the number of buggy programs where at least one of the buggy elements is ranked within the top n locations. The metric directly reflects how many elements developers should inspect before finding the first buggy element with FL.

5 RESULTS AND ANALYSIS

5.1 RQ1. Human-written test selection

Using human-written tests of Defects4J as the candidate test pool, we iteratively select tests with the five studied FDC metrics. In particular, for each failing test of a given buggy program (which composes an initial test suite), we iteratively select another 10 tests using an FDC metric and thus construct an augmented test suite composed of 11 human-written tests. To evaluate whether an FDC metric performs well (i.e., selecting tests to improve FL), we apply the SBFL technique Ochiai-agg to the constructed test suite and analyze its FL performance in terms of mAP and acc@n.

Compared with result-agnostic metrics, a result-aware metric uses extra inputs (i.e., test results), which are not always available in practice due to the widely-known test oracle problem. That is, the comparison between result-aware and result-agnostic metrics [5] seems to be unfair. In the remaining sections, we not only conduct a fair comparison in result-agnostic metrics (i.e., comparing RLFDC against the existing result-agnostic metrics), but also conduct an unfair comparison between RLFDC and the state-of-the-art result-aware metric FDG to learn their performance gap.

Fig. 3 shows the trend of mAP values for selecting n tests (where $n = 1, 2, \ldots, 10$), where the horizontal axis represents the number of selected tests. As result-aware and result-agnostic metrics differ in their inputs, we distinguish them

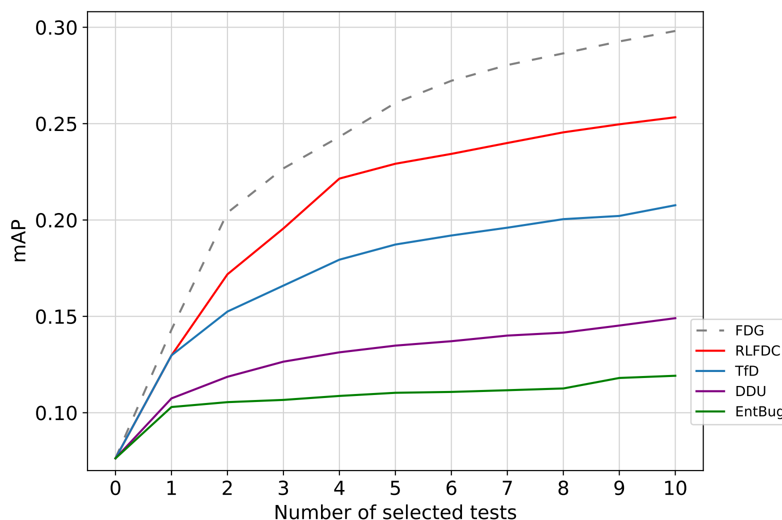

Fig. 3. mAP values on human-written tests

through dashed and solid lines in the figure. We observe that: (1) The FL accuracy increases with more tests. (2) With the same number of tests, the FL accuracy may vary depending on different test suites constructed by different metrics. These two observations demonstrate that the number and the FDC of tests influence FL performance to a large extent, indicating the importance of accurate FDC metrics in test selection.

From the figure, RLFDC is the best result-agnostic metric, outperforming TfD, DDU, and EntBug. When the number of selected tests increases from 0 to 10, RLFDC consistently outperforms other result-agnostic metrics and makes the mAP value increase from 0.076 to 0.253 (an increase of 233%). Besides, as expected, the result-aware metric FDG outperforms all the result-agnostic metrics.

Table 6. acc@n values on human-written tests

		@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10
Metric		EntBug(mAP=0.119)				DDU(mAP=0.149)				TfD(mAP=0.208)				RLFDC(mAP=**0.253**)				FDG(mAP=0.298)			
	Init.	4	30	45	65																
# Tests	1	9	40	61	80	10	43	64	85	**21**	51	67	84	**21**	**58**	**79**	**103**	24	64	87	109
	2	9	41	63	85	12	49	74	99	28	67	81	100	**32**	**75**	**105**	**136**	41	100	124	155
	3	9	40	64	90	14	53	76	107	32	71	88	112	**40**	**88**	**112**	**146**	48	107	135	175
	4	9	40	64	94	15	54	77	112	33	81	95	127	**46**	**104**	**129**	**168**	52	115	145	190
	5	9	41	67	95	15	54	79	115	35	82	98	130	**47**	**108**	**137**	**181**	57	121	158	204
	6	9	41	67	96	15	55	79	118	35	87	100	134	**47**	**112**	**141**	**183**	60	130	164	212
	7	9	41	68	96	15	56	81	123	36	89	105	142	**48**	**115**	**144**	**187**	64	135	170	221
	8	9	41	68	99	15	56	81	125	37	89	109	145	**49**	**117**	**149**	**192**	65	139	174	219
	9	11	43	68	99	16	58	82	127	37	91	111	149	**49**	**121**	**155**	**195**	66	145	178	220
	10	12	44	68	101	17	59	84	127	39	93	111	149	**50**	**124**	**156**	**203**	68	148	179	222
	Full	110	208	250	277																

We further present the acc@n values of Ochiai-agg based on human-written tests selected by each metric in Table 6. The last row presents the FL results using all human-written tests of each subject. As the proposed RLFDC is a result-agnostic metric, for any number of selected tests, we highlight the best results among the four result-agnostic metrics (i.e., EntBug, DDU, TfD, and RLFDC) with the bold font. The last four columns give the acc@n values of the result-aware metric FDG for reference. The mAP values for selecting ten tests are also shown in the brackets next to the metric names. From the table, RLFDC performs the best among all result-agnostic metrics, achieving the highest acc@n (n=1,

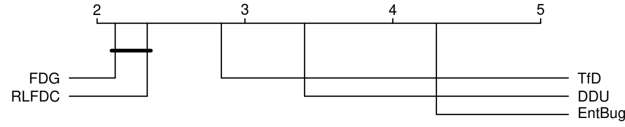

Fig. 4. Critical difference plot

3, 5, 10) when selecting any number of tests and the highest mAP. In particular, RLFDC achieves 50 in terms of acc@1 and 203 in terms of acc@10 when selecting ten tests. Compared to the second-best result-agnostic metric, TfD, the improvement on acc@1 and acc@10 is 28.2% and 36.2%, respectively. With ten tests selected by RLFDC, the acc@1, acc@3, acc@5, and acc@10 improve 11.5 times, 3.13 times, 2.47 times, and 2.12 times respectively compared to the initial test suite. When compared to the full test suite (which contains over 1K tests), the ten tests selected by RLFDC achieve 45.4% and 73.3% of its acc@1 and acc@10 values, respectively.

To investigate whether these metrics perform significantly differently in terms of mAP, we conduct a non-parametric ANOVA analysis utilizing the Friedman test with the Iman and Davenport extension [20], which is robust to non-normality. The analysis result rejects the null hypothesis of equal performance (with $p < 2.2 \times 10^{-16}$), indicating that at least one metric performs significantly differently from the others. Furthermore, to investigate which metrics perform significantly better, we run a post hoc test (i.e., Friedman post hoc test with Bergmann and Hommel's correction [10]) to compare each pair of metrics. The analysis results are given by Fig. 4, where each metric is placed on an axis according to its mean rank among the five metrics across all outcomes. Metrics with larger mAP are placed on the left. The metrics not grouped with a horizontal line are significantly different ($p < 0.05$). From the figure, the proposed result-agnostic metric RLFDC and the result-aware metric FDG do not perform significantly differently, but each of them significantly outperforms other metrics.

Table 7. acc@n values on human-written tests of ten additional subjects

		@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10
Metric		EntBug(mAP=0.175)				DDU(mAP=0.246)				TfD(mAP=0.237)				RLFDC(mAP=**0.285**)				FDG(mAP=0.312)			
	Init.	4	14	18	29																
# Tests	1	8	23	37	64	8	18	25	46	**15**	30	41	58	14	**34**	**48**	**80**	12	31	40	67
	2	10	27	42	71	19	32	53	86	21	35	52	85	**24**	**52**	**75**	**105**	18	49	67	94
	3	13	30	50	80	26	49	64	99	26	46	61	94	**30**	**64**	**91**	**119**	26	64	85	119
	4	17	36	59	90	30	55	77	108	27	52	70	104	**35**	**72**	**96**	**127**	40	80	103	146
	5	18	37	60	93	32	63	84	119	31	60	83	110	**37**	**74**	**95**	**133**	43	94	115	156
	6	18	39	62	96	34	64	87	124	36	66	87	120	**39**	**80**	**104**	**140**	50	100	121	160
	7	18	41	64	99	35	68	92	128	36	68	92	126	**41**	**87**	**108**	**144**	49	106	129	164
	8	20	48	69	106	36	70	95	134	36	66	92	130	**46**	**90**	**113**	**151**	50	108	133	167
	9	20	48	70	111	39	74	101	137	36	68	98	137	**47**	**95**	**114**	**152**	50	112	136	172
	10	22	50	72	117	40	76	103	139	38	74	100	140	**46**	**100**	**119**	**156**	54	112	138	173
	Full	80	143	167	196																

To further validate the generalizability of RLFDC, apart from the five subjects widely used for evaluating fault localization techniques [27, 28, 30, 48, 57], we also include ten additional subjects from Defects4J V2.0.0 in this RQ for further evaluation. We conduct an experiment with ten additional subjects, encompassing a total of 264 bugs, from Defects4J V2.0.0[6]. Table 7 presents how RLFDC and the four baselines perform in human-written test selection for the ten subjects. From the table, RLFDC consistently outperforms all other result-agnostic metrics, achieving the highest acc@n (n=1, 3, 5, 10) across various number of selected tests, as well as the highest mAP. In particular, RLFDC achieves

[6]We exclude the first 45 bugs in Jsoup and all bugs in Gson/JacksonCore because of reproducing problem. This problem can also be seen in previous work [34, 52].

46 in terms of acc@1 and 156 in terms of acc@10 when selecting ten tests. Compared to the second-best result-agnostic metric, TfD, the improvement on acc@1 and acc@10 is 21.1% and 11.4%, respectively. With ten tests selected by RLFDC, the acc@1, acc@3, acc@5, and acc@10 improve 10.5 times, 6.1 times, 5.6 times, and 4.4 times respectively compared to the initial test suite. When compared to the full test suite, the ten tests selected by RLFDC achieve 57.5% and 79.6% of its acc@1 and acc@10 values, respectively.

5.2 RQ2. Automatically generated test selection

Table 8. acc@n values on tests automatically generated by EVOSUITE

	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10
Metric	EntBug(mAP=0.215)				DDU(mAP=0.230)				TfD(mAP=0.226)				RLFDC(mAP=**0.236**)				FDG(mAP=0.239)			
Init.	4	26	39	57																
# Tests 1	16	47	58	75	**19**	**51**	**62**	**77**	16	40	53	73	14	49	60	76	20	52	66	84
2	20	53	63	80	**26**	**56**	**70**	86	22	46	62	78	23	55	68	**88**	26	59	74	95
3	24	53	64	86	**29**	**58**	**72**	91	25	50	63	84	27	57	71	**97**	31	63	78	104
4	25	55	64	89	**30**	59	73	96	26	55	67	87	29	**60**	**74**	**101**	34	65	81	110
5	26	56	65	91	30	60	73	98	29	57	69	89	**31**	**62**	**76**	**104**	34	67	83	112
6	26	57	67	92	31	62	75	100	30	59	70	93	**33**	**63**	**78**	**105**	34	67	83	114
7	27	58	68	94	31	63	76	103	31	60	71	96	**33**	**65**	**80**	**108**	34	67	84	114
8	28	58	69	95	32	63	78	106	32	62	73	101	**33**	**66**	**81**	**111**	34	67	84	113
9	29	59	71	98	33	64	79	107	33	63	76	105	**34**	**67**	**83**	**113**	35	67	83	113
10	30	59	72	100	33	64	80	107	33	63	76	107	**34**	**68**	**84**	**113**	35	68	84	113
Full	37	70	88	117																

In this section, we still evaluate the studied metrics in the test selection task, but the candidate test set is automatically generated by a test generation tool. This practical scenario aims to balance the effort of human labelling and FL performance, where FDC metrics are utilized to select tests from the generated test suite. In particular, for each buggy program, we use EVOSUITE or Randoop to generate tests for 120 seconds. The generated tests are then used as a candidate test pool. For each failing test, we iteratively select 10 tests from the generated test suite based on a studied metric, then use the 11 tests as the input of Ochiai-agg.

Table 8 presents the acc@n values with selected EVOSUITE-generated tests. From the table, on EVOSUITE-generated tests, RLFDC outperforms all result-agnostic metrics in terms of acc@n (n=1, 3, 5, 10) and mAP when selecting more than three tests. With ten tests selected by RLFDC, the acc@1 and acc@10 improve 7.5 times and 1.0 times respectively compared to the initial test suite. Moreover, the ten tests selected by RLFDC achieve 91.9% and 96.6% of the full test suite's acc@1 and acc@10, respectively, indicating that RLFDC helps developers save the cost of labeling oracles while still maintaining satisfactory FL performance by selecting tests with high FDC values. It is also worth noting that on EVOSUITE-generated tests, RLFDC has competitive performance with the result-aware metric FDG. Compared with Table 6, the performance of different metrics decreases and the advantage of RLFDC over other metrics becomes less obvious. This may be attributed to the quality of automatically generated tests. The quality of automatically generated tests often falls short of that produced by human-written tests. Developers possess a deep understanding of their projects, including knowledge of error-prone inputs and suspicious conditions, a grasp of each method's functionality, and an awareness of the correct sequence for invoking methods. In contrast, test generation tools lack this intimate project understanding. Consequently, they might generate irrelevant inputs, ineffective invocations, and call methods in an incorrect sequence, which can impair the fault diagnosis capability of the generated tests. Therefore, there is a lack of tests with high FDC values available for selection, which is also reflected by the steep fall of FL performance using the full test suite.

Table 9. acc@n values on tests automatically generated by Randoop

		@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10
Metric		EntBug(mAP=0.239)				DDU(mAP=0.291)				TfD(mAP=0.295)				RLFDC(mAP=**0.305**)				FDG(mAP=0.302)			
	Init.	2	22	28	43																
# Tests	1	10	32	41	52	13	**33**	**44**	**55**	**14**	**33**	39	51	11	**33**	43	52	17	36	46	54
	2	11	33	44	53	18	**39**	**50**	**58**	**19**	38	45	55	**19**	38	48	**58**	23	42	52	59
	3	11	33	44	54	19	**42**	**52**	**60**	21	39	48	56	**22**	**42**	50	**60**	24	45	53	61
	4	12	34	44	54	20	**43**	**54**	61	22	41	51	58	**25**	**43**	53	**62**	24	47	54	63
	5	13	34	44	55	20	**44**	53	61	23	42	52	59	**25**	**44**	**55**	**64**	24	46	55	64
	6	14	34	45	55	21	**45**	54	62	23	43	53	60	**24**	**45**	**56**	**66**	23	47	56	66
	7	14	35	45	55	23	**45**	54	63	23	44	53	60	**25**	**45**	**56**	**65**	24	48	56	66
	8	16	35	45	56	23	45	54	63	24	45	53	61	**26**	**46**	**56**	**66**	26	48	56	67
	9	16	36	45	56	23	**46**	54	64	25	**46**	53	62	**27**	**46**	**56**	**67**	26	48	55	67
	10	17	36	45	56	24	46	55	64	25	46	54	62	**28**	**47**	**56**	**67**	26	49	56	67
	Full	30	48	59	71																

Since EVOSUITE cannot generate compilable tests for 103 faults, to mitigate this limitation and investigate the generalizability of FDC metrics, we employed Randoop [37] across the five projects to generate tests. Specifically, we utilized the `gen_tests.pl` script provided by Defects4J to generate tests for relevant classes with a total time budget of 120 seconds. We conducted the experiment three times using different random seeds and reported the averaged results. Overall, Randoop generates compilable tests for 133 faults, and the FL results using these tests are displayed in Table 9. From the table, RLFDC continues to outperform all other result-agnostic metrics in terms of acc@n and mAP. When compared to the result-aware metric FDG, RLFDC not only achieves competitive acc@n but also a higher mAP (0.305 vs 0.302). Notably, with ten tests selected by RLFDC, the acc@1 and acc@10 improve 13.0 times and 0.6 times respectively, compared to the initial test suite. Furthermore, the ten tests selected by RLFDC achieve 93.3% and 94.4% of the full test suite's acc@1 and acc@10, respectively.

5.3 RQ3. Automated test generation

In this section, we evaluate the studied metrics by using them as guidance for test generation. EVOSUITE is a search-based test generation tool based on an evolutionary algorithm, which uses a fitness function to guide the evolution direction. EVOSUITE has several default fitness functions (e.g., line coverage). As this paper targets FL, we replace the fitness function of EVOSUITE with the studied metrics (i.e., DDU, EntBug, and RLFDC), and investigate whether the resulting test generation techniques (abbreviated as EVO-DDU, EVO-EntBug, EVO-RLFDC) can generate tests with high FDC. Besides, we include another baseline, i.e., the EVOSUITE with line coverage as the fitness function, which we refer to as EVO-line. Note that in RQ3 we discard TfD and FDG because prior work [40] shows that DDU is more powerful than TfD in terms of test generation. The result-aware metric FDG cannot be applied in this study because FDG requires test oracles, which is impractical to get for the whole population during the evolution process.

We compare the FL performance of all tests generated based on various FDC metrics. In particular, we run each generation technique, i.e., EVO-line, EVO-DDU, EVO-EntBug, and EVO-RLFDC, for 120 seconds and feed the corresponding generated test suite to Ochiai-agg. Their FL results are given by Row "Full" in Table 10. According to this row, EVO-RLFDC achieves an improvement of 48.6% and 8.5% in terms of acc@1 and acc@10 compared to EVO-line, and an improvement of 3.8% and 1.6% in terms of acc@1 and acc@10 compared to the state-of-the-art technique, DDU. That is, RLFDC outperforms all result-agnostic metrics in terms of acc@n (n=1, 3, 5, 10) when generating tests for FL within a given time limit. However, we also observe that these generation techniques generate various numbers of tests within 120 seconds, i.e., EVO-line, EVO-DDU, EVO-EntBug, and EVO-RLFDC generate 42,482, 24,347, 13,026, and 24,271 tests respectively. We wonder whether the various number of generated tests would influence the FL performance.

Table 10. acc@n values on tests generated under the guidance of different metrics

		@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10	@1	@3	@5	@10
Metric		EVO-line(mAP=0.236)				EVO-DDU(mAP=0.258)				EVO-EntBug(mAP=0.261)				EVO-RLFDC(mAP=**0.266**)			
# Tests	Init.	4	26	39	57												
	10	34	68	84	113	41	75	96	**121**	41	76	94	117	**43**	**81**	**98**	**121**
	Full	37	70	88	117	53	85	102	125	50	81	99	122	**55**	**86**	**103**	**127**

Table 11. acc@10 values in cross-project scenario

Train Subj.	Metrics	Chart	Time	Lang	Math	Closure
Chart	RLFDC	-	12	58	80	28
Time		20	-	55	79	26
Lang		20	8	-	83	24
Math		18	9	58	-	25
Closure		19	14	54	76	-
-	RLFDC	21	15	57	81	29
-	TfD	17	7	53	52	21

To address this concern, we conduct another experiment to align the number of tests used for FL. In particular, for each of these generated test suites, we iteratively select ten tests with the highest FDC values[7], and then feed the selected ten tests and the initial failing test to Ochiai-agg. The corresponding results are given by Row "10" in Table 10. From the table, RLFDC again outperforms all result-agnostic metrics in terms of both acc@n (n=1, 3, 5, 10) and mAP (given in brackets) in test generation for FL, even if we control the number of tests. In particular, with ten tests, EVO-RLFDC successfully helps the FL technique to localize 9 more faults within top-1, and 8 more faults within top-10 compared to EVO-line, and achieves an improvement of 4.9% and 3.1% in terms of acc@1 and mAP compared to EVO-DDU. In summary, the experimental results demonstrate that RLFDC can help guide the EVOSUITE to generate tests with high FDC and enhance the upper bound of FL.

5.4 RQ4. In the cross-project scenario

Our approach builds a better metric RLFDC through a training process, which can not only be conducted in cross-version scenario (RQ1-3), but also be conducted in cross-project scenario. To evaluate RLFDC in the latter scenario, for each project, we train a RL model using the human-written tests, resulting in five different RLFDC. Each RLFDC is applied to human-written test selection for FL on other four projects, and the corresponding FL results are reported by Table 11. Due to space limitation, we only report the acc@10 values. The first column presents the five subjects used for training. The 1st to 5th rows present the performance of RLFDC trained on these five subjects respectively. The 6th row presents the performance of RLFDC when being trained and used in different versions of a project as in RQ1, and the 7th row presents the performance of the state-of-the-art result-agnostic metric TfD for reference.

From the table, the acc@10 values in the 1st to 5th rows are all higher than the last row, indicating that even trained on other projects, RLFDC still outperforms the state-of-the-art result-agnostic metric TfD. Besides, the results of RLFDC in the 1st to 5th rows are usually smaller than the ones in the 6th row, indicating that RLFDC performs worse when it is built and used on different projects. The first observation is as expected since RLFDC can learn a more accurate strategy from the FL feedback compared to the fixed heuristic-based TfD. The second observation is also as expected, since in the cross-version scenario testing data share similar features with training data, while in the cross-project

[7]Here we use RLFDC as the selection criterion because RLFDC is demonstrated to be the best result-agnostic metric according to RQ1 and RQ2.

scenario testing data may have different characteristics. In summary, the results demonstrate that RLFDC still performs excellently and is the state-of-the-art result-agnostic metric even in the cross-project scenario.

5.5 RQ5. Ablation study

In this section, we perform an ablation study to investigate the contribution of different components of RLFDC, which are the designed features, the network structure, and the training algorithm. Besides, we systematically remove the RL component and use a weighted sum of "split" and "cover" similar to FDG [5] to serve as a baseline to investigate the performance contribution of the entire RL component.

In particular, We design different variants of RLFDC and compare their effectiveness with RLFDC in the human-written test selection scenario to justify the design choices of our approach. The variants of RLFDC are as follows. (1) RLFDC_{cover} and RLFDC_{split} directly use the calculated action features $cover$ (Eq. 5) and $split$ (Eq. 6) as the FDC values of tests, respectively. While RLFDC combines these two features to comprehensively predict the FDC values of tests, we investigate how these two features perform alone. (2) $\text{RLFDC}_{simpleNet}$ removes the embedding layer of the RL model, i.e., the state and the action features are directly concatenated and fed into the FDC prediction network to produce the predicted FDC value. (3) $\text{RLFDC}_{regularQ}$ uses regular Q-learning rather than double Q-learning as the training algorithm to train the RL model. (4) $\text{RLFDC}_{weight_\alpha}$ uses a weighted sum of "split" and "cover", where α is the weight for "split" and $(1-\alpha)$ is the weight for "cover", i.e., $\text{RLFDC}_{weight_\alpha} = \alpha \cdot split + (1-\alpha) \cdot cover$. We vary α from the discrete set $\{0.1, 0.3, 0.5, 0.7, 0.9\}$ following previous work [5] (RLFDC_{cover} and RLFDC_{split} actually correspond to cases in $\text{RLFDC}_{weight_\alpha}$ where α is set to 0 and 1, respectively).

Table 12 presents the performance of different variants of RLFDC. We also include the performance of RLFDC and the best result-agnostic metric TfD for reference. From the table, the designed features, the network structure, and the training algorithm all positively contribute to the effectiveness of RLFDC. Besides, the comparison between RLFDC and the simple weighted sum baselines substantiates the positive contribution of the entire RL component. In particular, RLFDC_{cover} and RLFDC_{split} are competitive result-agnostic FDC metrics, i.e., each of them outperforms TfD. However, both of them perform worse than RLFDC, demonstrating that both of the two features are important indicators of the FDC value, and combining them can further enhance the FDC prediction accuracy. Note that in the early stage (i.e., with a small number of selected tests) RLFDC_{cover} performs better. We manually investigate some samples and find that in the early stage, since most of the selected tests are passing tests, selecting tests that cover more suspicious elements can effectively help reduce the suspicious value of non-buggy elements and thus help localize the buggy elements. However, in the late stage, covering more elements has a limited effect on more accurately localizing the fault because many ambiguity groups have been formed. Therefore, the effectiveness may be further enhanced by additionally considering how to split the ambiguous groups. The results of $\text{RLFDC}_{simpleNet}$ show the necessity of the embedding layer of the RL model. The state and action features are initially in the different hyperspaces with different scales and the embedding layer helps project them to the same space, which makes learning easier. The results of $\text{RLFDC}_{regularQ}$ show the rationality of choosing double Q-learning as our training algorithm, since regular Q-learning tends to overestimate action values and results in poor performance. The results of $\text{RLFDC}_{weight_\alpha}$ show the contribution of the entire RL component. RLFDC consistently outperforms $\text{RLFDC}_{weight_\alpha}$ with various α. With the RL component, RLFDC automatically learns to keep a good balance between "split" and "cover" based on the current state (i.e., the number of tests and ambiguity groups). With the sense of the current state and direct feedback from fault localization results, RLFDC could learn a more precise FDC measurement and achieve better performance, as evidenced by the results in Table 12. In summary, different components of RLFDC all contribute to enhancing its effectiveness.

Table 12. acc@10 values for the test suites augmented by different variants of RLFDC

FDC Metric	# Tests			
	0	1	5	10
RLFDC$_{split}$	65	92	133	163
RLFDC$_{cover}$	65	**130**	173	195
RLFDC$_{simpleNet}$	65	102	161	198
RLFDC$_{regularQ}$	65	91	163	183
RLFDC$_{weight_0.1}$	65	106	164	195
RLFDC$_{weight_0.3}$	65	97	156	202
RLFDC$_{weight_0.5}$	65	95	160	196
RLFDC$_{weight_0.7}$	65	95	159	196
RLFDC$_{weight_0.9}$	65	93	158	191
RLFDC	65	103	**181**	**203**
TfD	65	84	130	149

6 DISCUSSION

6.1 Oracle Labeling Effort

We here discuss the oracle labeling effort, since the tests selected or generated still require manual effort to label the oracles. Note that developers are required only to verify high-level input-output relationships, which is a straightforward task for them given their intimate knowledge of the project. Initially, RLFDC can be utilized to prioritize which generated tests should be labeled first, based on their estimated impact on improving FDC. Therefore, the labeling effort could be reduced. In the worst-case scenario, each statement might need individual test coverage and differentiation from others, implying that the number of tests requiring labels would equal the number of statements. In this scenario, the overall labeling effort would scale linearly with the number of statements. However, such a scenario is uncommon, and generating such fine-grained tests would demand a highly capable test-generation tool. In such cases, developers might turn to interactive debugging techniques for additional support as discussed in Section 6.2. Manual oracle labeling could reduce the efforts of manual debugging because, with more tests, the faults could be more accurately pinpointed by SBFL which provides a preliminary focus for further investigation. Additionally, there are potential automated and semi-automated techniques that could assist in reducing manual efforts by automatically generating test oracles [12, 19, 32].

6.2 Bug Validation

SBFL only produces the probability of a program element that contains bugs. Some debugging techniques [29, 31, 50] can also help developers pinpoint the fault through interaction with humans. In particular, Li et al. [29] proposes ENLIGHTEN. ENLIGHTEN employs SBFL to form a ranking list of suspicious statements. It then seeks feedback from the developer through a series of queries about suspicious method invocations based on the SBFL results. The human feedback is encoded as extra virtual tests that can further improve SBFL results. Xu et al. [50] propose a probabilistic inference-based debugging technique. They model debugging as a probabilistic inference process where random variables represent the correctness/faultiness of each code statement and variable, and use conditional probability distributions to represent human knowledge and reasoning rules, along with program semantics. Lin et al. [31] propose a feedback-based debugging approach. The system constructs a trace model that records the program's execution and maps out causality relationships among the steps, such as data and control dependencies. Developers can interact

with the debugger by providing feedback on specific steps in the execution trace. Based on the feedback, the system iteratively refines its suspicion about which steps might be erroneous and guides the developers to likely sources of errors.

Our work can be integrated with these debugging techniques to further validate the bug location. Initially, for a given failing test, RLFDC can be employed to select/generate a certain number of additional tests to enrich debugging information. These additional tests can then serve as inputs to enhance SBFL, helping pinpoint faults with greater accuracy. The FL results can be used as guidance to the follow-up bug validation techniques, because SBFL can provide a ranked list of potentially faulty locations in the code, which serves as a preliminary focus for further investigation. For example, ENLIGHTEN [29] could present the enhanced SBFL results to developers and seek human feedback, potentially reducing the number of feedback iterations required. The probabilistic inference-based debugging technique [50] could benefit from the SBFL results by updating the fault probability of a suspicious statement according to its SBFL score, thereby more effectively recommending instances of faulty statements to users. The feedback-based debugging approach [31] could enhance its recommendation mechanism by utilizing SBFL results, specifically by focusing on the step in the trace with the highest SBFL score rather than relying solely on dominance relations. This strategy could potentially reduce the time required for manual checks.

7 THREATS TO VALIDITY

Threats to internal validity mainly come from the implementation of different metrics and the randomness caused by machine learning and evolutionary algorithms. To reduce the threats, we use the implementation of different metrics from previous work [5] and use the published EVOSUITE artifact for EntBug [11] and DDU [40]. Besides, we use five-fold validation to validate our RL model and run the experiments for multiple times to reduce randomness. Although prior work suggests 30 repetitions as a rule of thumb [6], conducting 30 repetitions is resource-intensive and our internal analysis of the result variance from each run indicates minimal fluctuation. Furthermore, the consistently superior performance of RLFDC on tests generated by EVOSUITE and Randoop bolsters our confidence in the advantages of RLFDC.

Threats to external validity mainly come from the subjects, the number of selected tests, the FL technique and the test generation tool. We conduct experiments on the widely-used FL benchmark, Defects4J. We exclude some buggy programs due to deprecated faults or EVOSUITE issues, but the number of programs can still support reliable experiments. We select ten tests to augment the initial test suite following prior work [5] to balance the efforts for human labeling and FL performance. Ochiai-agg is utilized as our FL technique since it is a representative and effective SBFL technique. We use the widely-studied EVOSUITE [13] as our test generation tool following prior work [5, 11, 40] with only modification on the fitness function, which may help for generalization to other search-based test generation tools.

Threats to construct validity mainly come from the metrics we use. To reduce these threats, we adopt two widely-used metrics, acc@n and mAP, to measure the performance of FL, following prior work [5]. Specifically, the former metric focuses on the highest rank of the buggy elements, while the latter metric captures the whole picture. The metrics complement each other and thus provide a comprehensive view when investigating the performance of FL.

8 RELATED WORK

FL is a time-consuming process in software debugging [47], and thus many FL techniques have been proposed [2, 28, 34] to reduce time cost for software debugging, which rely on numerous tests to diagnose faults [22]. Therefore, researchers

have put dedicated efforts in various directions around test for the purpose of FL, e.g., test augmentation [5, 7, 9, 11, 22, 40], test minimization [4, 14, 53, 55], test purification [51] and test prioritization [15, 54].

To improve the performance of FL, test augmentation techniques are proposed to generate or select more tests and enhance the diagnostic capability of the test suite. Some works focus on designing FDC metrics to evaluate the adequacy of an existing test suite for FL, and then selecting or generating more tests under the guidance of the metrics to extend the test suite. Baudry et al. [9] propose a test criterion, named test-for-diagnosis criterion (TfD), to evaluate the diagnosability of test cases in FL. They define dynamic basic blocks (DBB) to describe the indistinguishable statements of a program. They then implement TfD on a test optimization tool utilizing bacteriologic algorithm and aim to improve the accuracy of FL by generating more tests to reduce the size of DBB. Campos et al. [11] take advantage of probability theory concepts and propose an entropy-based measurement to guide test case generation and improve FL. By extending EVOSUITE, they provide a tool named EntBug to generate new test cases. Perez et al. [40] propose a new metric, named DDU, to evaluate a test suite's diagnosability for FL. Compared to previous metrics, DDU takes density, test diversity, and uniqueness into account simultaneously and is combined with EVOSUITE to generate tests to improve FL. An and Yoo [5] propose FDG to measure the fault diagnosability gain of a test case. By leveraging the SBFL results as metric input, it locates the suspicious part of the program and effectively helps the augmentation of test suites for better FL. However, the acquisition of SBFL results needs results of the tests. Therefore, FDG can hardly be applied as a fitness function in most test generation tools. Other works exploit automated test generation from other directions rather than designing FDC metrics. Artzi et al. [7] use directed concolic execution to generate tests. They aim at maximizing the path similarity between the generated tests and the failing tests to better help FL. BUGEX [41] extends EVOSUITE to generate tests to pinpoint the failure, which benefits from runtime properties like state predicates or branches. Jin et al. [21] propose BugRedux to reproduce field failures of real-world programs. They then further extends BugRedux and propose a technique F^3 [22] to generate failing and passing executions similar to the field failure to support its customized FL technique. These techniques are test generation tools assuming there exists automated oracles, while our technique is an FDC measurement which can not only be used to guide test generation, but also be used to select tests to relive the human efforts for labelling oracles.

Besides test augmentation, some researchers study the impact of test minimization, test purification and test prioritization to FL. Yu et al. [55] conduct an empirical study concerning the trade-off between test suite reduction and FL effectiveness, and propose a test reduction strategy that minimizes the impact of reducing tests on FL effectiveness. Gong et al. [14] propose a test case selection strategy based on Diversity Maximization Speedup (DMS) to remove some test cases while maintaining the effectiveness of FL. Yoo et al. [53] regard test minimization problem as a multi-objective problem. They propose a multi-objective formulation and two algorithms based on the concept of Pareto efficient to solve the test suite minimization problem. Inspired by previous work, Alipour et al. [4] propose two types of non-adequate test-case reduction metrics and evaluate the effectiveness of non-adequate reduction for test cases. In their evaluation, they find that this kind of reduction can reduce cost and maintain the similar or even better FL performance. Xuan et al. [51] propose test case purification, which breaks the tests into small fractions to make full use of test oracles. The purified test cases are then used for FL and result in better FL performance. Based on the interplay between FL and test prioritization, some researchers intend to improve FL with test prioritization [15, 54] by prioritizing tests considering entropy theory or ambiguity group reduction.

9 CONCLUSION

We propose RLFDC, the first RL-based metric to measure FDC. Different from previous FDC metrics defined by hand-crafted formulas, RLFDC is a trained RL model. In the training stage, the RL model improves its measuring strategy with feedback from FL results and thus automatically learns a precise measurement. We use three scenarios to evaluate RLFDC, i.e., test selection in human-written tests, test selection in automatically generated tests, and automated test generation. The experimental results show that RLFDC has the best effectiveness in selecting high FDC tests among result-agnostic metrics and has the best effectiveness in guiding EVOSUITE to generate more high FDC tests. We also evaluate our metric in the cross-project scenario and the results show that RLFDC still outperforms the state-of-the-art result-agnostic metric, indicating its generalization capability. The results of the ablation study show that each component of RLFDC positively contributes to its effectiveness. Our work points out a promising direction of combining RL with search-based test generation tools.

10 DATA AVAILABILITY

We make our code, data, and model checkpoints publicly available at **https://github.com/yifan-CodeDir/TOSEM-RLFDC**.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No. 2023YFB4503803 and the National Natural Science Foundation of China (Grant Nos. 62372005, 62232003, 62402482).

REFERENCES

[1] 2020. PyTorch. https://pytorch.org.

[2] 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.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.

[4] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. 2016. Evaluating non-adequate test-case reduction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 16–26.

[5] Gabin An and Shin Yoo. 2022. FDG: a precise measurement of fault diagnosability gain of test cases. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 14–26.

[6] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*. 1–10.

[7] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*. 49–60.

[8] Aritra Bandyopadhyay and Sudipto Ghosh. 2011. Proximity based weighting of test cases to improve spectrum based fault localization. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 420–423.

[9] Benoit Baudry, Franck Fleurey, and Yves Le Traon. 2006. Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on Software engineering*. 82–91.

[10] Borja Calvo and Guzmán Santafé Rodrigo. 2016. scmamp: Statistical comparison of multiple algorithms in multiple problems. *The R Journal, Vol. 8/1, Aug. 2016* (2016).

[11] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. 2013. Entropy-based test generation for improved fault localization. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 257–267.

[12] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.

[13] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[14] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. 2012. Diversity maximization speedup for fault localization. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 30–39.

[15] Alberto Gonzalez-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan JC van Gemund. 2011. Prioritizing tests for fault localization through ambiguity group reduction. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 83–92.

[16] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. 2010. Test input reduction for result inspection to facilitate fault localization. *Automated software engineering* 17 (2010), 5–31.

[17] Hado Hasselt. 2010. Double Q-learning. *Advances in neural information processing systems* 23 (2010).

[18] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 464–475.

[19] Soneya Binta Hossain, Antonio Filieri, Matthew B Dwyer, Sebastian Elbaum, and Willem Visser. 2023. Neural-based test oracle generation: A large-scale evaluation and lessons learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 120–132.

[20] Ronald L Iman and James M Davenport. 1980. Approximations of the critical region of the fbietkan statistic. *Communications in Statistics-Theory and Methods* 9, 6 (1980), 571–595.

[21] Wei Jin and Alessandro Orso. 2012. Bugredux: Reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 474–484.

[22] Wei Jin and Alessandro Orso. 2013. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 213–223.

[23] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[24] Lou Jost. 2006. Entropy and diversity. *Oikos* 113, 2 (2006), 363–375.

[25] René Just. [n. d.]. Defects4J Github repository. https://github.com/rjust/defects4j. Accessed on: 2024-08-18.

[26] 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.

[27] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1424–1446.

[28] 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.

[29] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering*. 82–92.

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

[31] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 393–403.

[32] Zhongxin Liu, Kui Liu, Xin Xia, and Xiaohu Yang. 2023. Towards more realistic evaluation for neural test oracle generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 589–600.

[33] 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.

[34] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[35] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[36] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[37] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[38] Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate" unknown" faults. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 691–700.

[39] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[40] Alexandre Perez, Rui Abreu, and Arie Van Deursen. 2017. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 654–664.

[41] Jeremias Röβler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating failure causes through test case generation. In *Proceedings of the 2012 international symposium on software testing and analysis*. 309–319.

[42] Sina Shamshiri, René Just, José 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering*

(ASE). IEEE, 201–211.

[43] 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.

[44] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[45] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.

[46] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[47] 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.

[48] Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. 2022. A universal data augmentation approach for fault localization. In *Proceedings of the 44th International Conference on Software Engineering*. 48–60.

[49] Huaqing Xiong, Lin Zhao, Yingbin Liang, and Wei Zhang. 2020. Finite-time analysis for double Q-learning. *Advances in neural information processing systems* 33 (2020), 16628–16638.

[50] Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu, and Baowen Xu. 2018. Debugging with intelligence via probabilistic inference. In *Proceedings of the 40th International Conference on Software Engineering*. 1171–1181.

[51] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 52–63.

[52] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[53] Shin Yoo and Mark Harman. 2010. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software* 83, 4 (2010), 689–701.

[54] Shin Yoo, Mark Harman, and David Clark. 2013. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on software engineering and methodology (TOSEM)* 22, 3 (2013), 1–29.

[55] Yanbing Yu, James A Jones, and Mary Jean Harrold. 2008. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th international conference on Software engineering*. 201–210.

[56] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 445–455.

[57] 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.

Received 14 March 2024; revised 26 November 2024; accepted 17 December 2024